TB²F: Tree-Bitmap and Bloom-Filter for a Scalable and Efficient Name Lookup in Content-Centric Networking

Wei Quan*, Changqiao Xu*, Athanasios V. Vasilakos†, Jianfeng Guan*, Hongke Zhang§ and Luigi Alfredo Grieco‡

*State Key Laboratory of Networking and Switching Technology,
Beijing University of Posts and Telecommunications, Beijing, China
†Kuwait University, Kuwait City, Kuwait
§Beijing Jiaotong University, Beijing, China
‡Politecnico di Bari, Bari, Italy
Email: *{quanwei, cqxu, jfguan}@bupt.edu.cn, †vasilako@cs.ku.edu.kw, §hkzhang@bjtu.edu.cn, ‡a.grieco@poliba.it

Abstract—Content-Centric Networking (CCN) is an entirely novel networking paradigm, in which packet forwarding relies upon lookup operations on content names directly instead of fixed-length host addresses. The unique features of CCN names, *i.e.,* variable length, huge cardinality, and hierarchical structure, introduce new challenges that could hinder the deployment of such a new architecture at the Internet scale. In this paper, we make an in-depth study of characteristics of large-scale CCN names, and propose a simple yet efficient CCN-customized name lookup engine (named by TB²F), which capitalizes the strengths of Tree-Bitmap (TB) and Bloom-Filter (BF) mechanisms, while counteracts their main limitations. To this end, TB²F splits CCN prefix into a constant size T-segment and a variable length B-segment with a relative short length, which are treated using TB and BF, respectively. Furthermore, an optimal length of the T-segment is found to improve the lookup efficiency. Experimental comparisons with respect to the reference Name Prefix-Trie and Bloom-Hash have been also carried out. The results show that TB²F properly configured has good scalability and efficiency by (i) speeding up lookup operations and reducing the false positive rate with respect to Bloom-Hash; (ii) requiring less memory than Name Prefix-Trie; (iii) achieving a low overhead in updating operations in the large scale case.

Index Terms—Name Lookup; Content-Centric Networking; Tree-Bitmap; Bloom Filter

I. INTRODUCTION

Motivated by significant changes witnessed in the usage of the Internet, Content-Centric Networking (CCN) [1] emerges as a novel Information-Centric Networking (ICN) [2] [3] paradigm, focusing on *what* rather than *where* the content is. Named Data Networking (NDN) [4], as a derived project of CCN, is developing worldwide. In contrast to the *host-centric* IP rationale, CCN adopts hierarchical content names to rule forwarding operations instead of fixed host addresses [5]. In this way, it becomes potentially possible to get rid of "*host IP address*" in traditional networking primitives and to obtain several advantages such as: (i) native support to multicast and mobility; (ii) content level security; (iii) reduction of servers' load, also thanks to in-network caching; (iv) simplified interop-erability among content distribution systems and applications. However, it is also facing with some challenges to be solved for the practical deployment at the Internet scale.

In CCN, each router/node is equipped with three specific components: *Forwarding Information Base* (FIB), *Pending Interest Table* (PIT) and *Content Store* (CS) [1]. The FIB is used to specify the faces which packets can be forwarded through; the PIT holds all "not yet satisfied" requests that have forwarded towards potential data sources but have not receive a response; the CS is the cache memory, where a copy of contents retrieved in the past are stored to answer future requests. When a node is willing to retrieve a content, it sends an *Interest* packet, indicating the name of the desired content. At each hop, forwarding decisions depend upon the outcome of lookup operations of the requested name in the FIB, PIT and CS tables. In case there is no matched content in the CS, the PIT is updated to keep track of the face the *Interest* arrived from. Then, the *Interest* is forwarded to the next hop (if required) after having looked up the FIB to search for the most appropriate outgoing face(s). Once a router/node has the content that matches the requested *Interest*, *Data* packet(s) will be returned back to the requesting node in the reverse path activated by the *Interest*. Besides, the CCN-based router may cache the *Data* packet(s) in CS, making itself as a provider for the following *Interest* requests.

Recently, some researchers have argued that current router technologies cannot meet the requirements of CCN [6][7][8]. Unlike fixed-length IP addresses (*i.e.*, 32 bits for IPv4 and 128 bits for IPv6), content names in CCN are *variable length* strings with a *hierarchical structure*, consisting of a sequence of *delimited components*. These emerging features of CCN names bring several unprecedented challenges in a practical large-scale use. Firstly, longest prefix matching (LPM) in CCN must match a prefix at the end of one component of the name, rather than at any digit as in IP address. Secondly, more complex lookup mechanisms are needed to adapt to the

variable-length of content names. Thirdly, CCN forwarding tables (*i.e.*, FIB) will be much larger than IP ones, because the cardinality of the set of content names can be many orders of magnitude larger than the one of IP addresses. Google has reported that the number of URLs exceeded to 1 trillion in 2008 [9]. Nevertheless, there are only about 800 million routable hostnames for all websites by the end of 2013 [10]. Although LPM has been heavily studied for IP lookup, most of proposed solutions become inefficient if applied to CCN names directly.

To tackle this inextricable challenge, the scalable and efficient name lookup solutions are researched herein, aiming at paving the way to the practical development of CCN routers at the Internet scale. The main contributions are as follows:

(i) We conceived a totally novel name lookup engine, TB^{2}F, which leverages consolidated Tree-Bitmap (TB) [11] and Bloom-Filter (BF) [12] solutions. In TB^{2}F, the CCN prefix is split into a constant size *T-segment* and a variable length *B-segment*. Due to the high aggregation of T-segments, T-segments can be processed using TB structure time-efficiently. Instead, B-segments will be handled using BF structure space-efficiently. In detail, TB^{2}F is made of a novel hybrid Data Structure ***TB^{2}F-DS***, a Parallel Lookup process ***TB^{2}F-PL***, and a Differentiated Update scheme ***TB^{2}F-DU***.

(ii) We made practical considerations based on large-scale names datasets and proposed a theoretical analysis of TB^{2}F in terms of computational complexity, false positive rate (*fpr*), aggregation and updating ratio. Based on such an analysis, we deduce that these metrics depend on the length of T-segment directly. Further, we proposed a methodology for discovering the optimal length of T-segments, able to minimize the lookup time subject to an affordable memory cost and *fpr*.

(iii) We performed extensive experimental evaluations to validate the TB^{2}F approach in comparison to two state-of-the-art solutions, namely, Name Prefix-Trie [13] and Bloom-Hash [14]. The results illustrate that, if properly configured, TB^{2}F enables to capitalize the strengths of TB and BF by (i) speeding up the lookup and reducing the false positive rate with respect to Bloom-Hash; (ii) consuming less memory with respect to Name Prefix-Trie; (iii) reducing the overhead of updating with respect to both two solutions.

The rest of the paper is organized as follows: Section II introduces the problem statement and details our novel name lookup solution. Some practical considerations and theory analysis for the proposed solution are presented in Section III. Extensive experiments are compared and analyzed in Section IV. Section V discusses the related work. Finally, concluding remarks and future works are presented in Section VI.

