applied sciences

MDPI

Article

Route Prefix Caching Using Bloom Filters in Named Data Networking

Junghwan Kim, Myeong-Cheol Ko, Jinsoo Kim and Moon Sun Shin *

Department of Software Technology, Konkuk University, Chungcheongbuk-do 27478, Korea; jhkim@kku.ac.kr (J.K.); cheol@kku.ac.kr (M.-C.K.); jinsoo@kku.ac.kr (J.K.)

* Correspondence: msshin@kku.ac.kr

Received: 24 February 2020; Accepted: 23 March 2020; Published: 25 March 2020

Abstract: This paper proposes an elaborate route prefix caching scheme for fast packet forwarding in named data networking (NDN) which is a next-generation Internet structure. The name lookup is a crucial function of the NDN router, which delivers a packet based on its name rather than IP address. It carries out a complex process to find the longest matching prefix for the content name. Even the size of a name prefix is variable and unbounded; thus, the name lookup is to be more complicated and time-consuming. The name lookup can be sped up by using route prefix caching, but it may cause a problem when non-leaf prefixes are cached. The proposed prefix caching scheme can cache non-leaf prefixes, as well as leaf prefixes, without incurring any problem. For this purpose, a Bloom filter is kept for each prefix. The Bloom filter, which is widely used for checking membership, is utilized to indicate the branch information of a non-leaf prefix. The experimental result shows that the proposed caching scheme achieves a much higher hit ratio than other caching schemes. Furthermore, how much the parameters of the Bloom filter affect the cache miss count is quantitatively evaluated. The best performance can be achieved with merely 8-bit Bloom filters and two hash functions.

Keywords: prefix caching; Bloom filter; named data networking; name lookup; routing table

1. Introduction

The Internet has provided many application services for several decades by means of exchanging packets between two hosts, where one consumes information and the other produces it. In other words, it is a host-centric network in which a consumer generates a packet destined for a producer's IP address, and intermediate routers forward the packet based on the destination address. On the other hand, the current Internet with such a communication pattern is gradually revealing its limitation in providing rapidly increasing information-oriented services. Such information-oriented services are often location-independent, while the IP addresses are associated with specific sites. Moreover, the Internet has a burden that the domain name should be translated to the IP address before a packet is sent. Named data networking (NDN) [1,2] is one of the representative Internet structures of the future. It enables a growing number of information-based services to be deployed efficiently and effectively as packets are delivered based on the content name rather than the destination IP address. In the delivery of an NDN packet, the name of the information is more important than the location of an information provider.

A packet in NDN contains a name or a content name, indicating what information is of interest. A name has a human-friendly hierarchical structure made up of several components, each of which consists of character strings. An NDN router manages a forwarding information base (FIB) containing a great number of name prefixes with associated output faces. To deliver an incoming packet, the router searches for a prefix in the FIB that is matched with the name in the packet. This process is called a name lookup. The name lookup is accomplished by the component-wise matching in NDN,

while IP address lookup is accomplished by bit-wise matching on the Internet. Since multiple prefixes can be matched during the name lookup, the NDN router must perform the lookup as the longest prefix matching (LPM), which is to select the longest one among them as the best match. Similarly, Internet routers perform IP address lookup as LPM [3].

An IP address has a fixed size of 32 bits (or 128 bits) for IPv4 (or IPv6). Thus, an IP prefix is also limited in its size. On the other hand, in NDN, the length of a component is variable and unbounded, and the number of components in a name is also unlimited. Thus, the lengths of both the name and the name prefix are varied and unlimited. It is expected that the number of names and name prefixes will both increase significantly as application services grow rapidly. Hence, compared to the Internet, the FIB of NDN is more complex and is to be much larger.

The name lookup is challenging due to the difficulty of the LPM process, as well as the complex structure and the large size of the FIB. Even though it is a complicated and time-consuming task, it must be carried out so as to meet the link speed in NDN. In recent days, many researchers focused on high-speed name lookup techniques.

Most name lookup schemes are usually based on tries or hashing techniques. The trie is a hierarchical search structure widely used in IP lookup. Similarly, a name is hierarchically composed of components in NDN. For this reason, several trie-based name lookup techniques were researched [4–8]. Those techniques primarily focused on reducing the average number of memory accesses per lookup to speed up the name lookup. Alternatively, hashing techniques were used to achieve efficient name lookups [9–15], since a name consists of several components, each of which is a variable-length string. The hashing techniques are generally used to map variable-size data to a fixed-size value. Some methods dealt with the name lookup using tries and hashing together [16,17].

Routing for a packet delivery is performed according to the matching prefix in the FIB; thus, the prefix is also called a route prefix. A route prefix represents a common route for multiple names. Thus, the name lookup time can be greatly reduced if the route prefix is cached, which is called route prefix caching or prefix caching. A few prefix caching schemes were proposed to perform the name lookup efficiently [18,19].

Prefix caching is now challenging because there is a problem that needs to be solved. In the FIB, a leaf prefix is a prefix that has no descendant prefix, whereas a non-leaf prefix has some descendant. Non-leaf prefix caching may incur a problem not found for leaf prefix caching. If a matching prefix is non-leaf, its descendants can also be matched. Thus, non-leaf prefix caching may result in a wrong LPM result if there are some descendants which are not cached yet. In this paper, we present an elaborate method to avoid such an incomplete prefix caching problem that occurs when a non-leaf prefix is cached. Our technique caches non-leaf prefixes together with the branch information of their children. The branch information is represented by a Bloom filter [20] which is used to check whether a child exists on the branch.

The rest of this paper is organized as follows: we describe the name lookup for the packet forwarding in NDN in Section 2. We also overview the route prefix caching and the related works. Section 3 explains the proposed route prefix caching scheme using Bloom filters in NDN. In Section 4, the performance is evaluated in terms of several variables. Finally, Section 5 concludes the paper.

2. Background

2.1. Name Lookup in NDN Packet Forwarding

The NDN is composed of numerous nodes, which can be categorized into hosts and routers, as well as communication links connecting them, as the Internet is. A host can be either a consumer or a producer of the data. As a consumer, a host issues an interest packet containing a name of data. The router forwards the packet to an output face which is determined by the name. The producer replies with the data packet which contains the data associated with the name.

The router has a content store (CS) and a pending interest table (PIT) in addition to an FIB to deliver packets efficiently. The CS is a cache that supports the in-network caching feature of NDN. It allows a previously cached data packet to be reused when an interest packet arrives with the same

name as before. PIT allows only one packet to be delivered when two or more identical interest packets arrive. If the content name of an incoming interest packet is the same as one of the names already kept in the PIT, only its incoming interface is added on that entry in the PIT, and the packet is no longer forwarded.

