Name prefix matching using bloom filter pre-searching for content centric network

ARTICLE *in* JOURNAL OF NETWORK AND COMPUTER APPLICATIONS · FEBRUARY 2016

Impact Factor: 2.23 · DOI: 10.1016/j.jnca.2016.02.008

3 AUTHORS, INCLUDING:

Hyesook Lim

Ewha Womans University

43 PUBLICATIONS **367** CITATIONS

SEE PROFILE

Journal of Network and Computer Applications 65 (2016) 36–47

Contents lists available at ScienceDirect

Journal of Network and Computer Applications

journal homepage: www.elsevier.com/locate/jnca

Name prefix matching using bloom filter pre-searching for content centric network

Jungwon Lee, Miran Shim, Hyesook Lim *

Department of Electronics Engineering, Ewha Womans University, Seoul, Korea

ARTICLE INFO

Article history:
Received 15 August 2015
Received in revised form
9 January 2016
Accepted 16 February 2016
Available online 23 February 2016

Keywords:
Content centric network
Name prefix matching
Bloom filter
Name prefix trie

ABSTRACT

As a new networking paradigm for future Internet, content centric networking (CCN) technology provides a contents-oriented communication infrastructure for the rapidly increasing amount of data traffic. For the successful realization of CCN, it is essential to design an efficient forwarding engine that performs high-speed name lookup. This paper proposes the use of a hashing-based name prefix trie and a Bloom filter. In the proposed approach, an off-chip hash table storing the nodes of the name prefix trie is only accessed when the Bloom filter states that the node under querying exists in the trie. In accessing the node depending on the result of the Bloom filter, we propose two algorithms that have different strategies. The first algorithm accesses the trie node for every positive result of the Bloom filter, while the second algorithm first attempts to determine the longest matching length using Bloom filter queries. Trie nodes are accessed from the possible longest length, and tracked back if there is no match. Simulation results show that the proposed approach can provide the output face of each input name, with a single node access on average and with two node accesses in the worst-case using a reasonable size of a Bloom filter.

1. Introduction

Emerging Internet applications such as social network services largely share image, video, and music files. As large and repeatedly requested contents, this traffic is not efficiently transferred in the current Internet, which has a host-based infrastructure. Content centric network (CCN) is a promising next-generation network designed to solve such issues of the current Internet. The CCN is also known as the information centric network (ICN) or named data network (NDN). While the current Internet uses host-to-host communication based on IP addresses, the CCN performs data communication based on content names (Jacobson et al., 2009; Vasilakos et al., 2015; Bari et al., 2012; Xu et al., 2014; Qiao et al., 2015; Wang et al., 2012; Esteve et al., 2008; Perino and Varvello, 2011). In Jacobson et al. (2009), basic CCN features are implemented and the resilience and the performance of CCN architecture are compared with the host-based communication in terms of file transfer, content distribution, and voice calls.

Instead of the concept of source hosts or destination hosts, the CCN uses the concept of content generator and content consumer. Content generators produce contents, and content consumers receive and consume the contents. Routers perform routing using content names instead of IP addresses. Unlike conventional routers, CCN routers have an additional role of caching, which stores contents temporarily, and sends them to consumers requesting the contents. In this way, CCN consumers can rapidly acquire the desired contents, and the same contents are not repeatedly transferred over a network.

CCN uses two types of packets: *Interest* and *Data*. The *Interest* is broadcasted by a consumer. The *Data* is produced by a content generator and transmitted by any node hearing the *Interest* and having the *Data*. A CCN router has three different tables: Contents Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). The CS is a cache storing *Data* packets. The PIT is used to forward *Data* packets, and the FIB is used to forward *Interest* packets. For wire-speed packet forwarding, it is essential to have efficient lookup algorithms that perform the longest name matching for every incoming *Interest* packet.

A trie is an ordered tree-based data structure, the name of which originates from the word re*trie*val. We differentiate the term *trie* from the *tree* as follows. In a trie structure, all the descendants of a node have a common prefix of the string associated with that node, while this is not always true in a tree structure. A name prefix trie (NPT) has been proposed as an extended trie for a name lookup (Wang et al.,

* Corresponding author. Tel.: +82 2 3277 3403.
E-mail address: hlim@ewha.ac.kr (H. Lim).

http://dx.doi.org/10.1016/j.jnca.2016.02.008
1084-8045/

2011). The NPT provides an intuitive way for the name lookup with the longest name matching, but has an issue in search performance, since an FIB table can have many millions of content names and each content name can be excessively long.

The motivation of this paper is to propose a new approach for the longest name matching used in FIB lookups in CCN routers. The proposed approach is based on the name prefix trie. In order to solve the search performance issue of the NPT, we propose to add an on-chip Bloom filter which is queried before the access to the NPT, which is stored in an off-chip hash table. As a space-efficient probabilistic data structure used to test whether an element is a member of a set, Bloom filters have been popularly applied to network algorithms (Song et al., 2005; Tong et al., 2014; Mun and Lim, 2015; Lim et al., 2014a, 2014b). Since an access to an off-chip memory takes 10–20 times longer than an access to an on-chip memory (Panda et al., 2000), by pre-searching the on-chip Bloom filter, the off-chip hash table storing trie nodes is only accessed when there is a high possibility of a matching entry in our proposed approach. The earlier and shorter version of this paper was presented in (Lim et al., 2015).

The performance of our proposed algorithms is evaluated through simulation. Since the format of CCN names is not yet defined, URL names that have hierarchical characteristics similar to CCN names are used for our simulation (Wang et al., 2011). Memory requirements for creating a Bloom filter and storing the NPT are also evaluated. Using inputs with 3 times the number of stored URLs, the search performance is also evaluated.

