Article

A New Bloom Filter Architecture for FIB Lookup in Named Data Networking

Hayoung Byun and Hyesook Lim *

Department of Electronic and Electrical Engineering, Ewha Womans University, Seoul 03760, Korea; hayoung77@ewhain.net

* Correspondence: hlim@ewha.ac.kr; Tel.: +82-2-3277-3403

Received: 27 November 2018; Accepted: 14 January 2019; Published: 17 January 2019

Abstract: Network traffic has increased rapidly in recent years, mainly associated with the massive growth of various applications on mobile devices. Named data networking (NDN) technology has been proposed as a future Internet architecture for effectively handling this ever-increasing network traffic. In order to realize the NDN, high-speed lookup algorithms for a forwarding information base (FIB) are crucial. This paper proposes a level-priority trie (LPT) and a 2-phase Bloom filter architecture implementing the LPT. The proposed Bloom filters are sufficiently small to be implemented with on-chip memories (less than 3 MB) for FIB tables with up to 100,000 name prefixes. Hence, the proposed structure enables high-speed FIB lookup. The performance evaluation result shows that FIB lookups for more than 99.99% of inputs are achieved without needing to access the database stored in an off-chip memory.

Keywords: Bloom filter; named data networking; functional Bloom filter; priority trie

1. Introduction

Currently, on the Internet, massive multimedia contents are being widely transferred, and the use of mobile applications is becoming increasingly popular. The amount of Internet traffic requiring high speed data transfer is therefore increasing exponentially. Since the Internet is based on a host-based communication infrastructure and forwards packets using IP addresses, transmission bottlenecks can easily occur if multiple users repeatedly request the same contents to a single host. In order to solve this problem, Named Data Networking (NDN) technology has been proposed as a promising future Internet architecture [1–3]. The NDN is an Internet architecture used to distribute content requests to different nodes in a network that had previously been concentrated on content sources. The NDN architecture effectively reduces network traffic by storing contents in network nodes such as routers, and repetitively providing the contents requested by different users.

Content producers and content consumers in the NDN infrastructure communicate using both *Interest* packets and *Data* packets. When a consumer broadcasts an Interest packet for some content, the Data packet generated by the content producer can be transmitted by any router receiving the Interest and holding the Data. In other words, if an NDN router has the corresponding Data packet on the path between the producer and the consumer, the NDN router transmits the content to the consumer. In this way, the Data packet is transmitted from the network node closest to the consumer among the nodes with the content.

An NDN router has three tables for packet forwarding: Content Store (CS), Pending Interest Table (PIT), and Forwarding Information Base (FIB). The CS is a cache memory for the temporary storage of Data packets [4]. The PIT is used for the transmission of the arrived Data packet by recording each forwarded Interest as well as the corresponding incoming face (port) of the Interest [5]. The FIB stores multiple output faces mapped to a name prefix in order to forward each Interest packet to its

producer [6]. Among the many issues that need to be addressed in realizing the NDN infrastructure, wire-speed FIB lookup is one of the most essential [7–9]. FIB lookup algorithms can be categorized based on their approach: trie-based algorithms, such as name prefix trie (NPT) and priority-NPT (p-NPT) [10,11]; hashing-based algorithms such as hashing-based NPT [9]; and Bloom filter-based algorithms such as NPT-BF, NPT-BF-chaining, and I(FIB)F [9,12].

The motivation of this paper is to propose a high-speed FIB lookup algorithm based on a name prefix trie (NPT). Generally, a name prefix trie (NPT) is implemented with an off-chip memory because of its size [10]. Starting from the root node, the search procedure in an NPT finds the longest matching prefix, called the best matching prefix (BMP). The off-chip NPT search should then continue until a leaf node is encountered, even if matching prefixes have already been found in the search path. Hence, the NPT does not provide very good search performance. In this paper, we attempt to improve the search performance in an NPT by effectively utilizing on-chip memories because the time to access an off-chip memory is 10–12 times slower than that to access an on-chip memory [13]. Our primary goal is to use a space-efficient data structure that can be implemented with on-chip memories so that the search procedure required in the FIB lookup is completed only through on-chip memory accesses. Our secondary goal is to reduce the amount of on-chip memory access by preferentially searching the prefix with the longest level.

In this paper, we propose a level-priority trie (LPT) algorithm for the FIB table lookup and a 2-phase Bloom filter architecture implementing the LPT. Bloom filters have been applied in many network algorithms to improve the search performances of routing tables because of their simplicity [14–16]. The proposed 2-phase Bloom filter architecture occupies small on-chip memories and completes FIB lookups only through accesses to these memories for more than 99.99% of inputs.

The remainder of this paper is organized as follows. Section 2 describes related works. Section 3 proposes a level-priority trie (LPT) and a 2-phase Bloom filter architecture implementing the LPT. Section 4 presents a theoretical analysis for the proposed architecture. Section 5 evaluates the performance of the proposed architecture; then, Section 6 concludes the paper.