II. PROBLEM STATEMENT AND OUR PROPOSAL: TB^{2}F

A. Problem statement

We consider a practical scenario: there are at least 10^7 names stored in the forwarding table, which should support millions of lookups per second [15]. *How to design a lookup engine for CCN names, which supports to store massive entries and enables to lookup at high speed in such large-scale*

TABLE I
KEY NOTATIONS

Notations	Definition
$\mathcal{N}, m$	Set of requesting names, and its total number;
$\mathcal{P}$	Set of prefixes entries stored in forwarding table (FIB);
$\mathcal{D},\mathcal{Q}$	Data structure and lookup algorithm for $\mathcal{P}$;
$\mathbb{M}(\cdot)$	Memory space function;
$\mathbb{T}(\cdot)$	Lookup time consumption function;
$\mathbb{U}(\cdot)$	Updating overhead function;
$\mathbb{F}(\cdot)$	False positive rate function;
$q_i^T,\zeta_{q_i}^T$	T-segment for a requested name q_i, and its length;
$q_i^B,\zeta_{q_i}^B$	B-segment for a requested name q_i, and its length;
τ,t_0	Request time interval, and observing time instant;

scenarios? To formalize the problem, we consider herein the case of a router that receives m *Interest* messages, within a time interval τ, each one asking for a given content name, then the requesting rate is m/τ. Table I lists key notations in this paper. Accordingly, let $\mathcal{N}$ denote the set of names, $\{n_1, n_2, n_3 \cdots n_m\}$, that were requested at the router during τ, and $\mathcal{P}$ be the set of total prefix entries, $\{p_1, p_2, p_3 \cdots p_k\}$, pre-stored in the forwarding table (*e.g.*, FIB) at the observing time instant of t_0. Let $\mathcal{D}$ denote the data structure adopted for storing $\mathcal{P}$, then the occupying memory space for $\mathcal{P}$ will be expressed as a function of $\mathcal{D}$ and $\mathcal{P}$, which is $\mathbb{M}(\mathcal{D},\mathcal{P})$.

Starting from this premise, the lookup problem can be translated in a process of membership queries: assuming we want to request the content by name q, $\forall q \in \mathcal{N}$, the longest prefix in $\mathcal{P}$ for q should be determined with a name lookup algorithm, $\mathcal{Q}$, and the corresponding forwarding face(s) should be returned in a short time interval $\mathbb{T}(\mathcal{N},\mathcal{P},\mathcal{D},\mathcal{Q})$ (shorter than m/τ); if no prefix matches, the default face is returned. Besides, we will refer to $\mathbb{U}(\mathcal{D})$ and $\mathbb{F}(\mathcal{Q})$ to denote the overhead of updating for $\mathcal{D}$, and be the false positive rate during the processing of $\mathcal{Q}$, respectively.

Our aim is to design a fairly simple $\mathcal{D}$ and an efficient $\mathcal{Q}$ for minimizing the lookup time as well as improving its scalability for a large-scale CCN name space (*i.e.*, 10^7 at least). In a more formal way, this problem can be expressed as follows:

$$\min_{\mathcal{N},\mathcal{P},\mathcal{D},\mathcal{Q}} \mathbb{T}(\mathcal{N},\mathcal{P},\mathcal{D},\mathcal{Q})$$
$$\text{subject to } |\mathcal{N}| >> 1, \ |\mathcal{P}| >> 1,$$
$$\mathbb{M}(\mathcal{D},\mathcal{P}) \leqslant \sigma, \qquad (1)$$
$$\mathbb{U}(\mathcal{D}) \leqslant \delta,$$
$$\mathbb{F}(\mathcal{Q}) \leqslant \varphi.$$

where σ is the maximal available memory, δ and φ are acceptable thresholds on updating overhead and false positive rate, respectively.

B. TB^{2}F-DS: data structure

In this section, we present a novel hybrid data structure for CCN name lookup, named by TB^{2}F-DS, which is motivated by the following three practical observations:

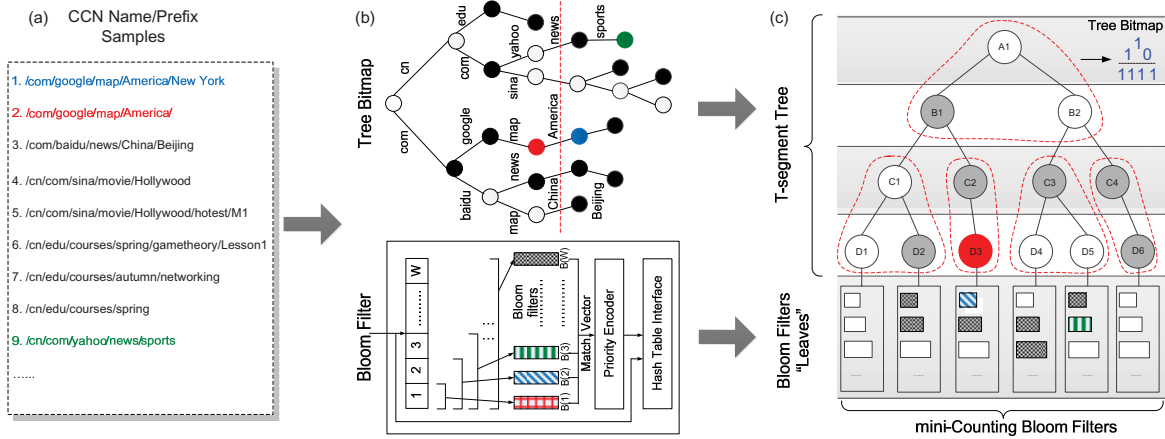(1) Tree-Bitmap is a compressed multi-bit Trie data structure which supports fast lookups. However, its performance

Fig. 1. Framework for TB²F-DS: (a) CCN names samples; (b) Tree-Bitmap and Bloom Filters; (c) Example of TB²F-DS with Stride Length of 2.

degrades linearly as the tree depth increases and this drawback makes TB unsuitable for unbounded CCN names.

(2) Bloom-Filter is a space-efficient probabilistic data structure which supports set membership queries. However, the false positive rate of a BF could become unacceptable in presence of a large set of names (as in CCN).

(3) The structure of CCN names is hierarchical. Statistically, if two different names are taken the probability that they will differ in the first components is much lower than in the last ones (detailed in Sec. III (c)). This feature can be fruitfully exploited by processing the first components of names using TB to avoid the high memory cost. Contrariwise, the rest of the name is better fitted to BF.

To overcome the drawbacks of TB and BF as well as capitalize their strengths, we propose to split each CCN name prefix into two parts. The first part, named by *T-segment*, is bounded with a certain number of components, and the rest belongs to the second one, named by *B-segment*. For ease of description, we first give a definition of *Split Level*.

Definition 1 (*Split Level*): For a CCN name n_i, a split scheme $\mathcal{L}$ is adopted to split n_i into an ordered set of two segments, T-segment n_i' and B-segment n_i''. We define Split Level (SL) for $\mathcal{L}$ as the number of components in n_i'.

