A Novel Prefix Cache with Two-Level Bloom Filters in IP Address Lookup

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

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

Received: 8 September 2020; Accepted: 13 October 2020; Published: 15 October 2020

Abstract: Prefix caching is one of the notable techniques in enhancing the IP address lookup performance which is crucial in packet forwarding. A cached prefix can match a range of IP addresses, so prefix caching leads to a higher cache hit ratio than IP address caching. However, prefix caching has an issue to be resolved. When a prefix is matched in a cache, the prefix cannot be the result without assuring that there is no longer descendant prefix of the matching prefix which is not cached yet. This is due to the aspect of the IP address lookup seeking to find the longest matching prefix. Some prefix expansion techniques avoid the problem, but the expanded prefixes occupy more entries as well as cover a smaller range of IP addresses. This paper proposes a novel prefix caching scheme in which the original prefix can be cached without expansion. In this scheme, for each prefix, a Bloom filter is constructed to be used for testing if there is any matchable descendant. The false positive ratio of a Bloom filter generally grows as the number of elements contained in the filter increases. We devise an elaborate two-level Bloom filter scheme which adjusts the filter size at each level, to reduce the false positive ratio, according to the number of contained elements. The experimental result shows that the proposed scheme achieves a very low cache miss ratio without increasing the number of prefixes. In addition, most of the filter assertions are negative, which means the proposed prefix cache effectively hits the matching prefix using the filter.

Keywords: prefix caching; two-level Bloom filter; prefix expansion; IP address lookup; Internet router

1. Introduction

The Internet has been providing more and more various types of services at a high speed for several decades. Accordingly, the Internet router, which is an important component of the Internet, has also been required to provide higher performance. The router determines the next hop for each incoming packet based on its destination IP address. To make this decision, it performs the IP address lookup, which is to search for the best matching prefix in the forwarding information base (FIB). For a given IP address, there can be multiple matching prefixes in the FIB, and the longest matching prefix (LMP) among them should be selected as the best result. However, the process of finding an LMP is very complex and consumes a lot of cycles. Therefore, many researchers have been working on high-performance IP address lookup [1–11].

Caching is one of the prominent techniques which can be applied to improve the performance of the IP address lookup. The caching techniques in the IP address lookup can be classified into IP address caching [12] and prefix caching [13]. The former uses only temporal locality to cache the 32-bit IP address and reuses the lookup result when the same address is given. On the other hand, the latter caches a prefix which represents a network address covering a range of IP addresses and reuses the result when one of the addresses in that range is given. In other words, it utilizes not only temporal
locality but also address-spatial locality. Thus, the prefix caching can achieve better performance by utilizing locality more effectively than the IP address caching.

Unfortunately, the prefix caching can cause a serious problem if non-leaf prefixes are allowed to be cached without changing. If a non-leaf prefix is cached but its longer descendant prefixes are not, it is not guaranteed that the result of a cache hit is the correct LMP because a longer matching prefix may exist in the FIB even though it is not in the cache. We call this problem the incomplete caching problem [14].

Prefix expansion [13] is one of the representative ways to resolve the incomplete caching problem. It is to expand a non-leaf prefix to leaf prefixes so that they become independent of other descendant prefixes. The prefix expansion in the original FIB increases the size of the FIB, making it difficult and expensive to implement. Besides, the expanded prefix is more difficult to update than the original prefix. In the case of prefix expansion in the cache, multiple expanded prefixes can be cached for one LMP, and an increase in the number of occupied entries can result in performance degradation due to capacity misses. Moreover, the expanded prefix may be less utilized in the address-spatial locality because the range of addresses it covers is narrower than the original prefix.

Our scheme can cache a non-leaf prefix in itself without sacrificing the spatial locality and resolves the incomplete caching problem by using the supplementary information for each prefix to indicate the existence of its child prefixes in the FIB. The scheme uses Bloom filters [15] to represent that information. The Bloom filters have been widely used for checking a membership in the longest prefix matching [9–11], but not in the prefix caching for fast IP address lookup. In our scheme, they efficiently determine whether there exists a matchable child prefix for the current matching prefix in a cache. The drawback of the Bloom filter is misleadingly indicating there is a member even though it is not true. Moreover, the false positive ratio increases as the number of contained elements grows. In this paper, we propose a two-level Bloom filter-based prefix caching scheme which keeps a low false positive probability by controlling the number of contained elements and adjusting the size of each filter within the total size.

The rest of the paper is organized as follows. In Section 2, we briefly describe the prefix caching for fast packet forwarding and discuss the research related to the prefix expansion techniques to overcome a problem that the prefix caching may cause. Section 3 introduces the Bloom filter and explains how to use it for solving the incomplete caching problem in the prefix caching. Section 4 proposes a two-level Bloom filter-based prefix caching (BFPC) scheme in IP address lookup. In Section 5, the performance is evaluated in terms of the cache miss ratio and other parameters. Finally, we conclude the paper in Section 6.

2. Background

2.1. Prefix Caching and Prefix Expansion

A prefix cache stores the prefixes which are the destination network addresses of the incoming packets. The prefix represents a network address, i.e., a range of IP addresses. Therefore, the prefix caching scheme can effectively exploit the prefix to match several IP addresses. In other words, the prefix caching exploits spatial locality as well as temporal locality. Thus, the prefix caching achieves a higher cache hit ratio than the IP address caching. Although prefix caching is more effective, it causes the incomplete caching problem if non-leaf prefixes can be cached.