This paper is organized as follows. Section 2 describes related works such as the name prefix trie, previous name lookup algorithms, and the Bloom filter theory. Section 3 introduces the building and searching procedures of the proposed algorithms. Section 4 shows the performance evaluation results, and Section 5 concludes the paper.

2. Related works

2.1. Name prefix trie

Each component in a URL is composed of variable-length character strings, and is differentiated by a dot or a slash. For example, the URL of http://www.youtube.com/user/PewDiePie is composed of 4 components: *youtube*, *com*, *user*, and *PewDiePie*. In storing the URL into a name

Table 1
An example of an FIB at CCN routers.

Content name	Output face
http://t-online.de	1
http://youtube.com	2
http://google.com	3
http://facebook.com	4
http://visitKorea.or.kr	5
http://warning.or.kr	6
http://google.com.hk	7
http://empowernetwork.com/25e8w	8
http://youtube.com/user/SkydoesMincraft	9
http://youtube.com/user/PewDiePie	10

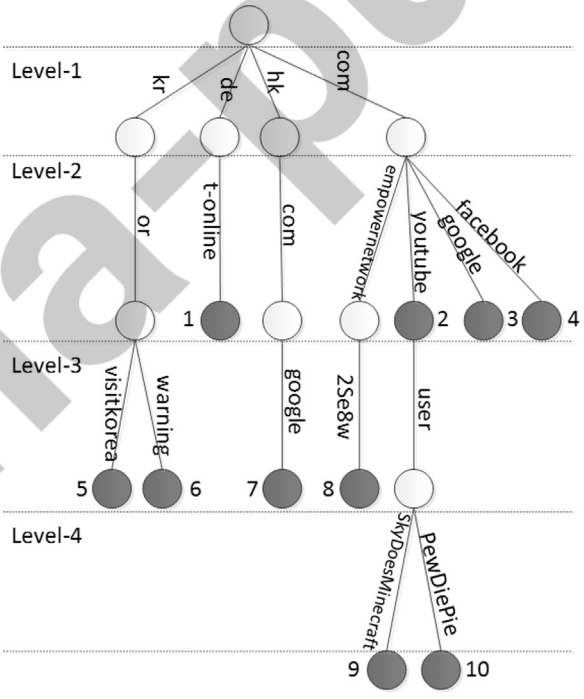


Fig. 1. Name prefix trie.

prefix trie, the example name is converted into *com/youtube/user/PewDiePie* by reversely ordering the components with dots and by differentiating components using a slash.

Table 1 shows an example of an FIB with 10 arbitrary content names and corresponding output faces. Fig. 1 shows the name prefix trie (NPT) constructed for the example.

Each black node represents a content name and stores an output face, to which inputs matching the content name have to be forwarded, while white nodes are created in the path to a black node from the root node. The NPT accommodates variable-length components and variable-length names as shown. The depth of an NPT is determined by the content name that has the most number of components.

Similar to the search in a binary trie for an IP address lookup, the search in an NPT starts from the root node, and sequentially follows the child pointer that has the matching component with a given input. When there is no matching component to follow, the search procedure is completed and returns the best matching name, which is the most recent matching name. For example, assume that an input youtube.com/user/musicbox is given. The input name that needs to be searched from the NPT is *com/youtube/user/musicbox*. At the node of com/youtube/user, since there is no edge to follow, the search is over and returns the output face 2, which is the output face of *com/youtube*. Assuming that the NPT is stored in an off-chip memory, the number of off-chip node accesses of this example is 4, counting the root node, *com*, *com/youtube*, and *com/youtube/user*.

The name lookup using the name prefix trie has an issue of search performance. Since the search is sequentially performed starting from the root node, and since the name lengths can be excessively long, the search performance in an NPT is not sufficient for a wire-speed name lookup. In Wang et al. (2013a), a high-speed name lookup engine implemented in GPU (Graphic Processor Unit) platform using name component encoding was introduced.

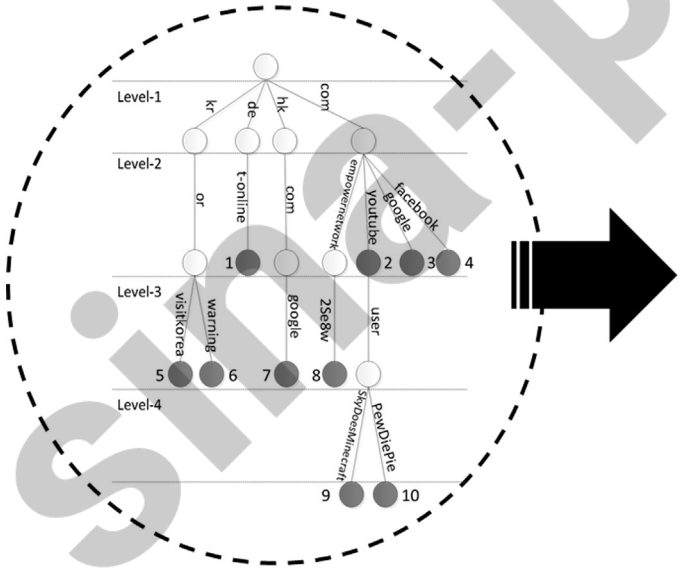
2.2. Previous name lookup algorithms

A parallel search algorithm (Wang et al., 2011) divides an NPT into a multiple number of modules, considering the accessing probability of a module, and performs a parallel search for the modules. However, it is difficult to estimate the accessing probability for dynamically changed input streams, and the parallel search involves high hardware complexity.

To improve the search performance of an NPT, in So et al. (2013), it is proposed to start the search, either from the level with the most number of components or the level with an accumulated number of components equal to 75% of the total number. Depending on the result of the first level access, the search can proceed to a shorter level or a longer level. This algorithm has a problem of prefix-seeking, in which the search should be continued until a matching entry is found or a root node is visited, when there is no matching entry.