Example: If the SL for a given $\mathcal{L}$ is 3, the name "a/b/c/d/e/1" will be split into "a/b/c" as its T-segment and "d/e/1" as its B-segment.
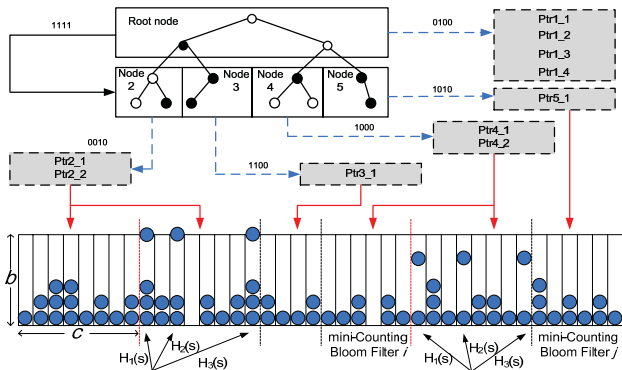
The framework of TB²F-DS is illustrated in Fig.1. Fig.

Fig. 2. Diagram of TB²F-DS.

1(a) lists a set of CCN name samples. Fig. 1(b) presents the traditional data structures of TB and BF respectively. Fig.1(c) shows a simple diagram of TB²F-DS. In TB²F-DS, all T-segments are stored using TB, constructing the **T-segment Tree**, while B-segments are inserted into a sequence of mini Counting BFs (mCBFs) [12][16], each of which is linked to the corresponding node of T-segment Tree. The association between T-segment Tree and mCBFs is implemented using a pointer for "children" of one node. In other words, mCBFs become leaves of T-segment Tree, which makes it possible to not bring much extra complexity in implementing TB²F-DS.

The diagram of TB²F-DS is detailed in Fig. 2. In the T-segment Tree, there are two kinds of bitmaps for each node. One bitmap is for all the internally stored prefixes and the other one is for the external pointers. All the children of a given node are stored contiguously, which allows us to use just one pointer for all of them (*i.e.*, each child node can be found using an offset from the single pointer)[1]. This yields a remarkable reduction of the memory requirements. On the other hand, k hash functions are adopted to compute the hash value for B-segments, which guides to fill the corresponding mCBF, equipped with c counters and b bits for each counter. In one extreme case, TB²F-DS supports to use a global CBF for all B-segments, which can be implemented more easily. The direct benefits of using many mCBFs (with a simple T-segment associated allocation scheme) instead than a unique CBF are to reduce the scale for each mCBF, thus alleviating the difficulty of finding perfect hash functions that remain valid for a universal name space and to avoid the chaos of association between T-segment and B-segment.

In this structure, the SL parameter plays a fundamental role on the efficiency of TB²F-DS. Firstly, the depth of the T-segment tree is bounded by the SL, which, as a consequence, directly benefits the processing time and occupying space. Secondly, all names with less than SL components will not be mapped in an mCBF at all. It can reduce the number of items inserted into the mCBFs, accordingly, lowers the *fpr*. Furthermore, it is worth noting that hash computing for a long

---

[1]It is noted that T-segment Tree is not necessary to be a binary tree. An encoding method can be used to resolve the issue of unbounded children.
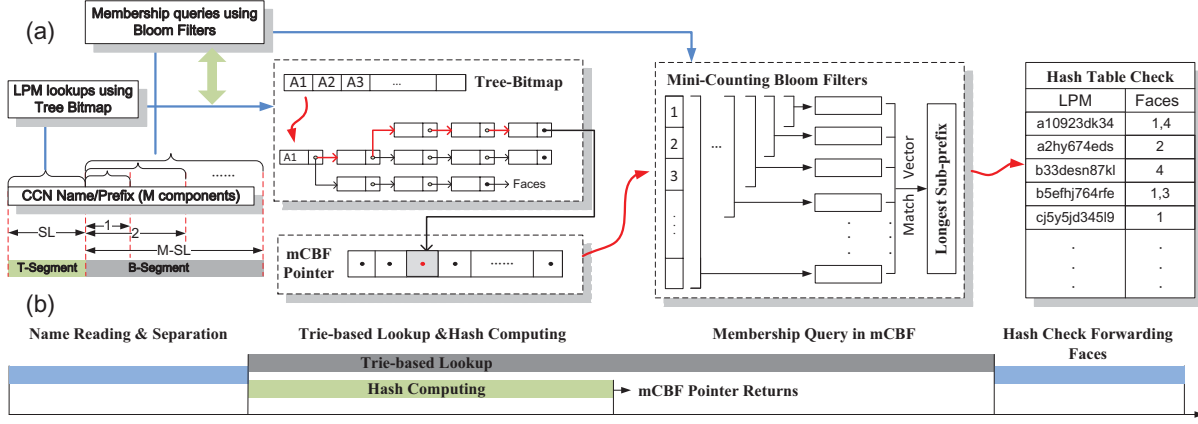
Fig. 3. (a) Diagram of TB^{2}F-PL; (b) Compositions of time consuming in lookup process.

string is expensive, a cut by SL components may improve processing speed in mCBFs.

To sum up, TB^{2}F-DS provides a simple but powerful data structure for CCN names lookup. We get three promising promotions: (1) TB^{2}F-DS transfers the long-unbounded names to be short-bounded, suitable for using TB; (2) TB^{2}F-DS accelerates the hash computing and lowers the *fpr* in mCBFs; (3) TB^{2}F-DS does not bring much extra space consumption, if configure SL properly, which makes TB^{2}F space efficient. The next section will provide an experimental analysis of the suitability of TB^{2}F-DS to large-scale CCN scenarios.

C. TB^{2}F-PL: parallel lookup process

TB^{2}F enables to provide fast processing thanks to a parallel lookup process (TB^{2}F-PL). It is a fact that frequently hash computing for strings is a time-consuming operation in CBF. To reduce the lookup time consumption, an available solution is to preprocess the hash value of each string before using it. In CCN, all hierarchical names are carried in the *Interest* and *Data* packets, and different routers may adopt different hash functions. Thus, it is very difficult to obtain the hash values of each substring before the lookup starting. Fortunately, TB^{2}F-PL allows to process two segments in parallel due to the decoupling of T-segment and B-segment. Thus, hash computing for all possible substrings of B-segment can be pre-processed in parallel during the T-segment matching. To some extent, this operation ensures the hash values of B-segment are prepared well when the B-segment's lookup starts, which can save the total lookup time.