2. Related Works

2.1. Name Prefix Trie

A name prefix consists of several name components separated by dot(.)s and slash(/)es. For example, *facebook.com/user* has three components: *facebook*, *com*, and *user*. As a simple structure for name lookup, a name prefix trie (NPT) stores a name prefix into a node corresponding to the path from the root node [10]. When storing a name prefix in a node of the NPT, name components separated by dots are enumerated in the reverse order, while those separated by slashes are enumerated in the forward order. Figure 1 shows an NPT constructed using 11 arbitrary name prefixes, shown together with the NPT. Each black node of the NPT stores a name prefix and the corresponding output face. The NPT structure has several issues that need to be considered before it can be applied for FIB lookup in NDN: First, the NPT is highly unbalanced because the number of child nodes is not constant and the lengths of the name prefixes are not fixed. Second, due to the presence of many empty nodes, the memory implementing the NPT is wasted, and the lookup performance is degraded. Third, in order to find the longest matching prefix, each name lookup needs to proceed until there is no further node to follow.

Figure 1. Name prefix trie.

2.2. Priority Trie

The priority trie [17] for IP address lookup [18] removes any empty internal nodes in a trie. The priority trie is constructed by repeatedly relocating the longest prefix included in the sub-trie of an empty node into that empty node. A node storing a prefix longer than its own level is called a priority node. The priority trie optimizes the memory requirement by removing empty nodes, and also improves the lookup performance by immediately terminating the search upon finding a matching priority node.

Since the NPT is a highly unbalanced trie with many empty nodes, the performance of the NPT can be improved by applying the priority trie [11]. The priority-NPT (p-NPT) can be similarly constructed by repeatedly relocating the longest and leftmost name prefix included in the sub-trie of an empty node into that empty node. In the search procedure of the p-NPT, if an input matches the name prefix stored in a priority node, the matching prefix is guaranteed to be the longest matching prefix. Hence, the search procedure can be immediately finished. As a result, the p-NPT requires less memory and provides a more effective search procedure than NPT.

2.3. Bloom Filter

A Bloom filter [19] is an efficient bit-vector-based data structure that represents the membership information of a set. A Bloom filter has an array of m bits, which are initialized with 0 s. Since a single cell of the Bloom filter is composed of a single bit, each cell can only indicate either 0 or 1. Bloom filters support two operations: programming and querying. In programming, for each element in a set, the k cells indicated by the hash indexes generated by k independent hash functions are set to 1.

By contrast, querying is used to determine whether or not a given input is included in the programming set. In querying an input, k hash indexes are also generated using the same hash functions that had been previously used to program the Bloom filter. If any of the k bits are 0, the input is definitely not a member of the set, and thus termed a *negative* result. If all of the k bits are 1, the input is identified as a member of the set and termed a *positive* result. Bloom filters never generate false negatives, but they may generate false positives resulting from hash collisions.

The false positive rate (f) is calculated as follows [20,21], where m is the size of a Bloom filter and n is the number of programmed elements:

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

The number of optimal hash functions minimizing the false positive rate can be found as follows [20,21]:

$$k = \frac{m}{n} \ln 2. \quad (2)$$

Recently, numerous variants of the Bloom filter have been proposed [22–25] and applied to various network applications [26–28]. A functional Bloom filter [24,25] allocates multiple bits in a single cell, and hence it can return the value corresponding to an input as well as the membership information of that input by checking the stored value. In programming, for each element in a set, the return value corresponding to the element is stored in the k cells indicated by k hash indexes. In the case that two or more different values need to be programmed to a single cell by different elements, the cell is denoted as a *conflict* value.

The purpose of querying to a functional Bloom filter is to both return the stored value and determine the membership of each given input. The querying result can be classified as either *negative*, *positive*, or *indeterminable* [24]. In querying an input, if any of the k cells is 0, or if some of their values differ from one another, the input is definitely not a member of the programming set, and therefore termed a *negative* result. If all of the cells, excluding conflict cells, have the same value, the input is identified as a member of the set, and the Bloom filter returns the stored value, termed a *positive* result. If all of the k cells have *conflicts*, the membership and return value of the input cannot be identified by the functional Bloom filter, and therefore termed an *indeterminable* result.

2.4. FIB Lookup Algorithms Applying Bloom Filters

The NPT-BF algorithm [9] has a Bloom filter applied in order to improve the search performance of the NPT. In this algorithm, the NPT is assumed to be stored in an off-chip memory because of its size while an on-chip Bloom filter is used to reduce the number of off-chip memory accesses. Every node of the NPT is programmed in a Bloom filter. Prior to accessing the NPT stored in an off-chip memory for an input, the Bloom filter is examined for each substring of the input in order to identify the existence of the substring. If the Bloom filter produces a positive result for a substring of some input, the off-chip NPT is accessed, and the search procedure remembers the output face stored with a matching name prefix (if one exists), then the search procedure continues on the next substring. If the Bloom filter produces a *negative* result, the search procedure finishes without accessing the off-chip NPT, since the Bloom filter can never produce false negatives and no matching prefix longer than the current longest matching name prefix is guaranteed in the NPT. Hence, the NPT-BF reduces the number of off-chip accesses by avoiding unnecessary off-chip access when the Bloom filter produces a *negative* result.