The name filter algorithm (Wang et al., 2013) uses 2-stage FIB lookups, in which both stages are composed of Bloom filters. Because of the false positive problem of a Bloom filter, which produces inaccurate results, this algorithm proposes the control of the false positive rate less than 10^{-8} by increasing the sizes of Bloom filters.

The adaptive Bloom filter algorithm (Quan et al., 2014) uses a Bloom filter, a hash table, and a name prefix trie. In this algorithm, each content name is separated by a B-prefix part and a T-Suffix part. The length of the B-prefix part is fixed and the remaining components are included in the T-suffix. The B-prefix part is implemented using a Bloom filter and a hash table, and the T-suffix part is implemented using an NPT. When there is a matching entry in the B-prefix part, the NPT search is performed at the T-suffix part. The performance of the algorithm depends on the selection of the B-prefix length for each given set of content names.



Hash table

Trie node	Output face
hkcomgoogle	7
hkcom	-
krorwarning	6
comgoogle	3
comyoutubeuser	-
com	-
det-online	1
hk	-
comyoutube	2
comempowernetwork	-
kror	-
comyoutubeuserSkydoesMincraft	9
comfacebook	4
kr	-
krorvisitKorea	5
comyoutubeuserPewDiePie	10
comempowernetwork25e8w	8
de	-

Fig. 2. Hashing-based NPT.

2.3. Bloom filter theory

A Bloom filter (Bloom, 1970) is a bit-vector representing the membership information of a set of elements. A Bloom filter that represents a set $S=\{x_1, x_2, \ldots, x_n\}$ of n elements is described by an array of m bits, initially all set to 0. The Bloom filter supports two operations: programming and querying. In programming, for an element x_i (for $i=1, \ldots, n$) in the set S, k different hash indices are computed, where $0 \le h_j(x_i) < m$ for $j=1, \ldots, k$. All the bit-locations corresponding to k hash indices are then set as 1 in the Bloom filter. In other words, each bit of a Bloom filter corresponds to a hash index, and instead of storing the data itself in the hashing index, the corresponding bit of a Bloom filter is set to 1 to represent the existence of the data.

Querying is performed to test whether an input y is a member of the set. For an input y, k hash indices are generated using the same hash functions that were used to program the filter. The bit-locations in the Bloom filter corresponding to the hash indices are checked. If any one of the locations is 0, then y is definitely not a member of set S, and it is termed *negative*. If all the locations corresponding to hash indices were 1, then the input may be a member of the set, and it is termed *positive*.

However, it is possible that these locations would have been set by some other elements in the set. This type of positive result is termed a *false positive*. For an element y that is not in S ($y \notin S$) under querying, the false positive probability f can be calculated as (Lim et al., 2014a)

$$f=(1-p)^k=\left(1-\left(1-\frac{1}{m}\right)^{kn}\right)^k. \tag{1}$$

For a given ratio of m/n, it is known that the false positive probability is minimized when the number of hash functions k has the following relationship (Dharmapurikar et al., 2006):

$$k=\frac{m}{2^{\lceil \log_2 n \rceil}} \ln 2 \tag{2}$$

On the whole, a Bloom filter may produce false positives but not false negatives. The false positive rate of a Bloom filter can be controlled sufficiently small by increasing the Bloom filter size, but cannot be completely removed.

3. Proposed algorithms

3.1. Hashing-based NPT (Hashing-NPT)

The aim of this paper is to improve the search performance of a name prefix trie(NPT) using Bloom filters. As the first step, we need to apply hashing in constructing the NPT, and we describe a hashing-based NPT algorithm in this section. Fig. 2 shows the hash table constructed for the name prefix trie shown in Fig. 1. Every node in a name prefix trie is stored into a hash table. Note that in storing a node at the *level-i* of the NPT, the *i* components from the root node to the *level-i* are concatenated, and the concatenated string is used as a hash key to obtain a hash index. The concatenated string that has a variable length should be converted into a fixed-length string, since each entry of the hash table has a fixed width. The converted fixed-length string is stored in the hash entry corresponding to the hash index. (The hash entry shown in Fig. 2 has a concatenated string instead of converted fixed-length string for simplicity.) In storing the L^{th} component, where L is the length of a content, the output face corresponding to the content name is also stored.

The procedure to store a content name in the hash table for the hashing-based NPT algorithm is shown in Algorithm 1. This procedure is repeated for all the content names stored in an FIB.

Algorithm 1. Storing a content name in the hash table in hashing-based NPT algorithm.

```
Function Store_Name(contentName, outFace)
    (c_1, c_2, ..., c_L) = Decompose(contentName);
    // L is the length of each contentName
    for (i = 1 to L) do
        conStr = Concatenate(c_1, c_2, ..., c_i);
        HT_ind = HashFunc(conStr);
        StoredStr = Convert(conStr);
        if (i == L) then
            StoreHT (HT_ind, StoredStr, outFace);
        else
            StoreHT (HT_ind, StoredStr, NULL);
        end
    end
end
```

Algorithm 2. Searching a content name in hashing-based NPT.

```
Function Search_NPT(inName)
    BMN = default;
    (c_1, c_2, ..., c_I) = Decompose(inName);
    // I is the length of the inName
    for (i = 1 to I) do
        conStr = Concatenate(c_1, c_2, ..., c_i);
        HT_ind = HashFunc(conStr);
        InStr = Convert(conStr);
        match = CompString(StoredStr[HT_ind], InStr);
        if (match != 1) then
            break; // not found
        else // a matching entry found

            if (outFace[HT_ind] != NULL) then
                BMN = outFace[HT_ind];
            end
        end
    end
    return BMN;
end
```