Figure 1 shows an example of the incomplete caching problem that may induce a wrong lookup result when a non-leaf prefix is cached. In this figure, there are a non-leaf prefix \( p = 1^* \) and two leaf prefixes \( a = 10010^* \) and \( b = 100110^* \). Let us assume that the length of the IP address is 7 bits instead of 32 for simplicity and that two packets arrive in turn. If the destination IP address of the first packet is \( IP_1 = 1000000 \), the LMP becomes \( p = 1^* \). Now, the prefix \( p \) is cached for the first time. Supposing that the address of the second packet is \( IP_2 = 1001000 \), the prefix \( p \) is hit in the cache. However, the LMP for \( IP_2 \) is not \( p = 1^* \) but \( a = 10010^* \), so this cache hit yields the incorrect result. If the second IP address
were \( IP_3 = 1001100 \), it could also hit the prefix \( p \) and cause the incomplete caching problem. Now, suppose that the second IP address was \( IP_4 = 1001111 \). Then, the hit result \( p \) should be the correct result because the prefix \( p \) is the LMP for \( IP_4 \).

There are three ways to deal with the incomplete caching problem of non-leaf prefixes. The first way is not to cache non-leaf prefixes but to cache only leaf prefixes. In the example in Figure 1, the LMP \( p \) for the first packet should not be cached. This method may limit performance gain through caching. The second way is to cache a non-leaf prefix together with all of its descendant prefixes at the same time. In this example, when prefix \( p \) is cached, its descendant prefixes \( a \) and \( b \) should be also cached. This method significantly reduces the cache efficiency by caching a lot of unnecessary prefixes in advance. The third way is to expand the non-leaf prefix into leaves so that the expanded prefixes do not overlap with other descendant prefixes. This method is called prefix expansion. Most of the previous works make use of this prefix expansion technique [13,14,16–18].

Figure 2 illustrates an example of how to resolve the incomplete caching problem using prefix expansion. The prefix may be statically expanded in the FIB for later caching, or dynamically expanded at the time of caching. This figure shows an example to explain the dynamic prefix expansion. In Figure 2, it is depicted that the non-leaf prefix \( p \) is expanded when it is matched as the longest one in the FIB. For the address \( IP_1 \), \( p \) is the LMP in the FIB and expanded to \( p_1 \). As the expanded prefix \( p_1 \) is cached instead of \( p \), it avoids the incomplete caching problem because \( p_1 \) has no descendent prefixes. Now, the second packet arrives with \( IP_2 = 1001000 \), and a cache miss occurs. In this case, the LMP \( a \) for the packet is cached without expansion because the LMP \( a \) is a leaf prefix. If the second IP address were \( IP_4 = 1001111 \), a cache miss would still occur because the LMP \( p \) has not been cached and \( IP_4 \) does not match the currently cached prefix \( p_1 \). Note that, since the original LMP \( p \) cannot be cached in itself in the prefix expansion-based schemes, the address-spatial locality is not fully exploited. For the IP address \( IP_4 \), the expanded prefix \( p_2 \) should be cached even though the LMP of \( IP_4 \) is the same as that of \( IP_1 \).

Although prefix expansion resolves the prefix caching problem, it restricts the cache performance compared to the scheme which caches the original prefix without expansion. Even though multiple IP addresses match the same non-leaf prefix, each of their expanded prefixes should be cached. Thus, it results in more capacity misses in the same size of cache. In addition, the expanded prefix is more specific, so it exploits less address-spatial locality than the original prefix. For example, consider the IP addresses, \( IP_1 \) and \( IP_4 \). If the original LMP \( p \) was cached on \( IP_1 \) lookup, it could be reused on \( IP_4 \) lookup. However, the more specific prefix \( p_1 \) is cached instead and that prefix cannot be used for \( IP_4 \). Furthermore, the static prefix expansion has an additional disadvantage of increasing the size of the FIB besides the above shortcomings.
was presented to dynamically expand non-leaf prefixes using a bitmap on a caching process with low RRC to expand non-leaf prefixes on caching without modifying the original FIB [16]. They presented the multi-zone pipelined cache (MPC) was also developed to extend the SPE scheme to the multi-zone prefix when it needs to cache the prefix. Thus, the size of the FIB is not increased. In RRC-ME, the matching prefix length for the performance improvement. The short prefix expansion (SPE) was the second IP address were this case, the LMP is expanded to be cached. However, it requires additional processing time after the expansion by CPTE. In CPTE, the degree of each node is either 0 or 2. Both PPTE and CPTE prefixes do not overlap with other descendant prefixes. This method is called prefix expansion (LMP). An LMP is expanded when it is matched as the longest matching prefix. However, it requires additional processing time for expansion in proportion to the expansion length. A bitmap-based prefix cache (BMcache) [17] was presented to dynamically expand non-leaf prefixes using a bitmap on a caching process with low overhead in determining the expansion length.

Multi-zone caching can split a cache into multiple sections often depending on the length of the matching prefix length for the performance improvement. The short prefix expansion (SPE) was introduced to expand only prefixes shorter than 17, which decreases overhead for prefix expansion [18]. The multi-zone pipelined cache (MPC) was also developed to extend the SPE scheme to the multi-zone caching technique. In MPC, while IP address caching is used for long prefixes, prefix caching is used for short prefixes.