The parallel lookup process is detailed in Fig. 3(a). Assuming $\forall q_i$ in $\mathcal{N}$ with a length ζ_{q_i} (here we use the length to indicate the number of components of prefix), we will refer to as q_i^T and q_i^B to indicate its T-segment and B-segment with a length of $\zeta_{q_i}^T$ and $\zeta_{q_i}^B$, respectively. First, q_i^T is looked up along the T-segment Tree. In the meantime, the hash computing for all substrings of q_i^B is conducted. If the longest matching prefix p_i' for q_i^T is found and its length is shorter than the SL, the forwarding face will be returned. In this case, the lookup terminates. On the other hand, if the length of p_i' is equal to SL, one corresponding pointer to an mCBF is returned. With the knowledge of hash values of substrings, the q_i^B is then looked up in this mCBF. After the membership queries for these

---

Algorithm 1: The insertion of an entry in TB^{2}F-DU

```
1:  procedure InsertEntry(Prefix x)
2:      (x^T, x^B)←GetSeparation(Prefix x); //split the prefix
3:      if mCBF_ptr ←Lookup(x^T) then
4:          Locate the split mCBF by mCBF_ptr;
5:          Position in mCBF: j←h_i(x^B); //hash computing
6:          Increment counter for j: C_j←C_j+1; // updating CBF
7:      elseif Node_ptr ←Lookup(x^T) then
8:          Locate the leaf multi-bit node contains an insertion;
9:          Bits in internal Bitmap: b_i←1; //updating TB
10:         Bits in external Bitmap: b_e←1;
11:         if x^B ≠ NULL;
12:             Locate the new allocated mCBF;
13:             Position in mCBF: j←h_i(x^B); //hash computing
14:             Increment counter for j: C_j←C_j+1;
15:         end if
16:     end if
17: end procedure
```

possible substrings, a match vector is returned, which indicates the longest matching substring p_i'' for the q_i^B. Combining with p_i', we can easily obtain the longest matching prefix $p_i \leftarrow \langle p_i', p_i'' \rangle$ for the entire name q_i. At last, a hash check for the next hop forwarding face is conducted. Note that default faces will be returned if no match is returned. A detailed timeline for time consumptions of TB^{2}F-PL is illustrated in Fig. 3(b). It will be used as a ground for the theoretical and experimental analysis carried out in next sections.

D. TB^{2}F-DU: differentiated update scheme

TB^{2}F provides flexible update process with low overhead thanks to a differentiated update scheme (TB^{2}F-DU), which takes full advantage of the low updating frequency of T-segment Tree as well as the simplified hash computing in mCBFs. The update process is triggered when new entries arrive in the FIB and replace the expired ones. This updating operation in TB^{2}F-DU includes insertion and deletion operations in both the T-segment Tree and the mCBFs.

Entry insertion (Algorithm 1): When a new entry has to be added to the TB^{2}F-DS, three cases have to be considered: (i) only the T-segment has to be updated (*i.e.*, the name has less than SL components); (ii) the entry just requires an update of one mCBF (*i.e.*, the T-segment Tree already maps the former components of the name); (iii) both the T-segment Tree and the mCBF have to be updated. Accordingly, upon an insertion, if the length of the inserting name is less than SL, we just update

Algorithm 2: The deletion of an entry in TB^{2}F-DU
1: **procedure** DeleteEntry(*Prefix x*)
2:    $(x^T, x^B) \leftarrow$ GetSeparation(*Prefix x*); //split the prefix
3:    **if** mCBF_ptr $\leftarrow$ Lookup(x^T) **then**
4:        Locate the split mCBF by mCBF_ptr;
5:        Position in mCBF: $j \leftarrow h_i(x^B)$; //hash computing
6:        Increment counter for j: $C_j \leftarrow C_j$-1; //updating CBF
7:    **elseif** Node_ptr $\leftarrow$ Lookup(x^T) **then**
8:        **if** x^B == NULL;
9:            Locate the leaf multi-bit node contains a deletion;
10:           Bits in internal Bitmap: $b_i \leftarrow 0$; //updating TB
11:           Bits in external Bitmap: $b_e \leftarrow 0$;
12:       **end if**
13:   **end if**
14: **end procedure**

TABLE II
DATASET COLLECTIONS

URL Datasets	Value	Names Dataset	Value
Baidu URLs	$\sim 1,960,000$	Crawling URLs	$\sim 14,800,000$
DMOZ URLs	$\sim 3,240,000$	CCN names[3]	$\sim 200,000,000$

the T-segment Tree, no need to make any operation in mCBFs. Otherwise, the T-segment Tree is queried to discover whether a possible match does exist. If the match exists then we are in the second case otherwise we are in the third one. In the second case, it is just required to update the mCBF (using an addition operation on the CBF) pointed by the leaf node of the T-segment Tree. Instead, if we are in the third case, the T-segment Tree is renewed to account for the new entry and the corresponding mCBF is updated with an addition operation.

Entry deletion (Algorithm 2): In parallel with the insertion, three cases are considered in the deletion process, (i) the updating of one mCBF when the length of name is bigger than SL; (ii) the updating of the T-segment Tree when the length of name is smaller than SL and (iii) the updating of both them when one T-segment has no B-segment following after deleting a B-segment. In the first case, one corresponding mCBF for a given x should be found before, then it carries out a deletion in mCBF by decreasing the associated counters by 1. In particular, if one corresponding mCBF cannot be returned, we believe the name to be deleted does not exist in TB^{2}F-DS originally. In the second case, one leaf multi-bit node in T-segment Tree should be updated by renewing both the internal and external bitmaps by setting the associated bits to zero, which is detailed in reference [11]. The third case can be seen as a special situation of case one, a further updating should be conducted in T-segment Tree because there is no B-segment following the T-segment.

Due to the differentiated updating operations, it can improve the updating efficiency and avoid unnecessary overhead in certain cases. We remark that the depth of the T-segment Tree is bounded by SL and that our experimental findings on huge datasets demonstrates that the first levels of the name tree are relative steady and unlikely to change frequently. As a consequence, the updating ratio can be kept under control using a proper SL. This allows us to avoid a fair complex tree updating process, and transfer it to the low-overhead updating in mCBF. Furthermore, the length of entries in the mCBF is
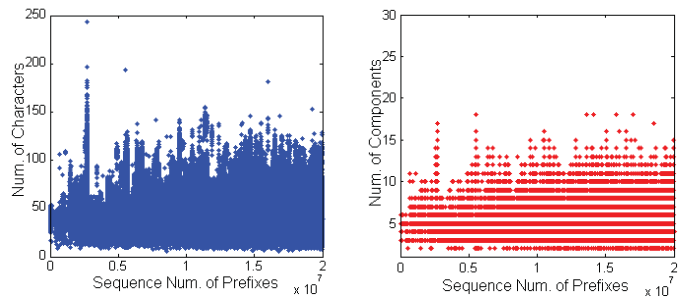

Fig. 4.   (a) Number of characters; (b) Number of components for each name.

shortened by SL, this also benefits the overhead of updating in mCBFs. In the sequel of the work, all the aforementioned qualitative evaluations will be proved experimentally to crisply highlight the relevance of TB^{2}F to CCN scenarios.

III. THEORY ANALYSIS BASED ON PRACTICAL CONSIDERATIONS

Is the proposed TB^{2}F fitted to large scale CCN scenarios? And how to choose an optimal SL for TB^{2}F? To answer these questions practically, we make an in-depth study of the characteristics of CCN names. We recognize that the set of Internet URLs is a subset of CCN names. Thus, we collected a large scale real-world URLs as our raw materials, which contain 2×10^7 URLs with about 1.3×10^8 components. Then, we processed them to build the experimental CCN-names dataset according to the reference [5]. Finally, the total dataset contains about 2×10^8 CCN names, occupies 9.725 GB. Table II shows the summary information of dataset collections. In the following, we will make a further analysis for the statistical features of names.

We seek to explore the shape and statistical characteristics of two very relevant features of the name space: (1) the number of the components $\ell(u_i)$ for each name u_i and (2) the number of the characters $\varphi_k(u_i)$ for the k-th component in u_i. Both of them are core indicators for the features of CCN names and dramatically affect lookup operations. Fig. 4 shows the characteristics of our dataset in terms of number of characters and components for each CCN name, respectively, where the names are reported with the same sequence as the crawling process provided them. Furthermore, their statistical distributions are shown in Fig. 5, some essential findings are observed in what follows.