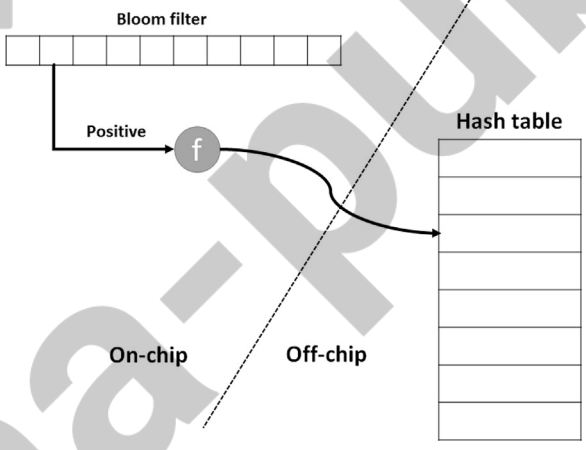


Fig. 3. Proposed name prefix trie with a BF (NPT-BF).

The search procedure used to determine the best matching name (BMN) in the hashing-based NPT algorithm is shown in Algorithm 2. For the content name in each *Interest* packet, the concatenated string of i components (for $1 \le i \le I$, where I is the length of the input name) is used to obtain a hash index. The concatenated string is also converted into a fixed length string, for comparison with the string stored in the hash entry corresponding to the hash index. If there is a matching entry, the search procedure continues, using the concatenated string of the first through the $(i+1)^{th}$ components. This procedure is repeated, until there is no matching entry, or until the search reaches the level equal to the length of the input content name. Since the output face of the longest matching entry should be returned, the output face of the current matching entry should be remembered in the search procedure. Note that in a trie structure, if there is no entry matching the current string of concatenated components, there is also no matching entry in longer levels. Hence, the search is immediately over.

For example, assume the same input as in the NPT search *com/youtube/user/musicbox*. The search procedure finds out matching hash entries with *com*, *comyoutube*, and *comyoutubeuser*, but no matching entry with *comyoutubeusermusicbox*. Hence, the search is over and returns the output face 2, which is the output face of *comyoutube*. The number of hash table accesses of this example is 4.

This paper proposes to improve the search performance of the hashing-based NPT algorithm using a Bloom filter. We propose two algorithms: NPT-BF and NPT-BF-Chaining. Both algorithms have the same structure, consisting of an on-chip Bloom filter and an off-chip hash table. However, they have different search strategies.

3.2. Name prefix trie with a bloom filter (NPT-BF)

The NPT-BF is shown in Fig. 3. A Bloom filter is queried before the hash table access; this Bloom filter reduces unnecessary hash table accesses by producing negative results when there is no matching entry.

Algorithm 3. Searching a content name in NPT-BF.

```
Function Search_NPT_BF(inName)
    BMN = default;
    (c_1, c_2, ..., c_I) = Decompose(inName);
    // I is the length of the inName
    for (i = 1 to I) do
        conStr = Concatenate(c_1, c_2, ..., c_i);
        BF_ind = MakeBFInd(conStr);
        if (BFQuery(BF_ind) == positive) then
            HT_ind = HashFunc(conStr);
            InStr = Convert(conStr);
            match = CompString(StoredStr[HT_ind], InStr);
            if (match != 1) then
                break; // false positive
            else
                // a matching entry found
                if (outFace[HT_ind] != NULL ) then
                    BMN = outFace[HT_ind];
                end
            end
        else
            // Search is over when BF negative
            break;
        end
    end
end
```

The search procedure of the proposed NPT-BF algorithm is shown in Algorithm 3. The Bloom filter is queried first using the concatenated string of i components (for $1 \le i \le I$, where I is the length of the input name). If the Bloom filter produces a positive result, the hash table is accessed. If a matching entry is found in the hash table, the Bloom filter query is continued using $(i+1)$ components. However, if no matching entry is found, which means that the Bloom filter positive result is false, the search is finished and returns the currently remembered output face. Otherwise, if the Bloom filter produces a negative result, the search is also finished, and returns the currently remembered output face.

For example, assume the same input youtube.com/user/musicbox. The Bloom filter produces positive results with *com*, *comyoutube*, and *comyoutubeuser*, and hence the hash table is accessed for these 3 nodes. However, the Bloom filter produces a negative result with *comyoutubeusermusicbox*. Hence, the search is over without accessing the hash table and returns the output face 2, which is the output face of *comyoutube*. The number of hash table accesses of this example is 3, which is one smaller than the hashing-NPT, since the hash table is not accessed for the negative result of the Bloom filter.

In summary, the difference between NPT-BF and the hashing-NPT is that the search can be immediately finished when the Bloom filter produces a negative result in NPT-BF, without accessing the hash table. Hence, NPT-BF can improve the search performance of the hashing-NPT, which is measured by the number of off-chip hash table accesses.

3.3. Name prefix trie with a bloom filter chaining (NPT-BF-Chaining)

The NPT-BF-Chaining algorithm utilizes the fact that the false positive probability of a Bloom filter can be controlled to be sufficiently low by increasing the Bloom filter size. In this algorithm, if a Bloom filter produces a positive result, the Bloom filter querying continues to the next level without accessing the hash table. If the Bloom filter produces a negative result, since it is always true, the search procedure moves one level back, which is the level of the last positive. The off-chip hash table is accessed at this level. If there is a matching entry with an output face, the search is completed and returns the output face. If there is no matching entry, it can be one of the two cases: the Bloom filter positive result is false, or the Bloom filter positive is true but the node is an internal node without an output face. In these cases, backtracking should occur, until a matching entry with an output face is found in the hash table. The overall architecture of the NPT-BF-Chaining is shown in Fig. 4.