Figure 1 shows an example of how interest packets and data packets are delivered when three consumers send the same interest in the NDN. In this figure, the numbers indicate the transmission sequence numbers. When the router x receives two interest packets (1 and 2), it stores their incoming interfaces in the PIT and transmits only one interest packet (3). Note that the data packet is delivered along the opposite direction to the transmission of the interest packet. When router y receives a data packet (5), the data are cached in the CS for future reuse. Later, when the router receives the interest packet (9), it can immediately transmit back the data packet (10) which is already cached in the CS.

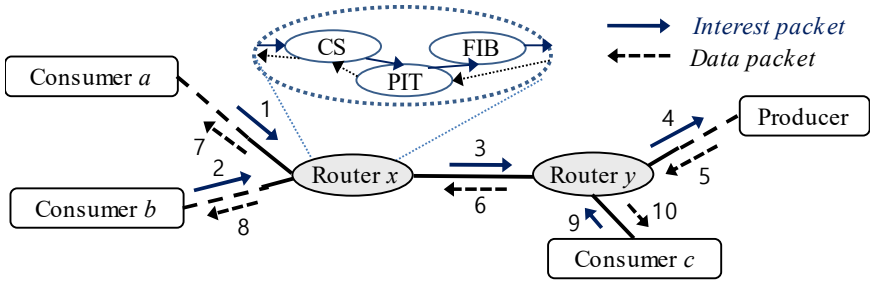

Figure 1. Example of packet forwarding when three consumers issue the identical interest packet in named data networking (NDN).

A content name comprises a series of components, each of which is a character string with a variable length. They are separated by slashes. For example, a name, /ride/wagon/horse, is composed of three components: ride, wagon, and horse. A name prefix is also represented as a sequence of components like a name. Therefore, the name lookup must perform matching between a name and a name prefix on a component basis rather than a bit basis, unlike IP address lookup. Thus, the name lookup is more complicated than the IP address lookup.

Since CS and PIT store names, exact matching (EM) is performed to find the exact name. On the other hand, FIB stores prefixes; hence, the name lookup must perform the LPM that selects the longest one among the prefixes matched with a name. For example, if prefixes /ride and /ride/wagon are stored in FIB, for a given interest, both prefixes, /ride and /ride/wagon, are matched for a given content name /ride/wagon/kw. Then, the LPM selects /ride/wagon as the final lookup result, whose length is the longest. The LPM is a more difficult and time-consuming process than EM. The performance of packet forwarding is highly dependent on the speed of the name lookup and is determined by the efficiency of the LPM. For these reasons, many previous studies on packet forwarding in NDN focused on improving the speed of the name lookup by increasing the efficiency of LPM [4–19].

2.2. Route Prefix Caching

A way to speed up the name lookup process is the route caching technique to store and reuse previous lookup results in the local cache associated with the FIB. The name lookup is to find an entry having the longest matching prefix for a content name. The entry is a pair of the prefix and its output face. The route caching can be classified into the content name caching and the name prefix caching according to the information stored in the cache. The content name caching stores a content name with the corresponding output face. It exploits a sort of temporal locality, but its effectiveness is limited because NDN already provides in-network caching. The name prefix caching stores a name prefix with the corresponding output face. In this caching scheme, a single name prefix in the cache can be matched as the LPM result for more than one content name. In other words, it also takes

advantage of spatial locality. Therefore, the name prefix caching is more effective than the content name caching.

Figure 2 shows some examples for the name prefix caching supposing that the FIB contains three prefixes *p*, *q*, and *r*. Assume that the name of the first packet is *name1* (= /ride/wagon/zo/fx), and its longest matching prefix *r* (= /ride/wagon) is cached. Then, the cached lookup result can be reused for fast forwarding when a subsequent content name is any of *name1*, *name4* (= /ride/wagon/zo/gr), or *name5* (= /ride/wagon/kw). Now, assume that the first content name is *name2* (= /ride/bike/du), and its lookup result *p* (= /ride) is cached. If the next content name is *name4* (= /ride/wagon/zo/gr), the prefix *p* is misleadingly regarded as its longest matching prefix due to the cache hit. In this case, the correct result for *name4* should be the prefix *r* but not the prefix *p*. The problem happens because the non-leaf prefix is cached but not all of its descendants are cached. This is called incomplete prefix caching. Note that the non-leaf caching does not always incur the problem. If the second content name was *name3* (= /ride/cy) in the above, the result prefix *p* would be the correct.

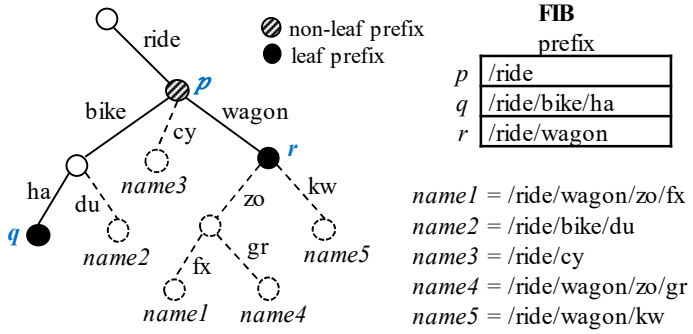

Figure 2. Example of name prefix caching in forwarding information base (FIB) containing three prefixes: *p*, *q*, and *r*.

2.3. Related Work

Several prefix caching techniques were studied for fast IP address lookup [21–23]. The prefix caching in IP address lookup suffers from the incomplete prefix caching problem like name lookup. Prefix expansion is one of the most efficient ways of resolving the incomplete prefix caching problem. This is accomplished whenever the longest matching prefix for a given IP address is a non-leaf. This creates virtual prefixes by expanding the prefix. The expanded prefixes do not have any dependency on other descendant prefixes of the longest matching prefix.

Complete prefix tree expansion (CPTE) and partial prefix tree expansion (PPTE) [21] were proposed to solve the incomplete prefix caching problem in IP address lookup. In these expansion techniques, all non-leaf prefixes should be expanded in the forwarding table. Thus, they enlarge the size of the original forwarding table. Moreover, it is impractical to apply those techniques to the name lookup because the degree of a prefix node is unbounded. The multi-zone pipelined cache (MPC) [22] performs either prefix caching with prefix expansion for short prefixes or IP address caching for long prefixes. The reverse route cache with minimal expansion (RRC-ME) [23] was also presented for prefix caching. It expands non-leaf prefixes dynamically without any modification of the original forwarding table. However, additional memory accesses may be required up to the extended length. Moreover, it is applicable only to trie-based IP lookup engines.