Zhang et al. developed a technique to find and cache the most popular prefixes based on traffic prediction [19]. When caching the prefix, there is an additional burden of caching all its descendant prefixes to ensure the correct result. Liu et al. introduced an FIB caching method to cache the popular prefixes into a limited size of ternary content addressable memory (TCAM) [20]. To resolve the incomplete caching problem, it caches a leaf prefix or a non-leaf prefix in the form of prefix expansion. The combined FIB caching and aggregation (CFCA) [21] scheme was presented to use TCAM-based cache to store frequently used prefixes. After expanding the non-leaf prefixes, it aggregates FIB prefixes according to the next-hop values. Rottenstreich and Tapolcai presented a cached classification method that applies lossy compression to the packet classifier [22]. Many packets are processed according to a
small number of popular prefix rules in a fast lossy classifier, and unclassified packets are processed in a conventional exact classifier.

Recently, some route prefix caching schemes [23,24] have been studied for fast name lookup in named data networking (NDN), which is one of the representative next-generation Internet. These techniques also use the prefix expansion technique to effectively cache non-leaf prefixes without any problem.

3. Bloom Filter-Based Prefix Caching

The prefix expansion limits the lookup performance due to the increase in the number of entries caused by expansion as well as the limited exploitation of locality from the expanded more specific prefixes. If it is possible to know whether the current matching prefix in a cache has a matchable child prefix in an FIB, the prefix cache can be utilized more effectively by eliminating the prefix expansion. Our scheme employs a Bloom filter to check the existence of child prefixes for each prefix.

A Bloom filter is widely used to check membership and is represented by an m-bit array for a set S of n elements. All bits in the filter are initialized to 0. After some element x is inserted into the filter, some bits \( h_i(x) \), \( 1 \leq i \leq k \) and \( x \in S \), are set to 1 where \( h_1, \ldots, h_k \) are k independent hash functions. Now, the membership of y is tested by checking whether \( h_i(y) \), \( 1 \leq i \leq k \), are all 1s. If so, y is probably an element of S (positive), otherwise y is clearly not an element of S (negative). Unfortunately, since hash functions can cause collisions, the filter can induce false positives with some probability. A false positive means that y is asserted to be an element of the target set, but it is not actually an element. The probability of the false positive must be lowered to increase the performance of the system employing the Bloom filter.

In our prefix caching, when an LMP is found in the FIB, the LMP is cached together with the corresponding Bloom filter which represents the existence of its child prefixes. Each element in a Bloom filter is identified by a bit string which represents the path from a prefix to its child.

The prefix caching scheme using a single Bloom filter is illustrated in Figure 3. Because the LMP for the first IP address \( IP_1 = 1000000 \) is \( p = 1^* \), the prefix \( p \) is cached with the corresponding Bloom filter BF as shown in the figure. In this example, the higher 5-bit string of the path from the prefix \( p \) to its child is used to represent a member for the BF. The child prefix length may differ according to children. We choose the higher bits to represent each child member, since those can be extracted without knowing the length. If the distance from a prefix to its child prefix is less than 5, then the child prefix is expanded to distance 5. For example, the distance from the prefix \( p = 1^* \) to its child prefix \( a = 10010^* \) is 4, the bit strings 00100 and 00101, which are the paths from prefix \( p \) to \( a_0 \) and \( a_1 \), respectively, are used as members for the BF. Note that, in our scheme, only the child prefix is expanded in its length to be a member and the original prefix \( p \) is not actually expanded, unlike other prefix expansion schemes. Because the bit string from the prefix \( p = 1^* \) to the prefix \( b = 100110^* \) is 5-bit 00110, the whole set becomes \( \{00100, 00101, 00110\} \), which corresponds to the prefixes \( a_0, a_1, \) and \( b, \) respectively. In addition, the Bloom filter in this example uses two hash functions: \( h_1(c_0c_1c_2c_3c_4) = c_0c_1c_2 \) and \( h_2(c_0c_1c_2c_3c_4) = c_2c_3c_4 \), where \( c_i \) denotes the ith bit of an element. To construct the BF, \( h_1(a_0), h_2(a_0), h_1(a_1), h_2(a_1), h_1(b), \) and \( h_2(b) \) are calculated. For example, \( h_1(a_0) = h_1(00100) = 001 \) and \( h_2(a_0) = h_2(00100) = 100 \). As a result, the 1st, 4th, 5th, and 6th bits of the BF are set to 1, while the other bits remain 0s.
In Figure 3, the Bloom filter BF for the prefix \( p \) is consulted when \( p \) is matched in the cache. Suppose that the destination IP address of the second packet is \( IP_2 = 1001000 \). It matches \( p = 1^* \) in the cache, and our scheme ignores the matching result of the cache and regards it as a cache miss through the BF test result. For the Bloom filter test of \( IP_2 \), the suffix after the prefix \( p \) is used as the input of the hash functions. Two hash values for \( IP_2 \), \( h_1(001000) = 001 \) and \( h_2(001000) = 100 \) are obtained, as shown in the figure. Since both BF[1] and BF[4] are 1s, it is determined that there is a child prefix in the FIB which \( IP_2 \) can match. Now, assume that the destination IP address of the second packet is \( IP_4 = 1001111 \). In this case, the same matching result \( p = 1^* \) is determined as the correct hit result. We obtain two hash values, \( h_1(001111) = 001 \) and \( h_2(001111) = 111 \). Since BF[1] = 1 and BF[7] = 0, there is certainly no child prefix of \( p \) which can be matched with the IP address \( IP_4 \).