While the NPT-BF accesses the off-chip NPT for every positive result of the Bloom filter, the NPT-BF-chaining [9] accesses the off-chip NPT for the longest positive result. In other words, the NPT-BF-chaining continuously performs Bloom filter queries without needing to access the off-chip NPT until a negative result is produced. The off-chip NPT is accessed for the last positive result, which may be the longest name prefix matching the substring of the input. If a matching name prefix is found with an output face (assuming that the matching output face is pre-computed to each empty node), the search procedure finishes and returns the output face. If no matching name prefix exists because of a Bloom filter false positive, back-tracking will occur.

However, in these algorithms, every node, including empty nodes, should be programmed into the Bloom filter, and the search performance is directly affected by the NPT depth since the Bloom filter is sequentially accessed from the shortest level to the longest level in the same way as the search in the NPT. Moreover, the search procedure cannot be completed only through Bloom filter accesses because the Bloom filter does not provide a return value.

The I(FIB)F structure [12] uses iterated Bloom filters, and can complete the search procedure without accessing the off-chip memory since an I(FIB)F is constructed for each output face, as opposed

to a single FIB that defines the next-hop. An I(FIB)F consists of d iterated Bloom filters (IBF) [29] with iterated hash functions. In order to construct an I(FIB)F, a m -bit standard BF is split in d IBFs of m/d bits. IBFs benefit from the properties of iterative trees, such as NPTs. In the search procedure, when an Interest packet arrives through a face, the I(FIB)Fs of all faces except for the incoming face are checked, and then the face with the largest match is selected for forwarding. Since the I(FIB)F structure considers a flat naming scheme, while, in this paper, we consider a structured naming scheme using name prefixes, the I(FIB)F is not directly compared with our proposed algorithm.

3. Proposed Algorithm

The role of the on-chip Bloom filter in [9] is to reduce the number of memory accesses to the off-chip NPT by producing a negative result in cases in which the current substring of an input does not match any node. This paper aims to complete NPT-based FIB lookup only through on-chip Bloom filter queries.

We propose a level-priority trie (LPT) and 2-phase Bloom filter architecture implementing the LPT. The LPT removes all of the empty nodes (except the root node) in an NPT and stores level information at each node. The 2-phase Bloom filter architecture implementing the LPT with on-chip memories uses two functional Bloom filters: the first returns the level of the best matching prefix, while the second returns the output face for the best matching prefix.

3.1. Level-Priority Trie

The motivation of the proposed LPT begins with applying the p-NPT for name lookup. Since the number of components of a name prefix stored in a priority node is not constant, each priority node should provide the number of components of the name prefix, which in our terminology is the level information. While the p-NPT only removes the longest name prefix in the sub-trie of an empty node by relocating it to the priority node, the LPT removes all of the name prefixes in the longest level by storing the longest level information in the level priority node.

Figure 2 shows the level-priority trie (LPT) transformed from the NPT shown in Figure 1. Since an NPT is expected to have many empty nodes, the LPT is constructed by repeatedly removing empty nodes (except for the root node) and storing the level information of the longest prefix. Each empty node becomes a level-priority node (LP-node) by storing the longest level information of its own sub-trie as well as removing every black node with a prefix in the longest level. The gray-colored nodes in Figure 2 are LP-nodes. For example, as an empty node, node *com* in Figure 1 is transformed into an LP-node in the LPT as shown in Figure 2, and stores level 4, because the longest level of its sub-trie is 4. Each non-deleted black node stores its own level. Therefore, the LPT is a lightweight trie that only stores level information, and the total number of nodes of the LPT is much lower than that of the NPT or that of the p-NPT (because every black node in the longest level is deleted). Figure 2 also shows a hash table storing the output face of each name prefix. Since the nodes in the LPT only store level information, a hash table is required in order to obtain the output face of the best matching name prefix.

Figure 2 shows a brief overview of the search procedure. When the LPT is searched, if an ordinary node is encountered, the level is remembered, and the search procedure proceeds to the next level. When there are no further nodes to follow, the hash table is accessed for the last matching level. Otherwise, if an LP node is encountered, the hash table is immediately accessed, since the LPT returns the longest level. If the input matches the name prefix at the returned level in the hash table, the search for a given input can be completed. In the case that the hash entry does not match the input at the returned level, the search procedure returns to the next level of the LPT.

For an example of a search, for an input *com/youtube/user/image*, since the first component *com* reaches an LP-node with level information 4, the hash table is accessed using the string of *com/youtube/user/image* as the hash key. The hash table entry does not match the string, and hence, the search procedure returns to the LPT. At the next level, the first two components *com/youtube* reach an ordinary node with level information 2. If the node has a matching child, the search should

continue on to the next level. However, since the node does not have any children, the hash table is accessed using the string of *com/youtube*. The hash entry returns output face 2, and the search procedure is complete.

Figure 2. Level-priority trie and a hash table.

3.2. 2-Phase Bloom Filter Architecture

For the implementation of the LPT and the hash table shown in Figure 2, we propose using two functional Bloom filters: a level Bloom filter (*l*-BF) and a port Bloom filter (*p*-BF). In the proposed architecture, the level information stored in the LPT is programmed to *l*-BF, while the output face information stored in the hash table is programmed to *p*-BF.