Observation 1: *In Fig. 5(a), it can be argued that more than 91% components have a length smaller than 10 characters, and nearly 40% components with a length of 3 characters.*

This feature means short components are popular to be used in CCN name, whereas long components exist but are rare. An intuitive explanation is that short components are easier to be remembered by human, and more convenient to be used.

Observation 2: *In Fig. 5(b), we observe that nearly 40% names have a length less than 4 components, and nearly 75% names have a length less than 5 components.*

This feature means if we consider the case of 50% names, the threshold of the length should lie in the set of [4,5].
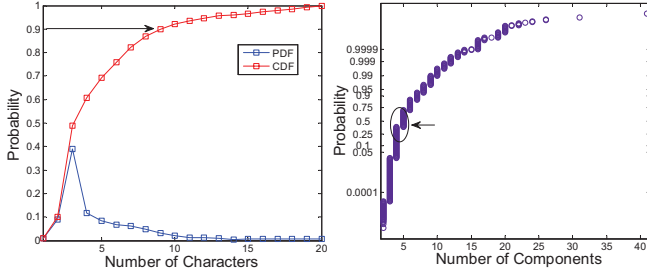
Fig. 5. The distribution of (a) Number of characters for each component; (b) Number of components for each name.

Fig. 6. (a) False positive rate; (b) Aggregation ratio of prefixes.

Based on these primary considerations, we deepen our analysis to assess the metrics in terms of *(a) computing complexity, (b) false positive rate, and (c) aggregation and updating ratio*, dependent on SL.

(A) Computing complexity: The lookup complexity in a trie is directly dependent on the depth of the trie [17], thus, the average time complexity of T-segment lookup achieves a reduction from $O\left(\bar{\zeta}_{q_i}\right)$ to $O\left(\bar{\zeta}_{q_i}^T\right)$, and is $O\left(SL\right)$ in the worst case. As for the B-segment, the membership query complexity is $O\left(1\right)$ in theory [14]. However, it is a fact the hash computing for each prefix of B-segment in CCN is expensive, which is dependent on the length of substrings [18]. Since it requires to compute for each level substring of q_i^B for membership queries, the computing complexity is $O\left(\bar{\zeta}_{q_i}^B \cdot \left(\bar{\zeta}_{q_i}^B - 1\right)/2\right)$. If the hash computing for each substring is independent, the time complexity will become $O\left(\bar{\zeta}_{q_i}^B\right)$ in the worst case relying on concurrent processing for all substrings. Let κ be $\max\left(SL, \bar{\zeta}_{q_i}^B\right)$. Since T-segment matching and B-segment hashing can be processed in parallel, the total time complexity for TB^{2}F-PL is $O\left(\kappa + 1\right)$ in theory.

The updating process includes entry deletions and entry insertions. Different from lookup operations, this process only conducts hash computing for the string to be deleted or inserted, and need not to care about its substrings in the CBF. For TB^{2}F-DU, the hash computing for the updating entry costs $O\left(\bar{\zeta}_{q_i}^B\right)$. The computing complexity of locating the proper leaf in TB is $O\left(SL\right)$ in the worst case. The update complexity in CBF is $O\left(1\right)$. Due to the parallel process, the time complexity for TB^{2}F-DU is also $O\left(\kappa + 1\right)$ in the worst case.

(B) False positive rate: In original BF-solutions, all names should be inserted into the filters. However, in TB^{2}F-DS not all prefixes need to be stored using BF, but only the B-segments. Based on the theory in [14], the false positive rate is the probability that all k bits that it hashes to are 1. We derive a new expression for false positive rate that is

$$p = \left(1 - \left(1 - \frac{1}{C}\right)^{k\mu n}\right)^k \approx \left(1 - e^{-k\mu n/C}\right)^k \quad (2)$$

where C is the number of buckets in filters, k is the number of hash functions, n is the total number of prefixes, and μ is the ratio of B-segments over the entire space of prefixes.

Formula (2) presents a direct relation between the false positive rate p and the ratio μ on condition of n being a known constant. Fig. 6(a) shows the varying of *fpr* at different filter sizes in the case of n=10 million and k=3 in theory.
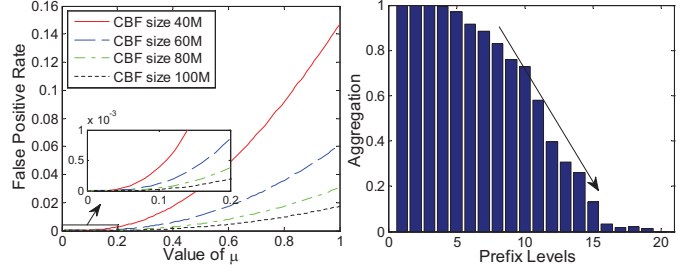
Observation 3: *From Fig. 6(a), it is observed that the false positive rate increases as the value of the ratio μ. Combining with formula (2), a higher ratio μ will bring a higher false positive rate if the total number of name prefixes is a constant.*

In TB^{2}F, only if the length of one name is bigger than SL, its B-segment would be inserted into CBFs. According to the Observation 2, different values of SL will bring different ratio μ. If the threshold of SL is 5, then only 25% names have corresponding B-segments. To this end, an optimal choice of SL may lead to a suitable μ, that drives a small false positive rate within an acceptable range.

(C) Aggregation and updating ratio: For ease of description, we first give a formal definition for aggregation ratio for a set of names.

Definition 2 (Aggregation ratio): Given a name set $\mathcal{S}$, in which the number of names is $|\mathcal{S}|$, and a prefix length l, $\forall l > 0, l \in \mathrm{N}^+$, we can calculate the total number of all different prefixes in $\mathcal{S}$ with a length l, that is $\mathcal{C}\left(l\right)$. Then, the aggregation ratio is defined as $1 - \mathcal{C}\left(l\right)/|\mathcal{S}|$.

Intuitively, the aggregation ratio represents the probability of many names sharing one same prefix, and it is an important indicator for the space compression of prefixes in the tree-based data structure. Fig. 6(b) shows the aggregation ratio as well as its rate of change greatly relying on the prefix length.

Observation 4: *According to Fig. 6(b), the aggregation ratio of the prefixes decreases with the length of prefixes. It is larger than 95% when the length of prefixes $l \in [1, 5]$, $l \in N^+$, and it starts decreasing quickly when $l > 6$, $l \in N^+$.*

This feature implies that it would have a high aggregation ratio in the first several components of CCN names. It will bring a great space saving in TB structure and provide benefits for the low updating probability of TB elements if the SL $\leqslant 5$.

Proposition 1: Given a tree $\mathcal{R}\left(\mathcal{P}_1\right)$ for name prefixes set $\mathcal{P}_1$, a new set of name prefixes $\mathcal{P}_2$ brings an updating for $\mathcal{R}\left(\mathcal{P}_1\right)$ to build $\mathcal{R}\left(\mathcal{P}_1 \vee \mathcal{P}_2\right)$. The updating probability for $\mathcal{R}\left(\mathcal{P}_1\right)$ increases with the depth of the tree.