4. A Proposed Caching Scheme: Two-Level Bloom Filter-Based Prefix Caching (BFPC)

4.1. A Two-Level Bloom Filter-Based Prefix Cache

For the matching prefix in a cache, the existence of their child prefix which matches a given incoming IP address can be tested with a Bloom filter. However, the Bloom Filter can misleadingly assert a positive, which means the positive is considered as the existence of a child even though there is no matching child. In turn, it raises a cache miss though the matching prefix is the correct result. The probability of false positive is introduced in [25]. For the filter size \( m \), the number of hash functions \( k \), and the number of elements \( n \) in the filter, it can be approximated by

\[
\left( 1 - \left( 1 - \frac{1}{m} \right)^{kn} \right)^k
\]

(1)

The probability increases as \( n \) grows. If the number of children for a matching prefix is large, the probability that the Bloom filter falsely asserts a positive is also high. For example, when we have a 16-bit filter and two hash functions, the false positive probability would be about 0.05 for \( n = 2 \), but 0.53 for \( n = 10 \).

For the best membership testing, the number of elements to be contained in a filter should be controlled. However, for a given matching prefix, the number of children cannot be changed. We propose a two-level Bloom filter that controls the number of elements to be contained. In the scheme, the first level is a false-positive free filter, and the second level is a filter in which the number of elements is restricted according to the filter size.

There is a set of hash functions that make a Bloom filter false-positive free if the number of elements \( n \) is bounded by some value [26,27]. In addition, if the filter size \( m \) is equal to the size of the universe set which is the set of all possible elements, the Bloom filter can be false-positive free by
one-to-one correspondence. For example, there are at most eight children at a distance 3, so the 8-bit filter can be false-positive free up to distance 3.

In our two-level Bloom filter, the first-level filter, BF1, provides a membership test up to distance $d_1$, and the second-level filter, BF2, up to distance $d_2$. BF1 has a size of $2^{d_1}$ to be false-positive free. However, the value of $d_1$ is restrictive because the filter size grows by the powers of two. In our scheme, the upper bound of $d_1$ is 4, i.e., the size of BF1 can be up to 16 bits. The value of $d_2$ is fixed to 5, but the size of BF2 is not 32 bits, therefore BF2 is not false-positive free. The size of BF2 is determined according to the number of elements to be contained so that the false-positive ratio is low. The total filter size is fixed, so the size of BF1 is obtained by subtracting the size of BF2 from the total size. Note that the scheme supports a membership test up to the distance 5, but we just use a total filter size of 16 bits.

The BF1 provides a membership test without a false positive up to a distance $d_1$, whereas the BF2 can do it for the limited number of elements to the greater distance. Figure 4 illustrates an example of a two-level Bloom filter for a prefix $p$. In this example, the sizes of BF1 and BF2 are 8 bits, i.e., $d_1 = 3$. In Figure 4a, the prefix $p$ has four children, $\{a, b, c, d\}$. The children $\{c, d\}$ are contained in BF2 because their distance is longer than $d_1$. The size of BF2 is determined as 8 bits because the number of elements to be contained is less than a constraint (three in this example). The children $\{a, b\}$ are contained in BF1 because their distances from $p$ are not longer than $d_1$. In the first level, there are two children $\{a, b\}$, but $a$ is expanded to the distance $d_1$. Thus, five elements $\{b, a_0, a_1, a_2, a_3\}$ are contained in BF1. For the BF2, a simple hash function is used, which outputs three consecutive bits after $p$. As a result, BF1[3:7] are set to 1. In the second level, two hash functions are used, which outputs the first and the last three bits after $p$, respectively. BF2[1], BF2[5], and BF2[6] are set to 1 as the result of the hash functions. Supposing that the size of an IP address is 7 bits, for an incoming IP address $k = 1001110$, $k$ matches $p$ if $p = 1^*$ is in the cache. Since $h_0(k) = 1$ and BF1[1] is 0, BF1 asserts a negative for $k$. If BF1 asserted a positive, it would raise a cache miss. In case that the BF1 asserts a negative, BF2 should be consulted. Since $h_1(k) = 1$ and $h_2(k) = 7$, both BF2[1] and BF2[7] should be checked. BF2[7] is not 1, so the result is a negative and it raises a cache hit with the matching prefix $p$.

![Figure 4](image-url)

Figure 4. An example of a two-level Bloom filter for a prefix $p$: (a) a trie; and (b) a two-level Bloom filter.

Figure 5 shows the structure of the two-level Bloom filter-based prefix cache (BFPC). Each prefix is stored in an associated memory, e.g., TCAM, but other information such as $d_1$, a two-level Bloom filter, and a port number is stored in a conventional SRAM. When an IP address matches some prefix, the corresponding Bloom filter is extracted and consulted. If the Bloom filter asserts a negative, it raises a cache hit. Although the prefix matching in the TCAM takes more time than SRAM access, searching is enabled in parallel. The search time depends on the number of entries in the TCAM. In a prefix cache, the number of entries is small enough compared to FIB, so the search time is not very much.

The BFPC can also be applied to IPv6 with 128-bit addresses. For IPv6, the TCAM should be extended in width to accommodate 64-bit prefixes, but three fields in SRAM basically remain the same.
length. The two-level Bloom filter size may change if needed, which depends on the prefix distribution in IPv6 forwarding table.

Figure 5. The structure of the two-level Bloom filter-based prefix cache (BFPC).