3.2.1. Construction

Figure 3 shows the proposed 2-phase Bloom filter architecture. The *l*-BF is programmed for every node (except for the root node) of the LPT and stores the level for each node. For example, since node *com* shown in Figure 2 has level 4, *k* cells corresponding to the hash indexes generated by key *com* are set to value 4. The *p*-BF is then programmed for every name prefix in an FIB table and stores the output face information corresponding to each prefix. For example, *k* cells corresponding to the hash indexes generated by name prefix *com/youtube/user/skyDoesMinecraft*, shown in Figure 1, are set to value 3, since the output face of the name prefix is 3. In programming the *l*-BF or the *p*-BF, it is important to note that a cell is set to the maximum value (denoted by **X** in this paper) in order to represent the *conflict*, if two or more values are programmed to a cell. In querying the *l*-BF or the *p*-BF, a value will be returned if the accessed cells (except conflict cells) have the same value. If every returned cell value

of the p -BF is considered to be a *conflict* cell for a given input, the output face cannot be determined. The hash table shown in Figure 3, the details of which will be described in the next section, is provided for this case.

Figure 3. Proposed 2-phase Bloom filter architecture.

3.2.2. Search

Figure 3 also shows a brief overview of the search procedure of the proposed architecture. The search procedure proceeds by interactively querying the l -BF and the p -BF. By querying the l -BF, the level is returned. The returned level is used in querying the p -BF, and the output face is obtained. This process is repeated until a negative result is returned in l -BF.

For example, assume an input *com/youtube/user/image*. The l -BF is first queried using *com* as the hash key. If at least one non-conflict cell exists, the l -BF returns level value 4. Since the returned value is larger than the number of components used in querying, it can be noticed that node *com* is an LP-node. The p -BF is queried with the key *com/youtube/user/image*. In this case, the p -BF returns a negative result. Hence, the l -BF is queried again using *com/youtube* and returns 2. Since the level information returned by string *com/youtube* is the same as the number of components, node *com/youtube* is considered to be an ordinary node. Hence, the search procedure queries the l -BF again using *com/youtube/user*. The l -BF returns a negative result, meaning that neither node *com/youtube/user* nor its children exist. Hence, level 2 is the last level and the p -BF is queried using *com/youtube*. The search procedure is completed by returning output face 2. In this case, the total number of Bloom filter queries is 5, and no accesses to the off-chip hash table occur.

As another example, assume input name *com/facebook*. First, the l -BF is queried using *com* and returns level 4. However, since the number of input components is fewer than 4, the l -BF is queried again using *com/facebook*. The l -BF then returns level 2. Since an ordinary node is accessed, the l -BF query should be continued, but there are no more input components. Hence, the p -BF is queried using *com/facebook*. Subsequently, the p -BF returns the output face and the search is complete. In this case, the total number of Bloom filter accesses is 3, without any off-chip memory accesses.

Algorithm 1 describes the search procedure in detail, which consists of the *l*-BF query and the *p*-BF query. Algorithms 2 and 3 show the detailed querying procedure of the *l*-BF and the *p*-BF, respectively.

Algorithm 1: Search Procedure

```

Function Search(inName)
  BMPport = default;
  for (i = 1 to Plevel) do
    // Plevel is the depth of the LPT
    nodeType = queryLevelBF(inName, i, &match_lvl);
    if (nodeType == ORD) then
      | last_match_lvl = i;
    else if (nodeType == PRI) then
      | BMPport = queryPortBF(nodeType, inName, match_lvl, &indet_lvl);
      | if (BMPport != default) then
      | | break;
      | end
    else if (nodeType == NEG) then
      | BMPport = queryPortBF(nodeType, inName, last_match_lvl, &indet_lvl);
      | break;
    else if (nodeType == INDET) then
      | BMPport = queryPortBF(nodeType, inName, C, &indet_lvl); // back-tracking
      | break;
    else
      | continue;
    end
    if (i == Plevel or C) then
      // C is the number of components of the inName
      | BMPport = queryPortBF(NEG, inName, last_match_lvl, &indet_lvl);
      | break;
    end
  end
  if (BMPport == INDET) then
    | BMPport = searchHT(inName, indet_lvl);
    | // search hash table for an indeterminable case
  end
  return BMPport;
end

```

For a given input name, querying the *l*-BF produces one of four types of results: ORD, PRI, NEG, and INDET, as shown in Algorithm 2. The ORD result occurs when an ordinary node is accessed. In this case, the current level is remembered as the *last_match_lvl*, and the *l*-BF querying continues as shown in Algorithm 1. The PRI result occurs when an LP-node is accessed. In this case, the *p*-BF is queried using the returned level. If the *p*-BF returns the output face by producing a *positive* result, the search immediately ends. Otherwise, if the *p*-BF produces a *negative* result (meaning that the longest name prefix does not match the input), the search procedure returns to the *l*-BF query for the next level. For the NEG (negative) result in the *l*-BF querying, the *p*-BF is queried from the *last_match_lvl* while the level of access is decreased until a matching output face is found in the *p*-BF.