Recently, a few works were conducted on prefix caching for the name lookup. The prefix expansion technique is also used to resolve incomplete prefix caching issues in the proposed name lookup. On-the-fly caching [18] caches the most specific non-overlapping prefix in the name lookup. It dynamically expands the longest matching prefix so that the expanded prefixes have no overlap with any other prefixes in the FIB. It caches the expanded prefixes like RRC-ME in the IP address lookup. Thus, it has similarities with the pros and cons of RRC-ME. Furthermore, on-the-fly caching in the name lookup is more complex and difficult to implement, because the branches of the trie node

are not easily represented in fixed size, unlike IP address lookup. A new type of prefix caching scheme was also presented in the name lookup using the prefix expansion based on critical distance [19]. The critical distance means the maximum distance between a prefix and its descendant prefixes. This prefix expansion ensures independence between the expanded prefix and other descendant prefixes.

The incomplete prefix caching caused by caching a non-leaf prefix is a significant problem for both name lookup and IP address lookup. Most studies suggested their prefix expansion technologies in various ways. However, since the expanded prefix is always longer than the original prefix, the number of name occurrences that can be matched would be smaller than the original. This paper proposes a technique to solve incomplete prefix caching without any prefix expansion using a Bloom filter.

3. Route Prefix Caching

3.1. Naïve Approaches to Prefix Caching

In a route caching scheme, the matching result from the FIB is cached to be looked up faster for subsequent reuse. If the matching result is one of the leaf prefixes in the FIB, that result can be safely cached without causing any coherency problem. However, in the case of non-leaf prefixes, it causes incomplete prefix caching as mentioned in Section 2.2. Such a problem occurs when a non-leaf prefix is cached, but not all of its descendant prefixes are cached. In Figure 3, suppose that prefix *p* is cached, but its descendant prefixes *q*, *r*, *s*, and *t* are not cached. Now, an interest packet arrives with a name, /ride/bike/ha/xo. Obviously, *q* is the longest matching prefix for the name; however, *q* was not cached and *p* is misleadingly matched with it in the cache. To avoid misleadingly caching, we could consider two simple approaches: any non-leaf prefix is not allowed to be cached, or all the descendant prefixes should be cached when a non-leaf prefix is cached. The benefit of the first idea is restrictive. As shown in Figure 3, the non-leaf prefixes *p* and *r* cannot be cached. Furthermore, in the second approach, when *p* is cached, its descendants *q*, *r*, *s*, and *t* should be cached together. This results in a huge burden to cache all the descendant prefixes, in the aspects of cache replacement and miss penalty. Thus, both approaches are considered not to be viable in practice.

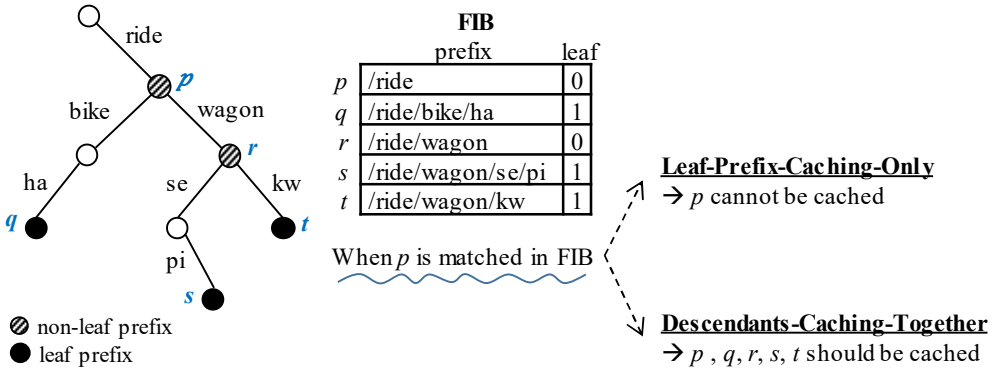

Figure 3. An example of FIB and two naïve caching approaches.

3.2. Prefix Caching Using Bloom Filters

In our route prefix cache, non-leaf prefixes are allowed to be cached without requiring their descendants to be cached. For that purpose, we provide a Bloom filter for each non-leaf prefix. The Bloom filter is a space-efficient data structure to test the membership of a given element [20]. In our scheme, the filter is used to test, with a given name, whether there exists a longer matching prefix in the FIB, other than the current matching prefix in the cache. The Bloom filter of a non-leaf prefix is a bit-vector of *m* bits. We use 8 bits for the bit-vector in this explanation. For each descendant of a non-

leaf prefix, the *i*-th bit is set whenever the position *i* is indicated by a hash value for a descendant. *k* different hash functions are used for that purpose.

Figure 4 illustrates how the Bloom filter is constructed and exploited. The non-leaf prefix *p* has three descendants, *q*, *s* and *t*, each of which contributes to set some bit in the Bloom filter. In Figure 3, BF_P denotes the Bloom filter of the prefix *p*. For *p*'s descendant prefix /ride/bike/ha (= *q*), the first component right after /ride (= *p*), is "bike"; thus, the hash values for "bike" are computed to obtain set positions in BF_P. Note that two hash functions, h_1 and h_2, are used for that purpose. The hash values are 1 and 4; hence, the first and fourth bits are set to 1 in BF_P. Similarly, the descendants, *s* and *t* also contribute to set the Bloom filter of *p*. Both descendants have the first common component right after *p*, which is "wagon". The hash values of "wagon" are 4 and 7; hence, the fourth and seventh bits are set in BF_P. Leaf prefixes also have Bloom filters, but they are always 0 because the leaf prefix does not have any descendants.

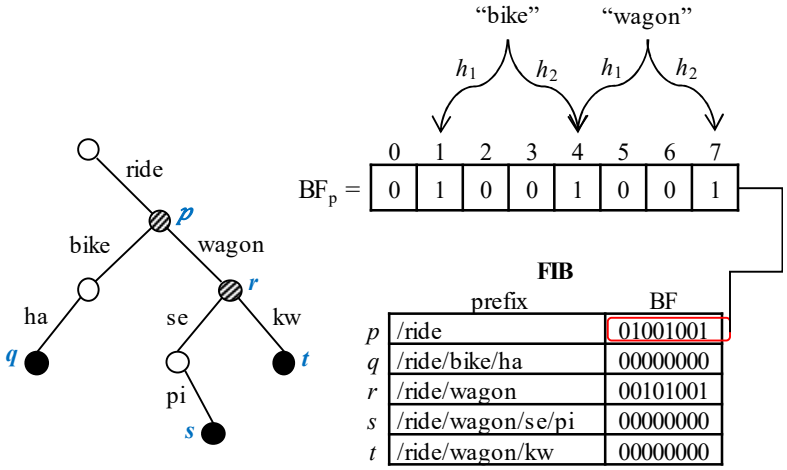

Figure 4. The Bloom filters of prefixes to represent the existence of their descendants.