Algorithm 4. Searching in NPT-BF-Chaining.

```
Function Search_NPT_BF_Chaining(inName)
    BMN = default;
    (c_1, c_2, ..., c_I) = Decompose(inName);
    // I is the length of the inName
    for (i = 1 to I) do
        conStr = Concatenate(c_1, c_2, ..., c_i);
        BF_ind = MakeBFInd(conStr);
        if (BFQuery(BF_ind)== negative) then
            break;
        end
    end
    if (i < I) then
        i = i − 1;
    end
    while (i > 0) do
        conStr = Concatenate(c_1, c_2, ..., c_i);
        HT_ind = HashFunc(conStr);
        InStr = Convert(conStr);
        match = CompString(StoredStr[HT_ind], InStr);
        if ( (match == 1) && (outFace[HT_ind]! = NULL) ) then
            BMN = outFace[HT_ind];
            break;
        else
            i = i − 1; // back-tracking
        end
    end
end
```

The search procedure of NPT-BF-Chaining is summarized in Algorithm 4. While the building procedure of the NPT-BF-Chaining is the same as that of NPT-BF, the search procedure differs, whereby the Bloom filter queries are sequentially performed until a negative result occurs. The search procedure of the NPT-BF-Chaining algorithm has two steps. The first step involves querying the Bloom filter sequentially for positive results. The first step continues until a Bloom filter negative result occurs or all the components of the input name are used.

The off-chip hash table is accessed in the second step. The negative result of a Bloom filter at level i means that there is no string in the hash table matching to the concatenated string of i components. Since there was a positive result in level $(i-1)$, the hash table is accessed

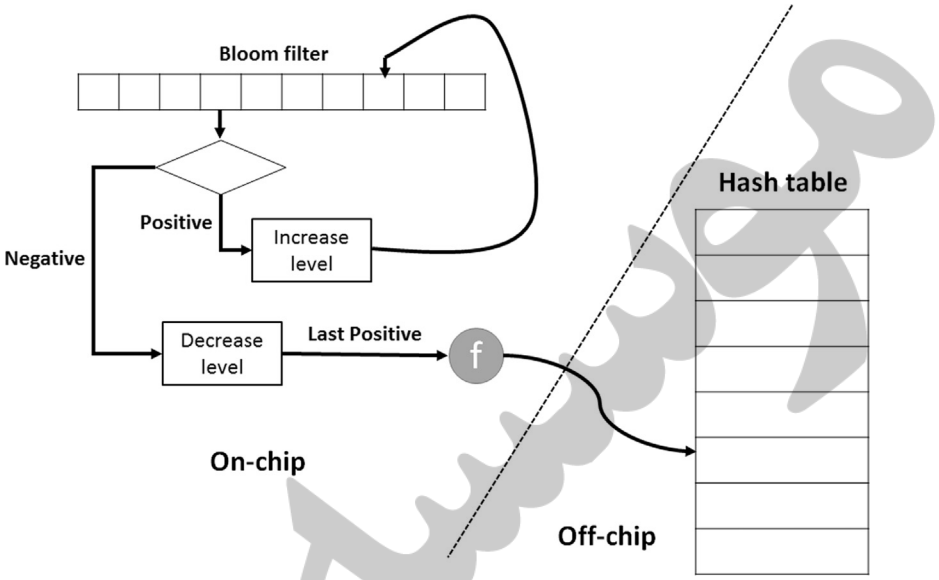


Fig. 4. Proposed name prefix trie with a BF chaining (NPT-BF-Chaining).

Table 2
Number of content names at each level.

Level	10 k	50 k	100 k	300 k
1	0	0	0	0
2	501	3943	9557	55,473
3	1429	8325	18,471	65,379
4	2059	11,284	23,579	62,189
5	2392	11,860	22,562	54,417
6	1693	7274	13,501	32,642
7	975	3964	6962	17,080
8	533	1868	3000	7151
9	216	783	1267	2970
10	100	338	554	1351
Longer	102	361	545	1337
Total	10,000	50,000	100,000	300,000

Table 3
Data structures of hash entry.

Field	Length (bits)
Stored string	64
Output face	8

for the concatenated string of $(i-1)$ components. (In case where the first step is finished because all the components of the input are used, the hash table should be accessed from level i.)

Three cases can be encountered when accessing the hash table: a matching node with an output face, no node, or an internal node without an output face. If a matching entry with an output face is encountered in the hash table, the search procedure is completed and returns the output face of the corresponding entry. Otherwise, if there is no node, then the positive of the Bloom filter is a false positive, which causes backtracking. Otherwise, if an internal node without an output face is encountered, then the positive result of the Bloom filter is true, but back-tracking should also occur. This problem is solved by pre-computing the output face of the best matching name into every internal node. The best matching name is the name in the direct ancestor of each internal node. In the performance evaluation section, we will show the simulation results for both cases: with and without the pre-computation.

For example, assume the same input youtube.com/user/musicbox. The Bloom filter produces positive results with *com*, *comyoutube*, and *comyoutubeuser*, but a negative result with *comyoutubeusermusicbox*. When the Bloom filter produces the negative result, the hash table is accessed for the last positive result, which is *comyoutubeuser*. Since it is an empty node without an output face, a back-tracking occurs. The hash entry of *comyoutube* is accessed, and the search is over by returning the output face 2. The number of hash table accesses of this example is 2. If the output face of *comyoutube* were pre-computed at the node of *comyoutubeuser*, the number of hash table accesses could be 1.