The INDET result occurs when every accessed cell of the *l*-BF has a *conflict* value. Even though the INDET is likely a *positive*, since the type (ORD and PRI) of the node causing the INDET is unknown, the search cannot proceed to the next level. Hence, for the INDET result, we perform *p*-BF querying beginning from the longest level of the input name until a matching output face is found in the *p*-BF.

Algorithm 2: Querying Procedure of the *l*-BF

```

Function queryLevelBF(inName, AccLvl, matchLvl)
  matchLvl = queryBF(levelBF, inName, AccLvl);
  if (matchLvl == AccLvl) then
    | return ORD;
  else if (AccLvl < matchLvl ≤ C) then
    | return PRI;
  else if ( $0 \leq \textit{matchLvl} < \textit{AccLvl}$ ) then
    | return NEG;
  else if (matchLvl == X) then
    | return INDET;
  else
    | return NOTHING;
  end
end

```

For the INDET result of the *p*-BF shown in Algorithm 3, a matching output face cannot be determined with only 2-phase Bloom filter queries. In order to address this issue, an off-chip hash table is also required. Backtracking should then occur from the indeterminable level in the hash table.

Algorithm 3: Querying Procedure of the *p*-BF

```

Function queryPortBF(nodeType, inName, firstAccLvl, indetLvl)
  if (nodeType == PRI) then
    | matchPort = queryBF(portBF, inName, firstAccLvl);
  else
    // back-tracking
    for (i = firstAccLvl to 1) do
      | matchPort = queryBF(portBF, inName, i);
      | if (matchPort == X) then
        | | indetLvl = i;
        | | return INDET;
      | else if (matchPort > 0) then
        | | break;
      | end
    // in case of a negative, continue
    end
  end
  return matchPort;
end

```

3.2.3. Update

The functional Bloom filters used in our proposed architecture can provide incremental insertions and deletions by ensuring that each cell is programmed by a single key. When programming the return value for a given key, if a cell indicated by the hash index generated for the given key already has a value (meaning that the cell has already been programmed by another element), the cell is set to *conflict X*. Since each cell with a value other than *X* is programmed by a single key, the functional Bloom filter can provide delete operations. In deleting an element from a functional Bloom filter, each cell among the *k* cells with a value other than *X* is changed to 0; conflict cells are not changed in this process.

Repeated insertions and deletions will increase the number of conflict cells, and, accordingly, degrade the performance of the functional Bloom filter. Hence, the proposed architecture should be

reconstructed if the number of conflict cells is larger than a pre-defined threshold value. Determining the threshold value for the desired performance in terms of false positive rate and *indeterminable* rate is beyond the scope of this paper.

3.2.4. Discussion

The LPT proposed in this paper provides benefits in terms of both the search performance and the memory requirement. If we assume a naive Bloom filter structure implementing an NPT, many empty nodes should be programmed and queried, starting from a root node. In order to avoid queries to empty nodes, if the search begins with the longest level, Bloom filter querying should continue until a matching name prefix is found while decreasing the level of access. Hence, if the NPT is more unbalanced, more memory accesses occur.

The proposed LPT takes advantage of both cases. The search procedure in an LPT initially begins with the root, and the level increases when ordinary nodes are encountered. However, when a level-priority node is encountered at a certain level, the input is compared with the longest prefix. Based on the comparison result, the search can either be completed or continue to the next level.

In addition, the LPT-based search can easily be implemented with the functional Bloom filters as described. A functional Bloom filter is a space-efficient data structure that only stores return values without the signature of each programmed element, since different combinations of cell indexes can work as the signature of each different key. Hence, the proposed structure can be implemented with on-chip memories, and the FIB lookup can be completed with a small number of on-chip accesses, as will be shown through simulation in a later section.

4. Bloom Filter Analysis

Since Bloom filters can produce false positives, and even though the false positive rate can be controlled by increasing the size of the Bloom filter, Bloom filter-based algorithms should carefully handle false positives. Functional Bloom filters produce indeterminable results, and hence functional Bloom filter-based algorithms should handle the indeterminable cases as well.

The *indeterminable* and *false positive* of the *l*-BF are resolved in our proposed algorithm by the *p*-BF. Hence, here we only consider the *indeterminable* and *false positive* of the *p*-BF. In our proposed algorithm, if the *p*-BF produces *indeterminable*, the search procedure should access the off-chip hash table to obtain the output face. If the *p*-BF produces false positives, the search procedure returns an incorrect output face. Therefore, we present a theoretical analysis of the probabilities of *indeterminable* and *false face return* for the *p*-BF.

Assume that a *p*-BF has *m* cells, *N* elements, *k* hash functions, and *L* different faces. Assuming that the name prefixes are equally distributed to each face, the number of elements for each face set, *n'*, is equal to $\frac{N}{L}$.

The probability that a specific cell has conflict value **X** can be calculated as follows. In programming, let *p_a* be the probability that at least one of the hash indexes for *n'* elements included in a specific face set points to a cell. Then, *p_a* can be calculated as

$$p_a = \left(1 - \left(1 - \frac{1}{m}\right)^{kn'}\right). \quad (3)$$

Since the elements included in a face set have the same output face, even though multiple hash indexes indicate the same specific cell, the cell has the output face value and not the conflict value.