Now, suppose that we are trying to look up a content name, /ride/bike/ha/ty, and there is *p* (= /ride) in the cache, but not *q* (= /ride/bike/ha) in the cache. If the content name was looked up in the FIB, both prefixes would be matched, and the longest matching result would be *q*. However, only *p* is matched in the cache. BF_P is now queried to check whether there is a descendant of *p* which is possibly matched with the content name. The first component of the content name right after *p* is "bike" and the bit positions of BF_P indicated by h_1 ("bike") and h_2 ("bike") are 1 and 4, respectively. Since both bits are set to 1, which indicates there possibly exists another longer matching prefix, the cache regards *p* as not the longest matching result by treating it as a cache miss. Conversely, if one of the two bits was 0, there could not be any longer matching prefix. In that case, *p* is the longest matching result and a cache hit occurs.

While all leaf prefixes are cacheable, some non-leaf prefixes are not cacheable. To segregate non-cacheable prefixes, Bloom filters and the information of the prefix length are used. Given a content name *c*, its longest matching prefix *p* in the FIB is classified into four types according to *p*'s Bloom filter and length. Figure 5 shows the four categories of the matching prefixes in the FIB. All the prefixes corresponding to Type 1, 2, and 3 are cacheable, while those corresponding to Type 4 are non-cacheable. If the matching prefix *p* has a Bloom filter for which bits are all set to 0s, it is clearly a leaf prefix and considered to be of Type 3. Now, let us consider the case where *p* is a non-leaf prefix. Let $p < c$ denote that the length of *c* is longer than that of *p*. Furthermore, $BF_P(x)$ represents whether the bits indicated by *k* hash values are all 1s or not, where *x* is the next component after *p*. If $BF_P(x)$ asserts a negative, this implies that some of the bits may be 0. If $p < c$ and $BF_P(x)$ is a negative, then this means that *c* does not share any path from *p* with any of *p*'s descendants. In this case, *p* corresponds to Type 2 and is cacheable. In case that *p* is the same as *c*, *p* corresponds to Type 1 and is cacheable regardless of whether it is a leaf or non-leaf. Lastly, if $p < c$ and $BF_P(x)$ is a positive, it

corresponds to Type 4 and p is not cached. Even if p is allowed to be cached, the content name c will not be matched with p in the cache on the subsequent queries.

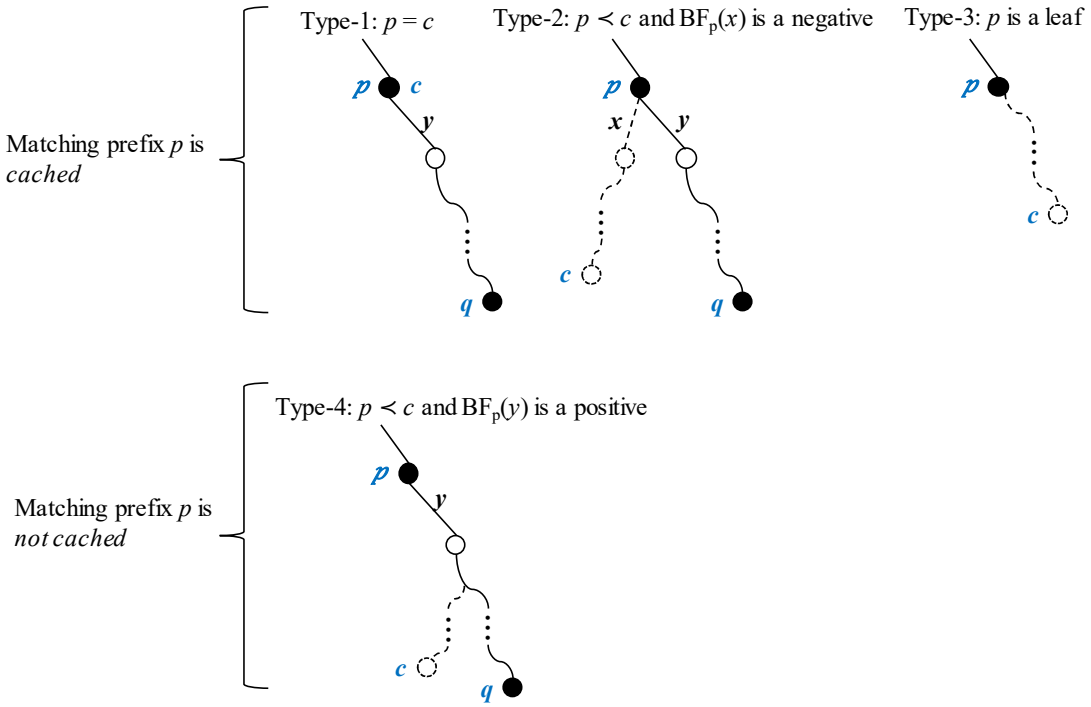

Figure 5. Type of matching prefix for a given name c.

3.3. Design of Route Prefix Cache Architecture

The name lookup system is composed of two parts, a Bloom filter-based route prefix cache and an FIB table, as shown in Figure 6. At the first step of the name lookup, a given name is looked up in the route prefix cache. If a cache hit occurs, the lookup procedure terminates at the step. Otherwise, on a cache miss, the FIB table is consulted. Note that the cache miss occurs not only in the event that there is no matching prefix in the cache, but also in the event that the Bloom filter of the matching prefix asserts a positive. On a cache miss, the matched entry is fetched from the FIB and stored in the cache. An associative memory is provided for storing prefixes to facilitate prefix matching in the cache. The other fields, such as the length, the Bloom filter, and the output face of a prefix, are stored in a conventional but fast memory. Those fields are accessed when the corresponding prefix is matched with a given name.

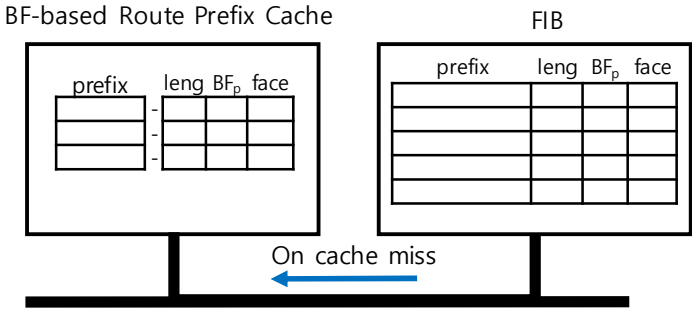

Figure 6. The organization of overall name lookup.

Ternary content addressable memory (TCAM) may be a candidate for the associative memory of the cache. TCAM supports a parallel associative search, and a variable-length key can be stored in

it and searched for. Using the TCAM, the variable-length prefix can be stored and searched for in our cache. The TCAM is an expensive memory; however, the proposed cache can be configured with a feasible size. Figure 7 shows how our Bloom filter-based route prefix cache works to find a matching result. For a given name, /ride/wagon/se/ty, the longest matching prefix /ride/wagon is found in the TCAM. The corresponding information, the length, the Bloom filter, and the face, are extracted from an SRAM. The length value is used for extracting the next component, "se". Then, the bits in the Bloom filter indicated by h_1 ("se") and h_2 ("se") are checked to see if those bits are all set to 1. If not, a cache hit is raised, and the face number is delivered as the result.