Figure 6 shows two algorithms for our two-level Bloom filter-based prefix cache: cache lookup and how to handle a cache miss. During the lookup process, the BFPC tries to find the longest matching prefix for a given IP address \( ip \). There are two types of cache misses: TYPE-1 and TYPE-2. If there is no matching prefix, then it raises a TYPE-1 cache miss. Otherwise, it consults a two-level Bloom filter for a matching prefix \( p \) with \( ip \). If it asserts a positive, i.e., there is probably some more specific prefix in the FIB, then it raises a TYPE-2 cache miss. Note that, despite the TYPE-2 cache miss, the matching prefix \( p \) could be the LMP because it was a false positive. If the Bloom filter asserts a negative, it is certainly a cache hit and the matching prefix \( p \) is the LMP.

Figure 6. The algorithms for BFPC: (a) cache lookup; and (b) processing a cache miss.
If a cache miss occurs for a given IP address \( ip \), the LMP should be searched for in the FIB. The LMP and its associated Bloom filter would be inserted into the prefix cache. Before the LMP, say \( p \), is inserted into the prefix cache, the Bloom filter for \( p \) should be consulted. If the filter asserts a positive for \( ip \), it implies that a cache hit never occurs for \( ip \) even though \( p \) has been inserted into the cache. In that case, \( p \) should be expanded to a 32-bit address, i.e., \( ip \) itself should be cached.

### 4.2. Construction of a Two-Level Bloom Filter

Although the total size of a two-level Bloom filter is fixed at 16 bits, each filter size can be adjusted. Each filter size is determined as the following rule. First, BF\(_2\) can exist only if there exists some child at a distance greater than or equal to 5. Second, the number of elements contained in the BF\(_2\) is restricted to a constraint. The constraint depends on the size of the BF\(_2\). Third, the size of the BF\(_2\) can be increased as long as the corresponding constraint is satisfied. Fourth, if the BF\(_2\) cannot satisfy any constraint, then only the BF\(_1\) is used.

Figure 7 depicts the algorithm to determine the size of each filter. In the algorithm, several notations are used as follows:

- \( n(d) \): the number of children at a distance \( \leq d \).
- \( \delta(d, x) \): the distance at which \( BF1 \) covers all the expanded children except \( BF2 \) covers.
- \( m1 \): the initial value of \( BF1 \) size (in the number of bits).
- \( m2 \): the initial value of \( BF2 \) size (in the number of bits).
- \( m \): the total filter size (= \( m1 + m2 = 16 \)).
- \( d1 \): the distance at which \( BF1 \) covers all the expanded children.
- \( d2 \): the distance at which \( BF2 \) covers all the expanded children except \( BF1 \) covers.
- \( c \): the constraint on a given \( x \).

Given \( I1, I2, m \) and \( c \),

determine \( d1, d2, m1, \) and \( m2 \)

\[
\begin{array}{c}
d2 = I2
\
\delta(d2, 1) \geq 1
\
\text{Let } x \text{ be the largest s.t. } \delta(I1 - 1, x) = 0
\
\begin{array}{c}
\delta(d2 - 1, 1) + \delta(d1, 1) \leq c
\
d1 = I1
\
m1 = 2^{d1}
\
m2 = 0
\
d1 = I1 - 1 - x
\
m1 = 2^{d1} \text{ (0 if } d1 \text{ is 0)}
\
m2 = m - m1
\end{array}
\end{array}
\]

\[
\begin{array}{cccc}
\delta & c & d1 & m2 \\
0 & 2 & 3 & 8 \\
1 & 3 & 2 & 12 \\
2 & 4 & 1 & 14 \\
3 & 5 & 0 & 16 \\
\text{--} & 4 & 0 & \\
d1 = \log_2 m1 \\
m2 = 16 - m1
\end{array}
\]

**Figure 7.** The algorithm for determining the size of each Bloom filter.

In Figure 7, \( x \) represents the gap distance between the BF\(_1\) and the BF\(_2\). \( c \) represents the constraint for a given \( x \). The BF\(_1\) size \( m1 \) and the BF\(_2\) size \( m2 \) are determined according to whether the constraint \( c \) is satisfied. The maximal set size for the BF\(_2\), i.e., the number of elements in the BF\(_2\) should be constrained by \( c \). The BF\(_2\) is used only if there is at least one element at a distance \( d2 \) and the number
of contained elements is less than or equal to \( c_x \). The BF\( _2 \) size depends on the satisfied \( c_x \), as shown in the table of Figure 7. For example, if \( x = 1 \) and the number of elements contained in the BF\( _2 \) is less than or equal to \( c_1 (= 3) \), then \( d_1 = 2 \) and the BF\( _2 \) size \( m_2 \) is 12 (\( = 16 - 2^d \)). Note that if \( x = 1 \) and the constraint was not satisfied (\( > c_1 \)), then BF\( _1 \) would be only used without BF\( _2 \). The constraint table is designed so that the BF\( _2 \) size \( m_2 \) increases as \( c_x \) grows.

Figure 8 shows examples of various filter sizes, \( m_1 \) and \( m_2 \), which are determined according to the positions of children and the satisfied constraint \( c_x \). Figure 8a–d shows the cases of \( x = 0, 1, 2, \) or 3. Figure 8e,f shows the cases that the BF\( _2 \) is not used. In Figure 8a, since \( x = 0 \) and the BF\( _2 \) tries to contain two prefixes (prefixes \( d \) and \( e \)), the number of which is less than or equal to \( c_0 \), \( m_1 \) and \( m_2 \) are determined as 8 bits, respectively. On the other hand, the BF\( _2 \) can accommodate more elements (prefixes \( c, d \), and \( e \)) in Figure 8b because \( x = 1 \) and the number of elements in the BF\( _2 \) is less than or equal to \( c_1 \). In this case, \( m_2 \) is determined as 12 bits. Note that the number of elements which is contained in the BF\( _2 \) is the sum of twice the number of prefixes at a distance 4 and the number of prefixes at distance 5. In this case, all prefixes are at distance 5. In another example of Figure 8d, four prefixes are contained in the BF\( _2 \), but the total number of contained elements is 5 because the prefix \( d \) is expanded to two elements.