Let *p_b* be the summation of the probabilities that at least one of the hash indexes for elements not included in the specific set indicates the same cell:

$$p_b = \sum_{i=1}^{L-1} \left(1 - \left(1 - \frac{1}{m}\right)^{k(N-in')}\right). \quad (4)$$

In calculating p_b , i is increased in order to avoid duplicate calculations. Then, the probability that a specific cell has conflict value $\mathbf{X}$ is p_a multiplied by p_b .

4.1. Indeterminable Probability

When querying the p -BF, if all of the k cell values of an input are *conflicts*, then the output face of the input cannot be identified, which is the *indeterminable* case. The *indeterminable* probability should be handled separately depending on whether or not the input is included in programming set S . In other words, the *indeterminable* probability ($P(I_t)$) is

$$P(I_t) = P(S)P(I|S) + P(S^c)P(I|S^c). \quad (5)$$

$P(I|S)$ is the probability that all hash indexes for an input among n' elements of a specific face set included in programming set S have conflict values. $P(I|S^c)$ is the probability that all of the hash indexes for an input not included in programming set S have conflict values.

In order to obtain $P(I|S)$, in querying an input among n' elements of a specific face set included in programming set S , let p_{ic} be the probability that the cell with the conflict value is indicated by a hash index of the input, which means the probability that at least one of the hash indexes for $(N - n')$ elements not included in the specific face set also programmed the cell. Then, p_{ic} is as follows:

$$p_{ic} = \left(1 - \left(1 - \frac{1}{m}\right)^{k(N-n')}\right). \quad (6)$$

In order for the input to be *indeterminable*, every hash index should indicate the conflict cell. Hence, $P(I|S) = p_{ic}^k$.

In order to obtain $P(I|S^c)$, in querying an input not included in programming set S , let p_{oc} be the probability that the cell with the conflict value is selected by a hash index for the input. The p_{oc} is the same as the probability that the selected cell has the conflict value, and hence, from Labels (3) and (4),

$$p_{oc} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn'}\right) \sum_{i=1}^{L-1} \left(1 - \left(1 - \frac{1}{m}\right)^{k(N-in')}\right). \quad (7)$$

Hence, $P(I|S^c) = p_{oc}^k$.

Based on Labels (5)–(7), the *indeterminable* probability

$$P(I_t) = P(S) \cdot p_{ic}^k + P(S^c) \cdot p_{oc}^k. \quad (8)$$

4.2. False Face Return Probability

Since a Bloom filter never produces false negatives, the inputs included in the programming set do not return false faces, even though they may return indeterminable results. Hence, the p -BF needs to be analyzed for *false face return* caused by negative inputs. In querying an input included in S^c , let p_{op} be the probability that a specific cell with a face value is selected. Then, p_{op} can be calculated as

$$p_{op} = \left(1 - \left(1 - \frac{1}{m}\right)^{kn'}\right) \left(1 - \frac{1}{m}\right)^{k(N-n')}. \quad (9)$$

The p_{op} is the product of the probability that at least one of the k hash indexes for n' elements included in the specific face set indicates a cell and the probability that none of the k hash indexes for $(N - n')$ elements not included in the set indicate the cell.

From Labels (7) and (9), the *false face return* probability

$$P(F_t) = L \sum_{i=1}^k \binom{k}{i} p_{op}^i \cdot p_{oc}^{k-i}. \quad (10)$$

$P(F_i)$ is the product of the probability that i of the k hash indexes return a specific face and the probability that $(k - i)$ of the k hash indexes return *conflicts*. In other words, when all of the cells, excluding conflict cells, have the same specific value, the value is returned as the output face. L is multiplied since $P(F_i)$ is the summation of the *false face return* probabilities for each face.

5. Performance Evaluation

The performance evaluation was carried out with C++ language using URL names provided by ALEXA [30]. In order to construct the FIB tables, we created three routing sets by randomly extracting 10,000, 50,000, and 100,000 names among 1 million names. For input sets to perform FIB lookup, three times the size of the routing set is used. One-third of each input set has names that are included in the corresponding routing set, while the remaining two-thirds has names that are not included in the set. The hash function used for our simulation is a 64-bit cyclic redundancy check (CRC) generator. A number of hash indexes are extracted by using different combinations of bits in the CRC code obtained from the CRC generator [9].

5.1. Routing Set Characteristics

Table 1 shows the characteristics of each set used for the construction of the FIB tables. As the number of elements included in each set, N refers to the number of name prefixes programmed to p -BF. As the number of nodes formed in a level-priority trie (excluding the root node), T refers to the number of nodes programmed to l -BF. The depth of the LPT is much smaller than the maximum number of components in a name prefix, since all of the empty nodes except for the root are deleted. The cell size of l -BF (c_l) depends on the maximum number of components, since l -BF returns a matching level. The cell size of p -BF (c_p) depends on the number of output faces. Assuming that the number of output faces is less than 254, eight bits are allocated for c_p .