Proof: Let $\mathcal{A}\left(t\right)$ denote the aggregation ratio of the prefixes set $\mathcal{P}^t$ with a length of t, $\mathcal{B}\left(t\right)$ be the updating probability for $\mathcal{P}^t$. Based on Observation 4, $\forall \varepsilon > 0$, we have $\mathcal{A}\left(t\right) \geqslant \mathcal{A}\left(t + \varepsilon\right)$. Given a new name prefix p_x, only if $p_x \notin \mathcal{P}^t$, one updating would occur. Since a higher $\mathcal{A}\left(t\right)$ means a bigger probability of $p_x \in \mathcal{P}^t$, thus it obtains a lower updating probability. Therefore, for $\forall \varepsilon > 0$, $\mathcal{B}\left(t\right) \leqslant \mathcal{B}\left(t + \varepsilon\right)$. It concludes our proof. $\square$

Based on the above observations and analysis, we provide

a methodology to estimate an optimal reference size for SL, subject to all the constraints of the problem we are dealing with: (i) to minimize the false positive rate, the SL should be as much as possible, which is restricted in $[4, \infty)$ (from *Observation 3*); (ii) to achieve a low memory cost as well as a low updating ratio for T-segment Tree, the SL should be as small as possible, which is restricted in the range $[1, 5]$ (from *Observation 4* and *Proposition 1*); (iii) to obtain a reasonable time complexity and memory cost, SL would better lie in the set $[4, 5]$ (from *Observation 2*). All these considerations guide us to conclude an inference for the estimation of SL.

Inference: *To maximize the benefit of TB^{2}F, thus, minimize the lookup time on the restriction of the affordable memory cost as well as acceptable false positive rate and updating overhead, an optimal size of SL is concluded with a set-intersection operation, which is suggested in the set of $[4, 5]$.*

In next section, experimental comparisons will analyze the performance tendency along with different SL values and further crisply verify the relevance of this inference.

IV. EXPERIMENTS EVALUATION

To evaluate the scalability and efficiency of the TB^{2}F, we carefully selected two state-of-the art solutions, namely Name Prefix-Trie [13] and Bloom-Hash [14] for comparisons. The two candidates are typical to represent the Trie-based and BF-based solutions, respectively. In the experiments, we focus on comparing our solution with both of them in terms of four metrics: *(i) memory cost, (ii) lookup time consumption, (iii) false positive rate* as well as *(iv) updating overhead*. To verify the optimal value of the SL, we further compare the performance in varying SLs. All logical algorithms are implemented with C++ in software, and run in a router testbed, which is equipped with 16G RAM, 2 × Intel Xeon(R) 4 Cores 2.27GHz CPU, 8 Forwarding Faces, 2TB Disk Storage. Each set of experiments repeat 100 times to get an average result.

To check the performance in various scale cases, we randomly select name prefixes from our dataset and insert them into the forwarding tables with randomly assigned forwarding faces. The number of inserted entries is ranged from 1 to 20 million. Then, we conduct names queries concurrently to simulate a practical CCN name lookup. We observe and compare the performance when the SL value varies from 1 to 6. We set the number of buckets in mCBFs as 100 times of the number of name prefixes (the load factor is 0.01). Each bucket occupies 8 bits, thus, the maximum counting value is $2^8 - 1$, which is enough to avoid the potential bucket overflow.

(i) Memory cost. In this experiment, we select 1-20 million names from the dataset and insert them into the FIB in order. Meanwhile, we record the memory cost each additional one million names. Fig. 7 shows memory cost in different solutions. It states that memory cost increases with the number of names. The Name Prefix-Trie has the biggest memory cost as well as the sharpest increasing rate. When the number of prefixes rises to 11 million, the memory cost reaches to the max limit of our hardware (nearly 16 GB). This means Name Prefix-Trie has the worst scalability at large scale systems.
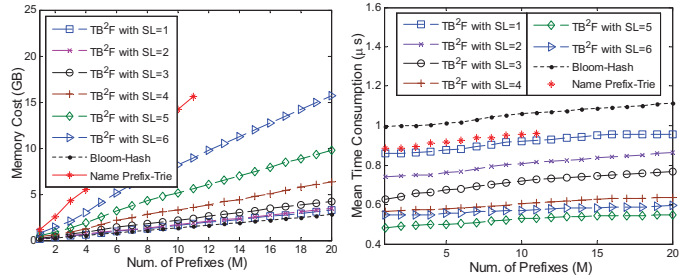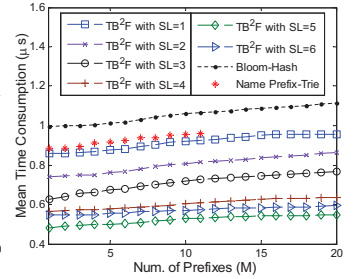

Fig. 7.   Memory cost.

Fig. 8.   Lookup time consumption.

TB^{2}F may provide a great improvement with respect to Name Prefix-Trie. The memory cost decreases as the SL decreases. In the worst case (SL=6), the memory cost is still acceptable by our hardware. In the case of SL=1, the memory cost of TB^{2}F approaches to the one of Bloom-Hash, which exhibits the smallest memory consumption.

(ii) Lookup speed. Fig. 8 presents that the mean lookup time for each lookup increases slightly with the scale of prefixes. Bloom-Hash has the worst performance because Bloom-Hash must conduct complex hash computing for all substrings of each name and cope with a number of conflicts, which consumes additional time. Name Prefix-Trie has a smaller processing time when the number of prefixes is less than 11 million. Unfortunately, as shown in Fig. 8, the usage of Name Prefix-Trie is restricted to a small set of prefixes due to a poor scalability. Concerning TB^{2}F, its processing time first decreases then increases as the SL increases. Note that it achieves to the minimized time consumption when SL=5.

To make a more detailed explanation, Fig. 9 shows the components of processing time for every lookup, including the name reading, T-segment lookup, B-segment lookup, and hash check processes in cases of SL=3,4,5,6. In this figure, the time consumptions in name reading and T-segment lookup are independent on the number of prefixes. Nevertheless, the time in B-segment lookup and hash check slightly increases by the number of prefixes, which contributes to an increase law for the total time. It is also observed that the total time decreases as SL increasing in Fig. 9 (a), (b) and (c), and gets the minimum in case of SL=5. In the case of SL=6, the total time increase again due to the time in TB lookup increases greatly. Further, we make a comparison in a view of different SLs. Here, we focus on two cases of 10M prefixes and 20M prefixes. In this figure, we observe that for increasing values of SL, the time in B-segment lookup drops down, while the time in T-segment grows up. Most excitingly, the time in B-segment lookup first falls below the one in T-segment when SL=5. The main reason is that a bigger SL brings a longer T-segment Tree as well as shorter B-segments, and time consumption highly relies on the length of both kinds of segments. This result confirms the rationality of previous inference, an optimal SL is in the range of $[4, 5]$.