5. Evaluation

The performance of BFPC was evaluated using a real-world prefix table and two generated traces. The BGP table at 18 August 2020-04:00 was obtained from [28]. For experiments, the BGP table was converted to an FIB which has the same prefixes as the BGP table. Recently, the size of the BGP table rapidly grows by 7–9% per year [29] and has become 817,583 entries. In publicly available packet traces, IP addresses are sanitized for privacy reasons. Such IP addresses will not match the prefixes in the real-world FIB. Thus, we used randomly generated traces that match the real-world prefixes as well as reflect the locality of references.

The characteristics of the FIB are shown in Table 1. The FIBs for three comparing schemes are also shown in the table. In CPTE, all non-leaf prefixes are pushed down and expanded until they become
leaf prefixes. Thus, there are only leaf prefixes and the FIB size is much bigger than the original one (total 988,828 entries). In PPTE, the FIB size is slightly increased. In NPE and BFPC, the table size does not change. The average prefix length of CPTE is greater than any others while the prefix length of NPE is the greatest on average when the prefix is cached. NPE cannot store the non-leaf prefixes in a cache and those should be expanded to 32 bits when they are cached. BFPC can utilize a prefix in a cache without expansion, regardless of whether it is a leaf or not. Thus, the average prefix length of BFPC is the same as the original one and BFPC also has the advantage that its FIB size is the same as the original one. On the other hand, CPTE cannot utilize non-leaf prefixes in the cache even though its average prefix length on a cache is slightly greater than BFPC.

Table 1. The characteristics of FIBs used in prefix caching schemes.

| FIB  | Num of Prefixes | Avg. Prefix Length | Avg. Length When Cached |
|------|-----------------|--------------------|-------------------------|
|      | Non-Leaf | Leaf | Total | Non-Leaf | Leaf | All | Non-Leaf | Leaf | All |
| Original | 82,707 | 734,876 | 817,583 | 20.333 | 23.027 | 22.754 | - | - | - |
| CPTE | 0 | 988,828 | 988,828 | 0 | 22.817 | 22.817 | - | 22.817 | 22.817 |
| PPTE | 82,707 | 759,206 | 841,913 | 20.333 | 22.974 | 22.715 | 32.000 | 22.974 | 23.861 |
| NPE | 82,707 | 734,876 | 817,583 | 20.333 | 23.027 | 22.754 | 32.000 | 23.027 | 23.934 |
| BFPC | 82,707 | 734,876 | 817,583 | 20.333 | 23.027 | 22.754 | 20.333 | 23.027 | 22.754 |

In Figure 9, the cache miss ratios of the five schemes are compared using two traces. CPTE shows the lowest miss ratio, but its number of table entries is larger than in any other scheme, as shown in Table 1. In addition, the FIB update overhead can be quite high in CPTE. For instance, a non-leaf prefix can be expanded to a lot of prefixes in CPTE and those expanded prefixes should be removed when the original prefix is removed. BFPC shows the miss ratio close to that of CPTE, with the same number of table entries as the original one. The miss ratio of BFPC is merely 0.003 in trace-1 or 0.027 in trace-2, in a cache with 8192 entries. The miss ratio of PPTE is not very good, even though its table size is bigger than that of BFPC. IPCache scheme does not cache any prefix, instead just caches the incoming 32-bit IP addresses. The poor miss ratio of IPCache results from a lack of exploiting address-spatial locality and the occupancy of many entries by caching individual IP addresses.

Figure 9. The cache miss ratio in several prefix caching schemes: (a) in trace-1; and (b) in trace-2.

Unlike prefix expansion-based caching schemes, BFPC caches an original prefix without expansion except that its Bloom filter asserts a positive for a given IP address. The cache miss reasons of BFPC are classified into two categories: TYPE-1 cache miss and TYPE-2 cache miss. The TYPE-1 cache misses are conventional and result from the absence of a matching prefix in the cache. Even though there is a matching prefix in the cache, the cache miss can still arise if a matching child prefix probably exists but is not cached yet. In such cases, BF1 and BF2 play a role in testing if there exists some matching child
prefix in an FIB. If BF₁ or BF₂ asserts a positive, then a TYPE-2 cache miss arises. Figure 10 shows the distribution of cache miss reasons. The cache misses mainly arise due to TYPE-1, which accounts for 75.0–96.4% of cache misses in trace-1, and 80.4–97.3% in trace-2. The cache misses due to BF₁-positive are the next most common misses, but relatively small. It seems that the relative portion of TYPE-2 increases when the cache size increases. The reason is that the actual total cache misses decreases while the cache size increases, and especially “unmatched” misses (TYPE-1) are relatively more reduced.

![Figure 10](image-url)

**Figure 10.** The distribution of cache misses according to their causes. “Unmatched” means there is no matched prefix in a cache, while “BF₁-positive” (or “BF₂-positive”) means there is a matched prefix but it results in a cache miss due to BF₁ (or BF₂) positive: (a) in trace-1; and (b) in trace-2.