Table 1. Routing set characteristics and Bloom filter cell sizes.

	10k	50k	100k
No. of names (N)	10,000	50,000	100,000
No. of valid nodes (T)	9763	48,602	96,557
Max. No. of components	23	32	32
LPT depth	4	5	5
Cell size of l -BF (c_l)	5	6	6
Cell size of p -BF (c_p)	8	8	8

Table 2 shows the data structure of the off-chip hash table storing name prefixes in the routing sets. Since neither the number of components of each name prefix nor the number of characters of each component is fixed, in order to assign a fixed width for a hash entry, each name prefix is converted into a 128-bit signature.

Table 2. Data structure of hash entry.

Field	Length (Bits)
name prefix	128
output face	8

5.2. Performance in Memory Requirements

Table 3 shows the comparison of the on-chip memory requirements with the NPT-BF and NPT-BF-chaining (with pre-computation) algorithms [9] when the size factor (α) of the Bloom filter is increased to 2, 4, and 8. The NPT-BF and NPT-BF-chaining algorithms include a standard Bloom filter which only provides membership information. In these algorithms, since all of the nodes of an NPT are

programmed to the Bloom filter, the on-chip memory requirement is equal to $\alpha S'$, where $S' = 2^{\lceil \log_2 S \rceil}$ and S is the number of nodes of the NPT.

The on-chip memory requirement for the proposed 2-phase BFs is the summation of the memory amounts for the l -BF and the p -BF. Since the search performance of the proposed algorithm is heavily dependent on the performance of the p -BF, the size factor of the p -BF is fixed at 16 in our evaluation. Hence, the memory amounts for the l -BF and the p -BF are equal to $\alpha T' c_l$ and $16 N' c_p$, respectively, where $T' = 2^{\lceil \log_2 T \rceil}$ and $N' = 2^{\lceil \log_2 N \rceil}$.

Since the NPT-BF and NPT-BF-chaining algorithms use a standard Bloom filter while our proposed algorithm uses two functional Bloom filters, the proposed 2-phase Bloom filters require more on-chip memory. However, the 2-phase Bloom filters are small enough to be stored in on-chip memories alone because the total amount is a few mega-bytes, as shown in Table 3.

Table 3. Comparison of on-chip memory requirements (KB).

Routing Set	α	NPT-BF	NPT-BF-Chaining	Proposed Algorithm
10k	2	16	16	276
	4	32	32	296
	8	64	64	336
50k	2	64	64	1120
	4	128	128	1216
	8	256	256	1408
100k	2	128	128	2240
	4	256	256	2432
	8	512	512	2816

Table 4 shows a comparison of the off-chip memory requirements. We assumed that the NPT should be implemented with an off-chip memory because of its requirement to store every node of the trie, where a node is represented by five fields: flag, child pointer, child edge, name prefix (128-bit signature), and output face. The memory requirements of the NPT are proportional to the number of nodes. The NPT-BF, the NPT-BF-chaining, and the proposed architecture make up a hash table with an off-chip memory. The hash table memory of the proposed architecture is smaller than that of the NPT-BF and the NPT-BF-chaining, since our proposed architecture only stores name prefixes, while the NPT-BF and NPT-BF-chaining need to store every node of an NPT.

Table 4. Comparison of off-chip memory requirements (KB).

Routing Set	NPT	NPT-BF	NPT-BF-Chaining	Proposed Algorithm
10k	682.4	1088.0	1088.0	272.0
50k	3372.5	4352.0	4352.0	1088.0
100k	6690.2	8704.0	8704.0	2176.0

5.3. Performance in On-Chip Memory Accesses

Table 5 shows a comparison of the on-chip search performance with those of other algorithms, which depends on the number of BF queries. A_b and W_b are the average number and worst-case number of Bloom filter queries, respectively. The NPT-BF and NPT-BF-chaining algorithms access the maximum level of an NPT in the worst case, and thus W_b is determined based on the depth of an NPT. The total number of Bloom filter queries of our proposed algorithm is determined by the summation of two Bloom filter queries. In the proposed l -BF, W_b is determined from the depth of the LPT, and A_b decreases as the Bloom filter size increases. Although the size of the p -BF is fixed at $16N'$, the number of queries of the p -BF depends on the size of the l -BF, since the querying to p -BF only occurs when the l -BF produces positive results.

Table 5. Comparison of on-chip search performance.

Routing Set	α	NPT-BF		NPT-BF-Chaining		Proposed Algorithm					
						<i>l</i> -BF		<i>p</i> -BF		Total	
		A_b	W_b	A_b	W_b	A_b	W_b	A_b	W_b	A_b	W_b
10k	2			3.136		1.838		1.345	17	3.182	19
	4	2.876	23	2.947	23	2.149	4	0.470	14	2.619	15
	8			2.884		2.179		0.378	6	2.557	8
50k	2			3.074		1.840		1.538	24	3.379	25
	4	2.926	32	2.948	32	2.204	5	0.517	14	2.721	16
	8			2.927		2.244		0.386	16	2.630	19
100k	2			3.116		2.053		1.314	31	3.367	32
	4	2.971	32	2.991	32	2.240	5	0.541	22	2.781	23
	8			2.971		2.287		0.400	8	2.687	11

It can be observed that the proposed algorithm results in significantly fewer accesses than other algorithms in the worst-case number of Bloom filter queries. When α is larger than 2, the proposed algorithm shows the smallest average number of Bloom filter queries. Even though the total A_b of the proposed algorithm is larger than that of the other algorithms when α is 2, this case is not generally used because of a large false positive rate.