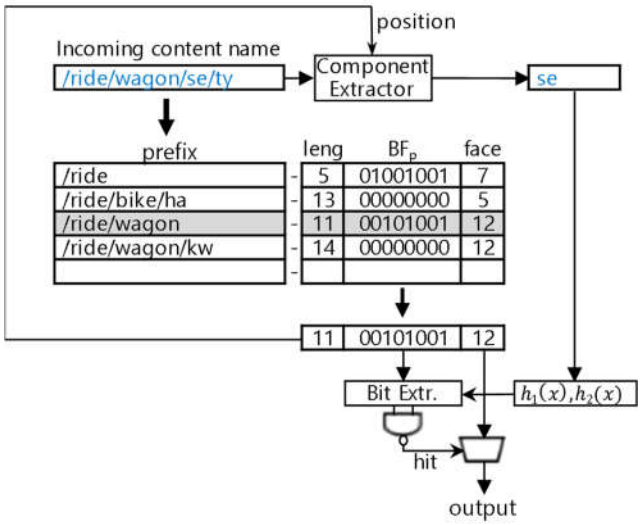

Figure 7. The architecture of a Bloom filter-based route prefix cache.

3.4. Construction of Bloom Filters

All the prefixes in a FIB table have their Bloom filters. Each prefix's Bloom filter is constructed using the branch information of its children. Algorithm 1 describes how to construct the Bloom filters in FIB. For a given *fib*, which is an FIB table, and the number of hash functions k, an array of Bloom Filters, BF, is constructed. Each of the hash functions, h_1, h_2, …, h_k, maps component strings to {0, 1, 2, …, m − 1} where m is the size of a Bloom filter.

For each prefix in *fib*, its parent prefix is identified as the form of /c_1/ c_2/…/c_i where the prefix is /c_1/ c_2/…/c_n and $i < n$. For example, in Figure 4, if prefix is currently /ride/bike/ha, then parent would be /ride. The *next_component* after *parent* is c_{i+1}. In this example, it is "bike". Now, hash functions, h_1, h_2, …, h_k, are computed with the input "bike" in turn, and the bit indicated by each hash value is set to 1 using bitwise-or. Supposing that $k = 2$ and h_1 ("bike") = 1 and h_2 ("bike") = 4, we could say that the parent's Bloom filter is 01001000 if the Bloom filter size is 8. Note that the Bloom filter is not completely constructed if there remain other children which are not processed yet. Whenever a child is processed, their parent's Bloom filter is updated. Thus, the Bloom filter of /ride is updated when its child, /ride/wagon, is processed. If h_1 ("wagon") = 4 and h_2 ("wagon") = 7, then the parent's Bloom filter is updated as follows using bitwise-or: 01001000 → 01001000 → 01001001.

Algorithm 1: Constructing Bloom filters in FIB

function(*fib*, *k*)
Input: *fib* is a prefix table sorted in component-level order
k is the number of hash functions to indicate bit positions in BF
Output: BF is the array of Bloom filters, each of which corresponds to an *fib* entry
1: **foreach** *prefix* **in** *fib* **do**
2: (c_1, c_2, …, c_n) is the list of components in *prefix*

```
3:      for i ← n - 1 to 1 do
4:          parent ← '/' c1 '/' c2 '/' … '/' ci
5:          if parent in fib then
6:              break
7:      done
9:      p ← parent's index in fib
10:     next_component ← ci+1
11:     for j ← 1 to k do
12:         BF[p] ← bitwise-or(BF[p], lshift(1, hj(next_component)))
13:     done
14: done
```

4. Evaluation

For the experiment, we used the route prefixes that came from a name set *dmoz* [24] because there is currently no real route prefix table for NDN. Since the name set was originated from Internet URLs, some conversions were made for the experiment. For instance, some intermediate nodes which are not prefixes were changed to prefixes randomly. Table 1 shows the FIB table containing 3,644,942 prefixes in total. The FIB table consists of leaf prefixes and non-leaf prefixes, which account for 77.6% and 22.4% of the total, respectively. Figure 8 shows the frequency of prefixes according to their degrees. The degree of a prefix represents the number of the first components appearing just after the prefix. This is related to the false positive ratio of a Bloom filter. All the leaf prefixes no doubt have a degree of 0, and most non-leaf prefixes have a degree of 1, which is 94.0% of the total non-leaf prefixes. The mean value of the degree is 2.023 for non-leaf prefixes; however, if an outlier is excluded, the mean value becomes 1.814.

Table 1. The number of prefixes in the FIB which was used in the experiment.

Leaf Prefixes	Non-Leaf Prefixes	Total
2,827,582 (77.6%)	817,360 (22.4%)	3,644,942 (100.0%)

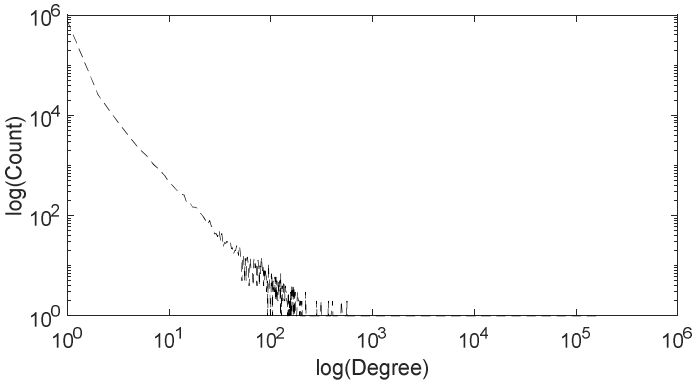

Figure 8. Frequency of prefixes according to their degrees in log scale.

Six traces were used for our experiment, each of which was generated stochastically using Zipf distribution. Internet traces are known to follow Zipf-like behavior [25].

$$P(r) \sim \frac{1}{r^{\alpha}} \quad (1)$$

where r is the rank in the frequency of packets. When the generated traces for α = { 0.6, 0.7, 0.8, 0.9, 1.0, 1.1 } were used, the cache hit ratio in our scheme was as shown in Table 2. As α increases, the high-ranking packets have a higher frequency of occurrences in a trace. Generally, a high cache hit ratio is achieved with a large cache and in the trace of a high α value. In cases where the cache size is greater than 160 KB, the cache hit ratio is not affected much by the value of α.

Table 2. Cache hit ratio in the traces generated using α.