Figure 11 shows the ratio that a Bloom filter asserts a negative. The negative ratio of a Bloom filter can be measured by cache hit counts among prefix matching counts in a cache. Remember that the cache hit arises only when there is a matching prefix in a cache and its Bloom filter asserts a negative. For any cache size, the negative ratio is higher than 0.986, which means, in more than 98.6% cases, the Bloom filter assures the matching prefix is the hit result. In Figure 11, the negative ratio slightly increases as the cache size increases.

![Figure 11](image-url)

**Figure 11.** The negative ratio of a Bloom filter.

In BFPC, each Bloom filter size is variable, but the total Bloom filter size is fixed to 16 bits. The size of the first-level filter, \( m₁ \), can be 0, 2, 4, 8, or 16. Figure 12 shows the first-level filter size over prefix length. There are many more cases in \( m₁ = 16 \) bits in which BF₁ is only used. The next most common cases are in \( m₁ = 0 \) in which BF₂ is only used. Generally, the number of non-leaf prefixes having a specific BF₁ size grows as the prefix length increases.

![Figure 12](image-url)
Figure 12. The number of non-leaf prefixes with a BF₁ size $m₁$ in log scale.

Figure 13 shows the number of non-leaf prefixes over the number of elements which are contained in their level filters. Since the BF₁ size is at most 16 bits, the number of elements to be contained is also at most 16 in the figure. In the BF₁, there is no general trend, other than that the number of non-leaf prefixes has a peak at a power of two. On the other hand, the number of non-leaf prefixes sharply decreases as the number of elements contained in the BF₂ increases. The number of elements contained in the BF₂ is at most 5 since it is restricted to that value. Note that the effectiveness of the BF₂ depends on the number of contained elements, whereas that of the BF₁ is independent of the number of contained elements.

Figure 13. The number of non-leaf prefixes according to the number of elements contained in their filters in log scale.

6. Conclusions

IP address lookup is a very crucial function in forwarding packets, and prefix caching is one of the techniques to enhance the lookup performance effectively. In general, a prefix cache can achieve better performance than an IP cache. The prefix cache exploits address-spatial locality as well as temporal locality, whereas the IP cache can exploit only temporal locality. However, if a non-leaf prefix is allowed to be cached, it incurs an incorrect result. The reason for such an incorrect result is that another longer matching prefix may exist in an FIB when a non-leaf prefix in the cache is matched. To resolve the problem, all the descendants should be cached together with the non-leaf prefix, or it should be assured that all the cacheable prefixes are disjoint to each other. The prefix expansion is one of the techniques which can obtain a set of cacheable disjoint prefixes. However, the prefix expansion technique increases the FIB size as well as makes the prefix to be expanded. The expanded prefix represents a more specific prefix and covers a smaller range of addresses. Thus, the exploitable spatial locality is reduced in those schemes.
This paper proposes the Bloom filter-based prefix caching scheme (BFPC). While most of the existing prefix caching techniques are based on prefix expansion, BFPC improves the caching technique so that non-leaf prefixes can be cached without the prefix expansion. A Bloom filter for each prefix plays a role in a membership test whether there exists another longer matching prefix. A prefix is stored in a cache entry together with its Bloom filter. If some prefix is matched in the cache during cache lookup, then its Bloom filter is consulted. The positive assertion of the Bloom filter implies that there is possibly another longer matching prefix, i.e., a cache miss occurs.

Generally, the Bloom filter size is relatively small compared to the number of all possible containable elements. However, the probability of false positive increases as the number of contained elements grows. Our design goal is to reduce the false positive by using a two-level Bloom filter. The two-level Bloom filter scheme adjusts the size of each level filter to reduce the false positives. The filter size is adjusted according to the number of elements to be contained in the second-level filter. The second-level filter can accommodate a limited number of child prefixes at a long distance to reduce the false positives. On the other hand, the first-level filter can accommodate a maximum number of child prefixes at a short distance.

The experimental result shows that BFPC achieves a very low cache miss ratio without increasing the number of FIB entries. The cache miss reasons are also observed in the experiment. Most cache misses are due to no matching prefix, and there are not many misses due to the positive assertion of the Bloom filters. In addition, the membership tests of the Bloom filters are observed to be negative assertions in most cases. It implies that, when the original prefixes are cached without expansion in BFPC, they can be effectively hit using the descendant test of BFPC.

Author Contributions: Conceptualization, J.K. (Junghwan Kim); Formal analysis, J.K. (Junghwan Kim) and J.K. (Jinsoo Kim); Investigation, M.-C.K. and M.S.S.; Methodology, J.K. (Jinsoo Kim); Validation, M.-C.K. and M.S.S.; Writing—original draft, J.K. (Junghwan Kim); and Writing—review and editing, J.K. (Jinsoo Kim) and J.K. (Junghwan Kim). 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 conflict of interest.