In summary, the NPT-BF-Chaining algorithm postpones the access to the hash table until the Bloom filter produces a negative result, while the NPT-BF algorithm accesses the off-chip hash table for every positive result of the Bloom filter. In other words, the NPT-BF-Chaining algorithm firstly determines the possible longest length matching an input string by using Bloom filter queries. If the false positive rate is controlled to be sufficiently low by properly sizing the Bloom filter, it can provide high-speed search performance. It is shown through simulation that the NPT-BF-Chaining algorithm provides a name lookup using a single off-chip hash table access on average.

4. Performance evaluation

Performance evaluation has been carried out using URL names, since URL names have a hierarchical structure similar to content names. Among the 1 million names provided in ALEXA (Alexa,), four sets with sizes of 10,000, 50,000, 100,000, and 300,000 names are extracted for our simulation. For these sets, NPT, hashing-NPT, NPT-BF, and NPT-BF-Chaining algorithms are constructed using C++ language.

Table 2 shows the number of content names according to the number of components in each name. The number of components in each name determines the level at which the content name is located in a name prefix trie. Each set is named using the number of elements in the set, such as 10 k, 50 k, 100 k, and 300 k.

Table 3 shows the data structure of the off-chip hash table storing nodes of the name prefix trie. We assume that a perfect hash function is given in storing and accessing trie nodes in the hash table. Each string stored into a NPT node is converted into a 64-bit string by the hash function, and hence each hash entry has a 9-byte width as shown in Table 3.

Table 4 shows the characteristics of name prefix tries constructed for each set. The number of entries in the hash table, N' is fixed as $2^{\lceil \log_2 N \rceil}$, where N is the number of nodes in a name prefix trie. Using the data structure shown in Table 3, the off-chip memory

Table 4
Trie characteristics.

Characteristic	10 k	50 k	100 k	300 k
No. of names	10,000	50,000	100,000	300,000
Trie depth	21	21	30	32
No. of nodes (N)	39,388	178,030	336,314	794,866
No. of entries (N')	2^{16}	2^{18}	2^{19}	2^{20}
Hash table size (KB)	576	2304	4608	9216

Table 5
Search performance: 10 k.

Metric	NPT	Hashing-NPT	NPT-BF			NPT-BF-Chaining (w/o pre-comp)			NPT-BF-Chaining (w pre-comp)		
BF size	–	–	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$
BF mem (KB)	–	–	32	64	128	32	64	128	32	64	128
No. of false pos	–	–	308	19	0	318	19	0	318	19	0
Wst false pos	–	–	–	–	–	2	1	0	2	1	0
Avg node acc	4.84	4.05	3.85	3.84	3.84	1.68	1.67	1.67	0.98	0.97	0.97

Table 6
Search performance: 50 k.

Metric	NPT	Hashing-NPT	NPT-BF			NPT-BF-Chaining (w/o pre-comp)			NPT-BF-Chaining (w pre-comp)		
BF size	–	–	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$
BF mem (KB)	–	–	128	256	512	128	256	512	128	256	512
No. of false pos	–	–	2242	153	4	2406	153	4	2406	153	4
Wst false pos	–	–	–	–	–	3	1	1	3	1	1
Avg node acc	4.56	3.80	3.58	3.56	3.56	1.67	1.65	1.65	0.99	0.97	0.97

Table 7
Search performance: 100 k.

Metric	NPT	Hashing-NPT	NPT-BF			NPT-BF-Chaining (w/o pre-comp)			NPT-BF-Chaining w pre-comp)		
BF size	–	–	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$
BF mem (KB)	–	–	256	512	1024	256	512	1024	256	512	1024
No. of false pos	–	–	3667	213	1	3873	213	1	3873	213	1
Wst false pos	–	–	–	–	–	3	1	1	3	1	1
Avg node acc	4.58	3.80	3.59	3.58	3.58	1.69	1.68	1.67	0.99	0.97	0.97

Table 8
Search performance: 300 k.

Metric	NPT	Hash-NPT	NPT-BF			NPT-BF-Chaining (w/o pre-comp)			NPT-BF-Chaining (w pre-comp)		
BF size	–	–	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$	$4N'$	$8N'$	$16N'$
BF mem (KB)	–	–	512	1024	2048	512	1024	2048	512	1024	2048
No. of false pos	–	–	18,392	1590	33	19,832	1599	33	19,832	1599	33
Wst false pos	–	–	–	–	–	4	2	1	4	1	1
Avg node acc	4.37	3.62	3.39	3.37	3.37	1.67	1.65	1.65	1.00	0.97	0.97

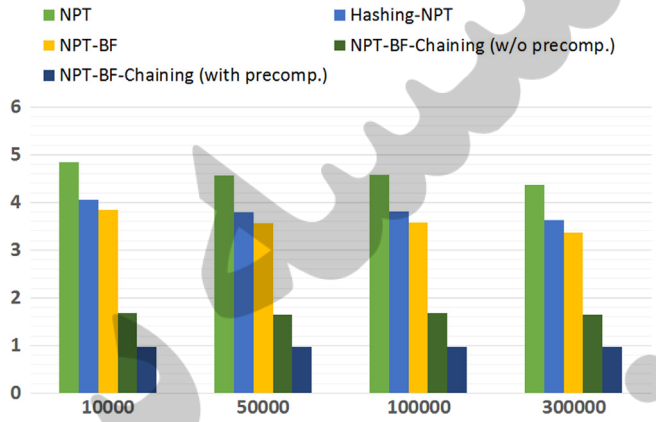


Fig. 5. Average number of node accesses.

requirement is estimated. The memory requirement is calculated by the number of entries N' multiplied by the entry width, which is 9 bytes. The memory requirement shown in Table 4 is denoted as kilobytes by dividing by 1024. Note that the memory requirements for the off-chip hash tables for hashing-NPT, NPT-BF, and NPT-BF-Chaining are the same, since they construct the same trie. Note that the size of the hash table does not fit in an on-chip memory for large name sets in current technology, even though we allocate the minimum size by assuming that a perfect hash function is given for storing each NPT node into the hash table.