Cache Size	α					
(kB)	**0.6**	**0.7**	**0.8**	**0.9**	**1.0**	**1.1**
20	0.20705	0.28115	0.37371	0.47811	0.58608	0.68681
40	0.32729	0.40353	0.49149	0.58517	0.67663	0.75890
80	0.50530	0.56871	0.63739	0.70672	0.77201	0.82941
160	0.76262	0.78908	0.81696	0.84517	0.87268	0.89801
320	0.87198	0.87757	0.88535	0.89524	0.90708	0.92020
640	0.87198	0.87757	0.88535	0.89524	0.90708	0.92020

Non-leaf prefix caching without any supplementary method causes an incomplete prefix caching problem. A few caching schemes to avoid the problem are compared in Figure 9. Our Bloom filter-based scheme is denoted by "bfilter", and the Leaf-Prefix-Caching-Only scheme, which allows only leaf prefixes to be cached, is denoted by "lpco". The No-Prefix-Caching scheme, which caches a content name itself instead of its matching prefix, is denoted by "npc". "lpco+npc" denotes a scheme which caches a matching leaf prefix or a content name in cases where the matching prefix is a non-leaf prefix. The Bloom filter-based prefix cache is far superior to the other schemes in the cache hit ratio. "npc" no doubt shows the worst cache hit ratio because it does not cache prefixes and, thus, it does not exploit any spatial locality. The cache hit ratio of "lpco+npc" is worse than that of "lpco" up to the cache size of 4096 entries. The content name caching requires more cache entries to utilize a similar level of the locality. Therefore, "lpco+npc" results in more conflict misses than "lpco" in a small cache. However, if the cache is sufficiently large, "lpco+npc" can utilize the space and reduce the cache misses, even in cases where the matching prefix is a non-leaf prefix.

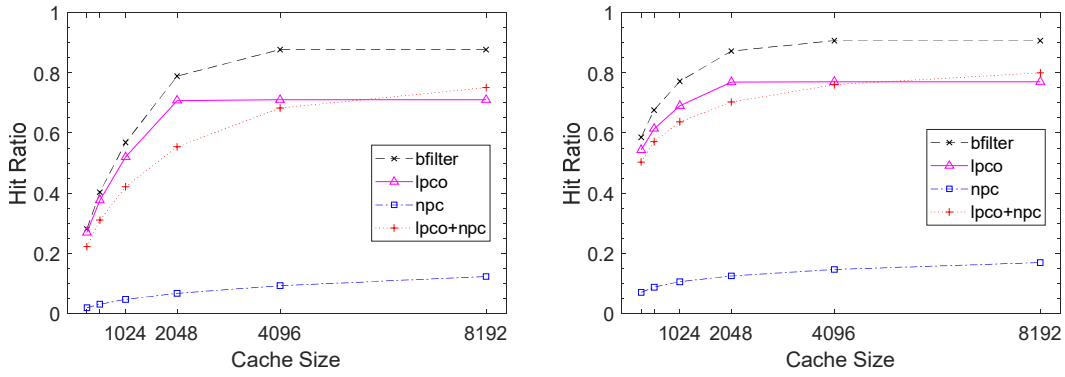

Figure 9. Comparison of caching schemes for cache hit ratio in the traces with (**a**) α = 0.7, and (**b**) α = 1.0.

A cache miss occurs when there is no matching prefix in the cache, or when the matching prefix's Bloom filter asserts a positive. The positive assertion of a Bloom filter implies that there is possibly a longer matching prefix in the FIB other than the cache. However, the Bloom filter may misleadingly assert a positive. The probability of the false positive is affected by filter size m, the number of hash functions k, and the number of elements n to be contained. It is calculated as follows [26]:

$$P_f = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k. \quad (2)$$

In our scheme, the number of elements n corresponds to the degree of a prefix. Hence, it is close to 1 (94%) as previously discussed, and the mean value of it is 1.814. When the filter size m is either 4 or 8, and the number of hash functions k is 1, 2, or 3, the probability of the false positive is as summarized in Table 3. In general, the false positive ratio decreases as the filter size m increases. However, the false positive ratio does not always decrease as k increases. For instance, the false positive ratio increases as k increases for $n \geq 2$ and $m = 4$.

Table 3. False positive ratio for Bloom filters with size m, the number of hash functions k, and the degree of a prefix n.

m	k	n				
		1	**2**	**3**	**4**	**5**
4	1	0.25000	0.43750	0.57813	0.68359	0.76270
4	2	0.19141	0.46730	0.67572	0.80980	0.89054
8	1	0.12500	0.23438	0.33008	0.41382	0.48709
8	2	0.05493	0.17125	0.30383	0.43085	0.54306
8	3	0.03596	0.16747	0.34203	0.50928	0.64736

Table 4 shows the cache hit ratio over the parameters of the Bloom filters, m and k. If m is 4, the cache hit ratio is rather lower on $k = 2$ than $k = 1$. For $m = 8$, the cache hit ratio increases as k grows. However, the gain from a Bloom filter with $k = 3$ is insubstantial when compared to a Bloom with $k = 2$.

Table 4. Cache hit ratio with Bloom filters having size m and the number of hash functions k.

Cache Size	**Bloom Filter Parameters**				
	$m = 4, k =$ **1**	$m = 4, k =$ **2**	$m = 8, k =$ **1**	$m = 8, k =$ **2**	$m = 8, k =$ **3**
256	0.57660	0.57663	0.58304	0.58608	0.58690
512	0.66347	0.66320	0.67254	0.67663	0.67757
1024	0.75574	0.75514	0.76698	0.77201	0.77309
2048	0.85337	0.85257	0.86665	0.87268	0.87393
4096	0.88660	0.88567	0.90075	0.90708	0.90840
8192	0.88660	0.88567	0.90075	0.90708	0.90840

The matching type in FIB is classified into four categories, as described in Section 3.2. In Type 1, the matching prefix is the same as the content name. If the matching prefix is a leaf prefix, then it is of Type 3. If the matching prefix is a non-leaf prefix but its Bloom filter asserts a positive, it is of Type 4. The matching prefix, which is a non-leaf prefix, is of Type 2 if its Bloom filter asserts a negative. Figure 10a shows the distribution of each matching type in the traces with α ranging from 0.6 to 1.1. Type 3 is the dominating matching type in all the traces. Type 2 and Type 4 occupy a small portion, and the packets of Type 1 are very few. Figure 10b shows the cache hit count in each type. The cache hit count increases as the cache size increases, especially in Type 3, because the packets of Type 3 are substantial in the trace. There is no cache hit in Type 4 because it is not cached.