(iii) False positive rate. We also observe the false positive rate with the number of hash functions. We compare the Bloom-Hash and TB^{2}F-based solutions with different SL values. In this experiment, the case of total 20 million names is considered. To reduce the impact from the hash function
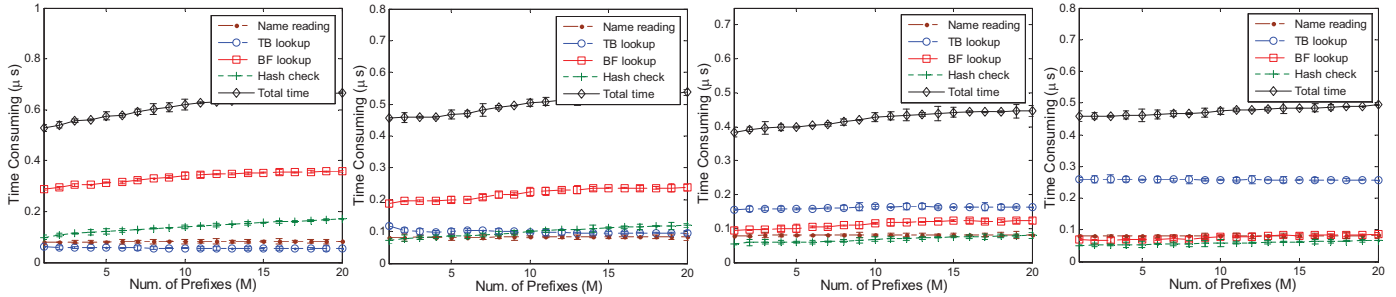
Fig. 9. The components of processing time for each lookup: (a) SL=3; (b) SL=4; (c) SL=5; (d) SL=6 (with 95% Confidence Interval).
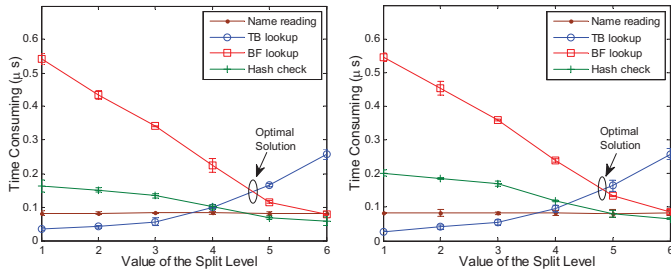

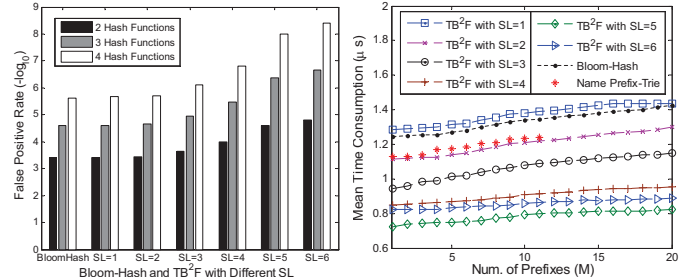Fig. 10. (a) 10M prefixes; (b) 20M prefixes (with 95% CI).

Fig. 11. False positive rate.

Fig. 12. Overhead of updating.

itself, different hash functions [25] are adopted for computing hash keys for CBF. Fig. 11 shows the comparison results when the number of hashes is 2,3,4 (the cases using more hash functions can be easily deduced). In this figure, TB^{2}F-based solutions have a lower false positive rate than Bloom-Hash, since Bloom-Hash has the biggest number of names need to be inserted into the filter. Further, we observe that the false positive rate decreases along with the SL increase. The reason is that the number of B-segments is reduced by the SL increasing, then achieves a smaller false positive rate. This result also validates our previous analysis.

(iv) Overhead of updating. To evaluate the overhead of updating, we first finish inserting 1-20 million prefixes into the FIB in different cases, then randomly delete/insert 20% prefixes from/into the data structure. The average time consuming for per prefix is used to denote the overhead of updating. Fig. 12 shows the overhead varies with the scale of prefixes. In this figure, the overhead keeps a relative high level in Name Prefix-Trie and Bloom-Hash. We observe that TB^{2}F with SL=1 is a little higher than both them, while the ones with other SLs have lower overheads. That's because TB^{2}F-DU bring benefits for the updating process with a limited extra overhead. However, its benefit is dependent on the value of SL. When SL=1, the benefit is too small to make up the extra overhead. As the SL increases, the benefits become more impressive, which results in a relative low update overhead. More importantly, the overhead of updating achieves to the minimum when SL=5, which is the result from the tradeoff of the length of T-segments and B-segments.

In total, above experiments verify that with a suggested SL=5, TB^{2}F achieves a relative good scalability and efficiency by (i) speeding up lookup operations and reducing the false positive rate with respect to Bloom-Hash; (ii) requiring less memory than Name Prefix-Trie; (iii) achieving a lower overhead in updating operations in the large scale case. In

fact, the set of requested names may have a strong regional disparity, and the optimal SL may be different due to additional constraints (*i.e.,* memory). The methodology in this paper can be adopted to guide how to decide an optimal SL locally.

V. RELATED WORK

A well-known solution for fast LPM is the Trie [20], based on which several solutions have been conceived [21]. These solutions are inherently able to handle name aggregations due to their tree structure [22]. The TB [11], a multi-bit expanded Trie, is believed as one of the most efficient schemes among them. Recently, their usage in CCN is being investigated by the research community. Wang *et al.* first propose an effective name component encoding solution for Name Prefix Trie to reduce the memory cost and accelerate name lookup [13]. Additionally, a parallel name lookup is proposed by allocating the logically tree-based structure to parallel physical modules [23]. Li *et al.* also proposed a fast longest-name-prefix lookup framework and implemented it using fat tree and extensible hybrid data structures [24]. Although Trie-based algorithms are simple and efficient, their performance degrades linearly as the tree depth increases. The tree depth can be very huge due to the length of CCN names, so that existing Trie-based solutions if used in CCN cannot easily scale to Internet scenarios.

Another family of alternative approaches rely on the adoption of BFs. The BF [12] provides a space-efficient probabilistic data structure to support set membership queries. Since standard BF does not allow element deletions, Counting BF (CBF) is proposed to tackle this limitation by adding a counter [16]. Dharmapurikar *et al.* first apply BF in the LPM [14]. Recently, there are some emerging BF-based solutions for CCN. You *et al.* propose a distributed PIT table, named DiPIT, which implements a sub-PIT on each CCN node face with a CBF [19]. So *et al.* design a fast forwarding table combining with BFs and data prefetching [25]. Wang *et al.* propose an

efficient lookup scheme for NDN by applying two-stage CBFs [26]. However, BFs bring a chance for false positive rate due to hash collisions, which partially depends on the number of entries inserted into the filters. Due to the potential huge number of CCN names, the false positive rate will be not easily limited at a large scale. What's worse, the hash computing for possible prefixes of CCN names is needed in BFs, which will lower the lookup efficiency, especially for the long names. All these facts impede BFs readily applied in CCN at a large scale.

Besides, hardware-based approaches are also evolved for CCN by taking advantage of its parallelism. Varvello *et al.* target the hardware design of a high-end content router, named by Caesar [27]. Wang *et al.* conduct a study on wire speed name lookup by exploiting GPU's massive parallel processing power [28]. These hardware-based technologies can bring a considerable improvement for the processing, however, they make sacrifice on the high cost and power consumption.