The number of inputs for evaluating the search performance of each algorithm is 3 times the number of elements in the sets. The 1/3 of the input trace includes the content names used to build the name prefix trie, and the remaining 2/3 includes the content names not used to build the name prefix trie.

For the number of nodes N in a name prefix trie that should be programmed into a Bloom filter, the basic size of a Bloom filter can be $N' = 2^{\lceil \log_2 N \rceil}$. Let m be the size of a Bloom filter, which is a multiple of the basic size. Then the number of hash indices $k = (m/N') \ln 2$ (Song et al., 2005). In this paper, we performed simulations for the different sizes of the Bloom filters, such that $m = 4N'$, $8N'$, and $16N'$, and evaluated the search performance related to the size of a Bloom filter.

A cyclic redundancy check (CRC) generator is an excellent bit scrambler, popularly used in generating hash indices. A simple 64-bit CRC generator has been used to obtain hash indices for the Bloom filter. The registers in the CRC generator are initialized as zero. Each input bit is shifted into the CRC generator, and after all bits of an input are shifted to the CRC generator, the register value is returned as a CRC code. A multiple number of hash indices with variable sizes can be easily extracted from the CRC code by combining bits in a CRC code (Alagu Priya and Lim, 2010).

Tables 5–8 show the simulation results. As stated in previous section, in the NPT-BF-Chaining algorithm, a back-tracking can occur even though the Bloom filter positive is true, in case if the accessed hash entry represents an internal node which does not have an output face. In this reason, for the NPT-BF-Chaining algorithm, we performed the simulation for two cases: *without the pre-computation* and *with the pre-computation* of the output face of the best matching name. In the NPT-BF-Chaining without pre-computation, whenever an internal node is encountered, the number of node accesses is increased, until a node with a content name is reached. On the other hand, in the NPT-BF-Chaining with pre-computation, the content name of the direct ancestor is pre-computed and stored into internal nodes, and hence the number of node accesses is not increased, when an internal node is encountered. Hence, the difference in the number of node accesses between *without pre-computation* and *with pre-computation* shows the number of internal node accesses.

Table 5 shows the search performance for the 10 k set. The NPT and the hashing-NPT algorithms determine the output face by 4.84 and 4.05 node accesses on average, respectively. Since these algorithms are based on trie structure, if a node does not exist in a level, search procedure is finished immediately. Some inputs do not have matching names at the first level, and hence the number of node accesses for these inputs is 1 for both algorithms. The hashing-NPT shows slightly better performance on average node accesses than the NPT, since the NPT requires one more access than the hashing-NPT, when all components of an input name are used up in the search procedure. For example, assume the input youtube.com/user. The number of node accesses for the NPT is 4, counting the root node, *com*, *com/youtube*, and *com/youtube/user*, while it is 3 for the hashing-NPT, counting *com*, *comyoutube*, and *comyoutubeuser*.

For algorithms with a Bloom filter, the memory requirement for Bloom filters is calculated related to the base size N'. The *wst false pos* is the worst-case number of false positives occurring, and converges to zero for the Bloom filter size of $16N'$.

While a trie node stored in an off-chip hash table is accessed for every positive result of the Bloom filter in the NPT-BF algorithm, the NPT-BF-Chaining algorithm continuously performs the Bloom filter queries without accessing the node, until the Bloom filter produces a negative result (or all the components of the input name are used).

For the Bloom filter size of $16N'$, while the average number of node accesses for the NPT-BF algorithm is 3.84, this improves to 1.67 for NPT-BF-Chaining without pre-computation, and improves further to 0.97 in NPT-BF-Chaining with pre-computation. The performance difference in the NPT-BF-Chaining algorithm without and with pre-computation results from avoiding the back-tracking at empty internal nodes. Note that inputs having a negative result in the first level do not have matching names, and hence the hash table is not accessed at all for these inputs in Bloom filter-based algorithms.

Tables 6, 7 and 8 show the search performance for 50 k, 100 k, and 300 k, respectively. The average number of node accesses in the NPT-BF-Chaining with pre-computation is 1 or less in these sets. The worst-case number of false positives is 1 for Bloom filter sizes of $16N'$. Hence, the worst-case number of node accesses is 2 in these sets. Note that the worst-case number of node accesses for the NPT and the hashing-NPT is equal to the trie depth, and it is a large number as shown in Table 4. For the NPT-BF algorithm, the worst-case number of node accesses is one smaller than the trie depth, since the Bloom filter avoids one hash table access by producing a negative result, after the search procedure reaches the bottom of the trie.

Fig. 5 compares the average number of node accesses in each algorithm for the Bloom filter size of $16N'$.

It shows that the search performance of each algorithm is not much related to the number of content names stored in the name prefix trie. NPT-BF-Chaining with pre-computation provides the best performance, and it can determine the output face through a single node access on average and two node accesses in the worst-case.

5. Conclusion

This paper proposed new algorithms to improve the search performance of FIB lookups for packet forwarding in a content-centric network. Our proposed approach is based on the fact that accesses to an off-chip memory takes much longer than accesses to an on-chip memory. We proposed to use an on-chip Bloom filter to reduce accesses to an off-chip hash table. Hence the amount of memory and a number of queries for the Bloom filter are traded off the reduced number of hash table accesses in our proposed approach.