References
1. Ruiz-Sanchez, M.; Biersack, E.; Dabbous, W. Survey and taxonomy of IP address lookup algorithms. IEEE Netw. 2001, 15, 8–23. [CrossRef]
2. Ravikumar, V.C.; Mahapatra, R.N. TCAM architecture for IP lookup using prefix properties. IEEE Micro 2004, 24, 60–69. [CrossRef]
3. Zheng, K.; Hu, C.; Lu, H.; Liu, B. A TCAM-based distributed parallel IP lookup scheme and performance analysis. IEEE/ACM Trans. Netw. 2006, 14, 863–875. [CrossRef]
4. Huang, J.-Y.; Wang, P.-C. TCAM-Based IP Address Lookup Using Longest Suffix Split. IEEE/ACM Trans. Netw. 2018, 26, 976–989. [CrossRef]
5. Nilsson, S.; Karlsson, G. IP-address lookup using LC-tries. IEEE J. Sel. Areas Commun. 1999, 17, 1083–1092. [CrossRef]
6. Wu, Y.; Nong, G.; Hamdi, M. Scalable pipelined IP lookup with prefix tries. Comput. Netw. 2017, 120, 1–11. [CrossRef]
7. Kim, J.; Ko, M.-C.; Shin, M.S.; Kim, J. IMT: A Memory-Efficient and Fast Updatable IP Lookup Architecture Using an Indexed Multibit Trie. KSII Trans. Internet Inf. Syst. 2019, 13, 1922–1940. [CrossRef]
8. Lim, H.; Seo, J.-H.; Jung, Y.-J. High speed IP address lookup architecture using hashing. IEEE Commun. Lett. 2003, 7, 502–504. [CrossRef]
9. Dharmapurikar, S.; Krishnamurthy, P.; Taylor, D.E. Longest prefix matching using bloom filters. IEEE/ACM Trans. Netw. 2006, 14, 397–409. [CrossRef]
10. Song, H.; Hao, F.; Kodialam, M.; Lakshman, T.V. IPv6 Lookups using Distributed and Load Balanced Bloom Filters for 100Gbps Core Router Line Cards. In Proceedings of the IEEE INFOCOM 2009—The 28th Conference on Computer Communications, Rio de Janeiro, Brazil, 19–25 April 2009; pp. 2518–2526.
11. Byun, H.; Li, Q.; Lim, H. Vectored-Bloom Filter for IP Address Lookup: Algorithm and Hardware Architectures. *Appl. Sci.* 2019, 9, 4621. [CrossRef]
12. Chiueh, T.C.; Pradhan, P. High-Performance IP Routing Table Lookup using CPU Caching. In *Proceedings of the IEEE INFOCOM ‘99—Conference on Computer Communications*, New York, NY, USA, 21–25 March 1999; pp. 1421–1428.
13. 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.
14. Kim, J.; Park, M.; Han, S.; Kim, J. An Efficient Prefix Caching Scheme with Bounded Prefix Expansion for High-Speed IP Lookup. *IEICE Trans. Commun.* 2012, 95, 3298–3301. [CrossRef]
15. Bloom, B.H. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM* 1970, 13, 422–426. [CrossRef]
16. 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 August 2004; pp. 41–46.
17. Bitmap-based Prefix Caching for Fast IP Lookup. *KSII Trans. Internet Inf. Syst.* 2014, 8, 873–889. [CrossRef]
18. Kasnawi, S.; Berube, P.; Gaudet, V.; Amaral, J.N. A cache-based internet protocol address lookup architecture. *Comput. Netw.* 2008, 52, 303–326. [CrossRef]
19. Zhang, W.; Bi, J.; Wu, J.; Zhang, B. Catching popular prefixes at AS border routers with a prediction based method. *Comput. Netw.* 2012, 56, 1486–1502. [CrossRef]
20. Liu, Y.; Lehman, V.; Wang, L. Efficient FIB caching using minimal non-overlapping prefixes. *Comput. Netw.* 2015, 83, 85–99. [CrossRef]
21. Grigoryan, G.; Liu, Y.; Kwon, M. Boosting FIB Caching Performance with Aggregation. In *Proceedings of the 29th International Symposium on High Performance Interconnects*, Stockholm, Sweden, 23–26 June 2020; pp. 221–232.
22. Rottenstreich, O.; Tapolcai, J. Optimal Rule Caching and Lossy Compression for Longest Prefix Matching. *IEEE/ACM Trans. Netw.* 2016, 25, 864–878. [CrossRef]
23. Chen, X.; Zhang, G.; Cui, H. Investigating Route Cache in Named Data Networking. *IEEE Commun. Lett.* 2017, 22, 296–299. [CrossRef]
24. Kim, J.; Kim, J. An Efficient Prefix Caching Scheme for Fast Forwarding in Named Data Networking. *Stud. Inform. Control* 2019, 27, 175–182. [CrossRef]
25. Broder, A.; Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Math.* 2004, 1, 485–509. [CrossRef]
26. Kiss, S.Z.; Hosszu, E.; Tapolcai, J.; Ronyai, L.; Rottenstreich, O. Bloom Filter with a False Positive Free Zone. In *Proceedings of the IEEE INFOCOM 2018—IEEE Conference on Computer Communications*, Honolulu, HI, USA, 16–19 April 2018; pp. 1412–1420.
27. Rottenstreich, O.; Reviriego, P.; Porat, E.; Muthukrishnan, S. Constructions and Applications for Accurate Counting of the Bloom Filter False Positive Free Zone. In *Proceedings of the Symposium on SDN Research*, San Jose, CA, USA, 3 March 2020; pp. 135–145.
28. BGP Routing Table. Available online: [http://thyme.apnic.net/ap-data/2020/08/0400/](http://thyme.apnic.net/ap-data/2020/08/0400/) (accessed on 19 August 2020).
29. BGP in 2019—The BGP Table. Available online: [https://blog.apnic.net/2020/01/14/bgp-in-2019-the-bgp-table/](https://blog.apnic.net/2020/01/14/bgp-in-2019-the-bgp-table/) (accessed on 15 January 2020).

**Publisher’s Note:** MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

© 2020 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).