It is this investigation that guides us to focus on novel name lookup design in data structure and consider its suitable hardware implementation. One idea is to split the long CCN name into relative short segments by an alternative split rule. With this simple operation, it allows us to make full use of the advantages of existing lookup mechanisms. Our recent work has originally explored a name lookup mechanism using adaptive prefix Bloom filters [29], which is featured with using a limited number of filters for prefixes and adaptive adjustment for popular prefixes. Different from the previous one, this work focuses on a skillful CCN-customized name lookup engine by leveraging TB and BF with an optimized name split rule based on extensive practical experiments.

VI. CONCLUSION

In this paper, a novel CCN-customized name lookup solution (TB^{2}F) has been presented. TB^{2}F makes a simple partition for hierarchical unbounded CCN names, and explores a scalable data structure as well as efficient lookup scheme by leveraging existing TB and BF. Practical analysis combining with extensive experiments at a large scale suggests an optimal value of the split level, and verifies the performance with fast lookup with affordable memory cost, low false positive rate and low updating overhead, which are believed to be suitable for CCN at large-scale deployment. We believe this fundamental work can provide a novel insight for further developing high scalable and efficient lookup solutions for large-scale CCN usage. In our future work, more optimizations and further enhancements for TB^{2}F will be planned, such as taking into account of caching optimization, names distribution in content retrieval [30][31], evolution for frequent names and GPU implementation to improve the CCN name lookups.

ACKNOWLEDGMENT

REFERENCES

[1] V. Jacobson, D. K. Smetters, *et al.*, "Networking Named Content," *Commun. of the ACM*, vol. 55, no. 1, pp. 117-124, 2012.
[2] B. Ahlgren, C Dannewitz, *et al.*, "A Survey of Information-Centric Networking," *IEEE Communications Magazine*, vol. 50, pp. 26-36, 2012.
[3] G. Piro, L. A. Grieco, G. Boggia and P. Chatzimisios, "Information-Centric Networking and Multimedia Services: Present and Future Challenges," *ETT Trans. on Emerging Telecommun. Technologies*, 2013.
[4] L. Zhang, D. Estrin, V. Jacobson, *et al.*, "Named Data Networking (NDN) project," *Technical Report*, NDN-0001, 2010.
[5] M. F. Bari, S. Chowdhury, *et al.*, "A Survey of Naming and Routing in Information-Centric Networks," *IEEE Communications Magazine*, vol. 50, no. 12, pp. 44-53, 2012.
[6] D. Perino and M. Varvello, "A Reality Check for Content Centric Networking," *Proc. ACM SIGCOMM workshops on ICN*, 2011.
[7] A. Ghodsi, T. Koponen, B. Raghavan, *et al.*, "Information-Centric Networking: Seeing the Forest for the Trees," *Proc. HotNets-X*, 2011.
[8] H. Yuan, T. Song and P. Crowley, "Scalable NDN Forwarding: Concepts, Issues and Principles," *Proc. IEEE ICCCN*, 2012.
[9] http://googleblog.blogspot.com/2008/07/we-knew-web-was-big.html
[10] http://news.netcraft.com/archives/category/web-server-survey
[11] W. Eatherton, Z. Dittia, *et al.*, "Tree Bitmap: Hardware/Software IP Lookups with Incremental Updates," *ACM SIGCOMM Computer Commun. Review*, vol. 34, no. 2, pp. 97-122, 2004.
[12] B. H. Bloom, "Space/Time Trade-offs in Hash Coding with Allowable Errors," *Commun. of the ACM*, vol. 13, no. 7, pp. 422-426, 1970.
[13] Y. Wang, K. He, *et al.*, "Scalable Name Lookup in NDN Using Effective Name Component Encoding," *Proc. IEEE ICDCS*, 2012.
[14] S. Dharmapurikar, P. Krishnamurthy and D. E. Taylor, "Longest Prefix Matching Using Bloom Filters," *IEEE/ACM Trans. Netw.*, vol. 14, no. 3, pp. 397-409, 2006.
[15] W. So, A. Narayanan, D. Oran and Mark Stapp, "Named Data Networking on a Router: Forwarding at 20Gbps and Beyond," *ACM SIGCOMM Demonstration*, 2013.
[16] L. Fan, P. Cao, J. Almeida and A. Z. Broder, "Summary Cache: A Scalable Wide-area Web Cache Sharing Protocol," *IEEE/ACM Trans. Netw.*, vol. 8, no. 3, pp. 281-293, 2000.
[17] M. Waldvogel, G. Varghese, *et al.*, "Scalable High-Speed Prefix Matching," *ACM Trans. on Computer Systems*, vol. 19, no.4, pp. 440-482, 2001.
[18] Y. Mansour, N. Nisan and P. Tiwari, "The computational complexity of universal hashing," *Proc. IEEE Structure in Complexity Theory*, 1990.
[19] W. You, B. Mathieu, *et al.*, "DiPIT: A Distributed Bloom-Filter Based PIT Table for CCN Nodes," *Proc. IEEE ICCCN*, 2012.
[20] E. Fredkin, "Trie Memory," *Commun. of the ACM*, vol. 3, no. 9, pp. 490-499, 1960.
[21] H. Song, F. Hao, *et al.*, "IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards," *Proc. IEEE INFOCOM*, 2009.
[22] M. Bienkowski and S. Schmid, "Competitive FIB Aggregation for Independent Prefixes: Online Ski Rental on Trie," *Proc. SIROCCO*, 2013.
[23] Y. Wang, H. Dai, *et al.*, "Parallel Name Lookup for Named Data Networking," *Proc. IEEE GLOBECOM*, 2011.
[24] F. Li, F. Chen, J. Wu and H. Xie, "Fast Longest Prefix Name Lookup for Content-Centric Network Forwarding," *Proc. ACM ANCS*, 2012.
[25] W. So, A. Narayanan, *et al.*, "Toward Fast NDN Software Forwarding Lookup Engine Based on Hash Tables," *Proc. ACM ANCS*, 2012.
[26] Y. Wang, T. Pan, *et al.*, "NameFilter: Achieving Fast Name Lookup with Low Memory Cost via Applying Two-stage Bloom Filters," *Proc. IEEE INFOCOM mini-conference*, 2013.
[27] M. Varvello, D. Perino and J. Esteban, "Caesar: a Content Router for High Speed Forwarding," *Proc. ACM SIGCOMM wsp on ICN*, 2012.
[28] Y. Wang, Y. Zu, *et al.*, "Wire Speed Name Lookup: A GPU-based Approach," *Proc. ACM NSDI*, 2013.
[29] W. Quan, C. Xu, *et al.*, "Scalable Name Lookup with Adaptive Prefix Bloom Filter for Named Data Networking," *IEEE Communications Letters*, vol. 18, no. 1, pp. 102-105, 2014.
[30] W. Quan, J. Guan, *et al.*, "Content Retrieval Model for Information-Centric MANETs: 2-Dimensional Case," *Proc. IEEE WCNC*, 2013.
[31] W. Quan, J. Guan, *et al.*, "A Content Retrieval Model for Information Centric MANETs: 1-Dimensional Case," *Proc. IEEE GLOBECOM*, 2012.