We first described the name prefix trie used for the content name search that can be implemented using hashing-based architecture: hashing-NPT. In order to improve the search performance of the hashing-NPT algorithm, we proposed two algorithms using Bloom filters: NPT-BF and NPT-BF-Chaining. Both algorithms have the same construction procedure, though with different search strategies. The NPT-BF algorithm accesses the off-chip hash table for each positive result of a Bloom filter. The NPT-BF-Chaining algorithm utilizes the fact that the false positive rate of a Bloom filter can be sufficiently small, as the size of a Bloom filter increases. Hence, the NPT-BF-Chaining algorithm continues the query to the next level for the case of a positive result at the current level, until the Bloom filter produces a negative result. Since Bloom filter negative results are always true, the NPT-BF-Chaining algorithm accesses the node in the previous level, which is the level of the most recent positive. In this way, the NPT-BF-Chaining algorithm can determine the longest level that has the matching node. The search performance of the NPT-BF-Chaining algorithm is further improved by pre-computing the output face of the best matching name for internal nodes.

Simulation results show that the proposed NPT-BF-Chaining with pre-computation can provide the output face of each input name less than a single node access on average and with two node accesses in the worst-case, using a reasonable size of a Bloom filter.

Acknowledgments

This research was supported by the National Research Foundation of Korea (NRF), NRF-2014R1A2A1A11051762 and NRF-2015R1A2A1A15054081. This research was also supported by the Ministry of Science, ICT and Future Planning (MSIP), Korea, under the Information Technology Research Center (ITRC) support program (IITP-2015-H8501-15-1007) supervised by the Institute for Information & communications Technology Promotion (IITP).

References

Alagu Priya AG, Lim H. Hierarchical packet classification using a bloom filter and rule-priority tries. Comput. Commun. 2010;33(June (10)):1215–26.
Alexa the Web Information Company. ⟨http://www.alexa.com/⟩.
Bari MdF, Chowdhury SR, Ahmed R, Mathieu B. A survey of naming and routing in information-centric networks. IEEE Commun. Mag. 2012(December):44–53.
Bloom B. Space/time tradeoffs in hash coding with allowable errors. Commun. ACM 1970;13(7):422–6.
Dharmapurikar S, Krishnamurthy P, Taylor DE. Longest prefix matching using Bloom filters. IEEE/ACM Trans. Netw. 2006;14:397–409.
Esteve C, Verdi FL, Magalhaes MF, 2008. Towards a new generation of information-oriented Internet working architectures. In: ACM CoNEXT.
Jacobson V, Smetters DK, Thornton JD, Plass M, Briggs N, Braynard R, 2009. Networking named content. In: ACM CoNEXT.
Lim H, Lee N, Lee J, Yim C. Reducing false positives of a bloom filter using cross-checking bloom filters. Appl. Math. Inf. Sci. 2014a;8(July (4)):1865–77.
Lim H, Lim K, Lee N, Park K. On adding bloom filters to longest prefix matching algorithms. IEEE Trans. Comput. 2014b;63(February (2)):411–23.
Lim H, Shim M, Lee J, 2015. Name prefix matching using bloom filter pre-searching. In: IEEE/ACM ANCS, May, pp. 203–204.
Mun J, Lim H. New approach for efficient ip address lookup using a bloom filter in trie-based algorithms. IEEE Trans. Comput. 2015 Early Access.
Panda P, Dutt N, Nicolau A. On-chip vs. off-chip memory: the data partitioning problem in embedded processor-based systems. IEEE Trans. Design Autom. Electron. Syst. 2000;5(July (3)):682–704.
Perino D, Varvello M, 2011. A reality check for content centric networking. In: Proceedings of ACM, August, pp. 44–49.
Qiao X, Nan G, Peng Y, Guo L, Chen J, Sun Y, Chen K. NDNBrowser: an extended web browser for named data networking. J. Netw. Comput. Appl. 2015;50(April):134–47.
Quan W, Xu C, Guan J, Zhang H, Grieco LA. Scalable name lookup with adaptive prefix bloom filter for named data networking. IEEE Commun. Lett. 2014;18(January (1)):102–5.
Song H, Dharmapurikar S, Turner J, Lockwood J, 2005. Fast hash table lookup using extended bloom filter: an aid to network processing. In: ACM SIGCOMM, pp. 181–192.
Tong E, Niu W, Li G, Tang D, Chang L, Shi Z, Ci S. Bloom filter-based workflow management to enable QoS guarantee in wireless sensor networks. J. Netw. Comput. Appl. 2014;39(March):38–51.
Vasilakos AV, Li Z, Simon G, You W. Information centric network: research challenges and opportunities. J. Netw. Comput. Appl. 2015;52(June):1–10.
Wang Y, Dai H, Jiang J, He K, Meng W, Liu B, 2011. Parallel name lookup for named data networking. In: IEEE GLOBECOM, December, pp. 1–5.

Wang Y, He K, Dai H, Meng W, Jiang J, Liu B, Chen Y, 2012. Scalable name lookup in NDN using effective name component encoding. In: IEEE ICDCS, 2012.
Wang Y, Pan T, Mi Z, Dai H, Guo X, Zhang T, Liu B, 2013. NameFilter: achieving fast name lookup with low memory cost via applying two-stage Bloom filters. In: IEEE INFOCOM, pp. 95–99.
Wang Y, Dai H, Zhang T, Meng W, Fan J, Liu B. GPU-accelerated name lookup with component encoding. Comput. Netw. 2013a;57:3165–77.
So W, Narayanan A, Oran D, 2013. Named data networking on a router: fast and DoS-resistant forwarding with hash tables. In: ACM/IEEE Symposium on Architectures for Networking and Communications Systems, 2013.
Xu Y, Li Y, Lin T, Wang Z, Niu W, Tang H, Ci S. A novel cache size optimization scheme based on manifold learning in content centric networking. J. Netw. Comput. Appl. 2014;37(January):273–81.