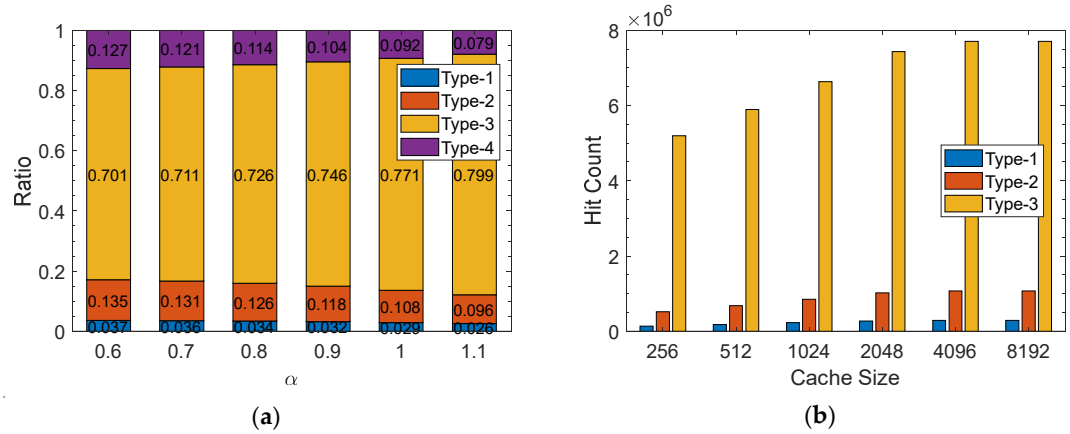

Figure 10. Observations with regard to the type of matching prefixes: (**a**) distribution of matching prefixes in the traces with α, from 0.6 to 1.1; (**b**) cache hit count on matching prefix types.

The conventional cache misses can be classified into compulsory, capacity, and conflict misses. In our cache scheme, a conflict miss does not occur because the cache memory is fully associative. The capacity and compulsory misses occur when there is no matching prefix in the cache due to the replacement or being never loaded before, respectively. In the Bloom filter-based cache scheme, we have two more miss types: BF miss and non-cacheable miss. In the BF miss, there is a matching prefix in the cache, but its Bloom filter asserts a positive. The non-cacheable miss occurs if the matching prefix in the FIB is not cached because it corresponds to Type 4. The capacity misses do not include the non-cacheable misses in this context.

Figure 11a shows the distribution of the cache miss counts according to their types. For a small cache, the capacity misses and the compulsory misses are a substantial portion of the total misses. There are very few capacity and compulsory misses if the cache size is greater than 2048. The BF misses and the non-cacheable misses become a relatively large portion as the cache size increases. The actual miss count for various Bloom filter configurations is shown in Figure 11b. There is no doubt that the compulsory and the capacity misses do not change over the Bloom filter configuration. The BF miss count is lower when m is 8 than when m is 4. For m = 8, the BF miss counts are nearly similar when compared for k = 2, 3. Furthermore, for m = 4, the BF miss counts are similar for k = 1, 2.

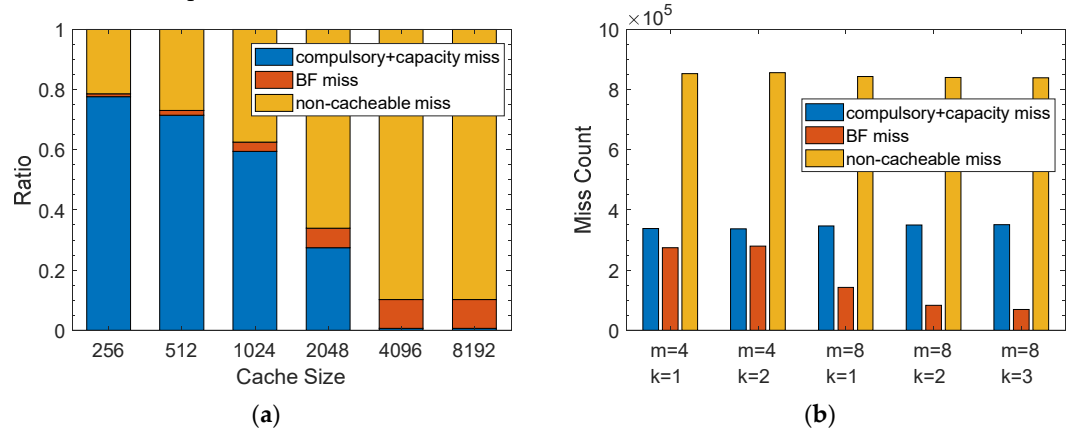

Figure 11. Observations with regard to the cause of cache miss: (**a**) distribution of cache miss count; (**b**) cache miss count against the Bloom filters with size m and the number of hash functions k when the cache size is 2048.

5. Conclusion

In this paper we proposed a novel route prefix caching scheme for NDN. The fundamental idea is to use a Bloom filter to test whether there is a longer matching prefix in the FIB. The route prefix

cache can be used to reduce the name lookup latency. In NDN, the name lookup process is required to find the longest matching prefix as the result. Prefix caching is challenging because the matching result from a cache would be incorrect unless all the matching prefixes are already in the cache. In the event that the longest matching prefix is not cached but the shorter one is cached, an incorrect result can be issued by the cache.

To get around the problem, the content name itself can be simply cached instead of the matching prefix. However, in that scheme, the cache hit occurs only when the given name is exactly the same as the key. Therefore, the cache hit ratio would be much lower than for other prefix caching schemes with the same capacity. Caching only for leaf prefixes is another way to avoid incorrect prefix matching. In that scheme, non-leaf prefixes would never be cached; hence, the caching effect is restrictive. Our design goal is to cache non-leaf prefixes without incurring an incorrect result. A Bloom filter is provided for each prefix to test whether there is a longer matching prefix.

We devised a Bloom filter-based prefix cache considering two key elements. Firstly, our design does not require much space. Non-leaf prefix caching could be made efficient just by adding a 4-bit or 8-bit Bloom filter onto each prefix. Secondly, miss penalty and hit time are not significantly affected by the Bloom filter. On a cache miss, the Bloom filter is fetched together with the matching prefix from the FIB. The miss penalty for delivering the additional 4-bit or 8-bit filter is not a heavy load. Hashing for the Bloom filter affects the hit time, but it is only one-component hashing and requires little processing time.

The experimental result shows that the cache hit ratio of the Bloom filter-based prefix cache is superior to others even if it has small filters and few hash functions. The probability of the false positive of a Bloom filter can be analyzed through the filter size, the number of hash functions, and the number of elements. The cache miss counts due to Bloom filter's positive assertion were compared to the experiment result, and the 8-bit filter with two hash functions was considered to be suitable for our prefix caching scheme.

Author Contributions: Conceptualization, Junghwan Kim; formal analysis, Junghwan Kim and M.S.S.; investigation, M.-C.K. and Jinsoo Kim; methodology, Jinsoo Kim; validation, M.-C.K. and M.S.S.; writing—original draft, Junghwan Kim; writing—review and editing, Jinsoo Kim and M.S.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Conflicts of Interest: The authors declare no conflicts of interest.

References

1. Zhang, L.; Afanasyev, A.; Burke, J.; Jacobson, V.; Crowley, P.; Papadopoulos, C.; Wang, L.; Zhang, B. Named data networking. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 66–73. doi:10.1145/2656877.2656887.
2. Saxena, D.; Raychoudhury, V.; Suri, N.; Becker, C.; Cao, J. Named data networking: A survey. *Comput. Sci. Rev.* **2016**, *19*, 15–55. doi:10.1016/j.cosrev.2016.01.001.
3. Ruiz-Sanchez, M.A.; Biersack, E.W.; Dabbous, W. Survey and taxonomy of IP address lookup algorithms. *IEEE Netw.* **2001**, *15*, 8–23. doi:10.1109/65.912716.
4. Wang, Y.; Dai, H.; Jiang, J.; He, K.; Meng, W.; Liu, B. Parallel name lookup for named data networking. In Proceedings of the 2011 IEEE Global Telecommunications Conference (GLOBECOM 2011), Houston, TX, USA, 5–9 December 2011; pp. 1–5. doi:10.1109/GLOCOM.2011.6134161.
5. Wang, Y.; He, K.; Dai, H.; Meng, W.; Jiang, J.; Liu, B.; Chen, Y. Scalable name lookup in NDN using effective name component encoding. In Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS), Macau, China, 18–21 June 2012; pp. 688–697. doi:10.1109/icdcs.2012.35.
6. Lee, J.; Lim, H. A new name prefix trie with path compression. In Proceedings of the 2012 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), 2016; pp. 1–4. doi:10.1109/icce-asia.2016.7804781.
7. Li, D.; Li, J.; Du, Z. An improved trie-based name lookup scheme for named data networking. In Proceedings of the 2016 IEEE Symposium on Computers and Communication (ISCC), Messina, Italy, 27–30 June 2016; pp. 1294–1296. doi:10.1109/iscc.2016.7543915.

8. Ghasemi, C.; Yousefi, H.; Shin, K.G.; Zhang, B. A fast and memory-efficient trie structure for name-based packet forwarding. In Proceedings of the 2018 IEEE 26th International Conference on Network Protocols (ICNP), Cambridge, UK, 25–27 Sept. 2018; pp. 302–312. doi:10.1109/ICNP.2018.00046.
9. So, W.; Narayanan, A.; Oran, D.; Wang, Y. Toward fast NDN software forwarding lookup engine based on hash tables. In Proceedings of the 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Austin, TX, USA, 29–30 October 2012; pp. 85–86. doi:10.1145/2396556.2396575.
10. Wang, Y.; Pan, T.; Mi, Z.; Dai, H.; Guo, X.; Zhang, T.; Liu, B.; Dong, Q. Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters. In Proceedings of the 2013 INFOCOM, Turin, Italy, 14–19 April 2013; pp. 95–99. doi:10.1109/infcom.2013.6566742.
11. Lee, J.; Shim, M.; Lim, H. Name prefix matching using bloom filter pre-searching for content centric network. *J. Netw. Comput. Appl.* **2016**, *65*, 36–47. doi:10.1016/j.jnca.2016.02.008.
12. Wang, Y.; Qi, Z.; Dai, H.; Wu, H.; Lei, K.; Liu, B. Statistical Optimal Hash-based Longest Prefix Match. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Beijing, China.18–19 May 2017; pp. 153–164. doi:10.1109/ancs.2017.29.
13. Dai, H.; Lu, J.; Wang, Y.; Pan, T.; Liu, B. BFAST: High-Speed and Memory-Efficient Approach for NDN Forwarding Engine. *IEEE/ACM TON* **2017**, *25*, 1235–1248. doi:10.1109/tnet.2016.2623379.
14. Yu, Y.; Belazzougui, D.; Qian, C.; Zhang, Q. Memory-efficient and ultra-fast network lookup and forwarding using othello hashing. *IEEE/ACM Trans. Netw.* **2018**, *26*, 1151–1164. doi:10.1109/TNET.2018.2820067.
15. Lee, J.; Byun, H.; Lim, H. Dual-load Bloom filter: Application for name lookup. *Comput. Commun.* **2020**, *151*, 1–9. doi:10.1016/j.comcom.2019.12.029.
16. Quan, W.; Xu, C.; Guan, J.; Zhang, H.; Grieco, L.A. Scalable name lookup with adaptive prefix bloom filter for named data networking. *IEEE Commun. Lett.* **2014**, *18*, 102–105. doi:10.1109/lcomm.2013.112413.132231.
17. Kim, J.; Ko, M.C.; Shin, M.S.; Kim, J. Scalable Name Lookup for NDN Using Hierarchical Hashing and Patricia Trie. *Appl. Sci.* **2020**, *10*, 1023. doi:10.3390/app10031023.
18. Chen, X.; Zhang, G.; Cui, H. Investigating route cache in named data networking. *IEEE Commun. Lett.* **2017**, *22*, 296–299. doi:10.1109/LCOMM.2017.2769680.
19. Kim, J.; Kim, J. An Efficient Prefix Caching Scheme for Fast Forwarding in Named Data Networking. *Stud. Inform. Control* **2018**, *27*, 175–182. doi:10.24846/v27i2y201805.
20. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. doi:10.1145/362686.362692.
21. Liu, H. Routing prefix caching in network processor design. In Proceedings of the Tenth International Conference on Computer Communications and Networks, Scottsdale, AZ, USA, 15–17 October 2001; pp. 18–23. doi:10.1109/ICCCN.2001.956214.
22. Kasnavi, S.; Berube, P.; Gaudet, V.; Amaral, J.N. A cache-based internet protocol address lookup architecture. *Comput. Netw.* **2008**, *52*, 303–326. doi:10.1016/j.comnet.2007.08.010.
23. Akhbarizadeh, M.J.; Nourani, M. Efficient prefix cache for network processors. In Proceedings of the 12th Annual IEEE Symposium on High Performance Interconnects, Stanford, CA, USA, 27–27 August 2004; pp. 41–46. doi:10.1109/CONECT.2004.1375199.
24. Directory of World Wide Web. Available online: https://web.archive.org/web/20170312160530/http://rdf.dmoz.org/rdf/content.rdf.u8.gz (accessed on 12 March 2017).
25. Shi, W.; MacGregor, M.H.; Gburzynski, P. Synthetic Trace Generation for the Internet: An Integrated Model. In Proceedings of the 2004 Int. Symposium on Performance Evaluation of Computer and Telecommunication Systems, San Jose, CA, USA, 25–29 July 2004; pp. 471–477. doi:10.1109/WWC.2001.990756.
26. Broder, A.Z.; Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Math.* **2004**, *1*, 485–509. doi:10.1080/15427951.2004.10129096.