5.4. Performance in Off-Chip Memory Accesses

The search performance of an FIB lookup algorithm mainly depends on the number of off-chip memory accesses because the time to access an off-chip memory is much slower than that to access an on-chip memory. Figures 4 and 5 show the average number and worst-case number of off-chip memory accesses, respectively, for each algorithm when the size factor of a Bloom filter is fixed at 8. For our proposed algorithm shown in Figures 4 and 5, note the numbers, since bars are barely seen. The proposed algorithm does not need to access the off-chip memory, except for when the *p*-BF produces *indeterminable* results. Hence, the proposed algorithm involves much fewer off-chip accesses compared to the other algorithms. As shown in Figure 4, the proposed algorithm can terminate the search without any off-chip hash table accesses for the 10k set, and only terminates the search with hash table accesses from 0.00002 to 0.00003 in the average for other sets. In other words, the name lookups for more than 99.99% of the inputs can be completed without accessing the off-chip hash table in our proposed algorithm.

Figure 4. Average number of off-chip memory accesses.

Figure 5. Worst-case number of off-chip memory accesses.

5.5. Comparison between Theoretic and Simulation Results

From Label (8), the *indeterminable* probability ($P(I_t)$) for our input sets is

$$P(I_t) = \frac{1}{3}p_{ic}^k + \frac{2}{3}p_{oc}^k \quad (11)$$

because $\frac{1}{3}$ of each input set has the names included in the corresponding routing set, and the remaining $\frac{2}{3}$ has names not included in the set. However, in the search procedure for an input, since not only is the input itself queried but also are substrings (prefixes) of the input, several queries for each input occur until the matching face is returned, and hence the probability should be calculated in consideration of the number of queries. Since some substrings are name prefixes included in the routing set while others are not included in the set, all of the substrings that are queried should be considered when calculating the probability. Table 6 shows the notation and definition of the query types. The number of queries of each type is obtained from the simulation results. Hence, the *indeterminable* probability ($P(I_t)$) for the queries of our input sets is

$$P(I_t) = \frac{Q_{tp} + Q_{ii}}{Q_t}p_{ic}^k + \frac{Q_{fp} + Q_{oi} + Q_n}{Q_t}p_{oc}^k. \quad (12)$$

$Q_{tp} + Q_{ii}$ refers to the queries for the hash keys (inputs and substrings) included in the routing set and $Q_{fp} + Q_{oi} + Q_n$ refers to the queries for the hash keys (inputs and substrings) not included in the routing set.

Table 6. Notation and definition of query types.

Symbol	Meaning
Q_{tp}	Number of queries with true positives
Q_{fp}	Number of queries with false positives
Q_{ii}	Number of queries for hash keys included in routing set among queries with <i>indeterminable</i>
Q_{oi}	Number of queries for hash keys not included in routing set among queries with <i>indeterminable</i>
Q_n	Number of queries with negatives (negatives are always true)
Q_t	Number of total queries ($Q_{tp} + Q_{fp} + Q_{ii} + Q_{oi} + Q_n$)

Simulations are carried out by only constructing the p -BF for the same routing set as the size of p -BF increases, then querying the p -BF from the longest level for an input name. According to Table 6, the *indeterminable* rate in the simulation is defined as

$$P(I_s) = \frac{Q_{ii} + Q_{oi}}{Q_t}. \quad (13)$$

According to Table 6, the *false face return* rate in the simulation is defined as

$$P(F_s) = \frac{Q_{fp}}{Q_{fp} + Q_{oi} + Q_n}. \quad (14)$$

If output faces are returned for the queries ($Q_{fp} + Q_{oi} + Q_n$) for the hash keys not included in the routing set, they are false faces.

Figure 6 shows the *indeterminable* rate for the p -BF, comparing between the theoretic result (described in Section 4) and simulation result, and Figure 7 shows the *false face return* rate for the p -BF. As shown in Figures 6 and 7, the simulation results are close to the theoretic results. As the size of the p -BF increases, the theoretical probabilities and simulation rates decrease similarly. In Figure 7a, it is shown that *false face return* does not occur in the simulation set 10k at $8N'$ or higher. As shown in Figures 6 and 7, the *indeterminable* rate and *false face return* rate in the proposed algorithm are very small, from 10^{-3} to 10^{-5} , for the reasonable size of a Bloom filter.

Figure 6. Comparison of *indeterminable* rate for p -BF: (a) 10k; (b) 50k; (c) 100k.

Figure 7. Comparison of *false face return* rate for p -BF: (a) 10k; (b) 50k; (c) 100k.

6. Conclusions

In this paper, a new Bloom filter-based algorithm is proposed for the FIB lookup. Our proposed 2-phase Bloom filter architecture consists of two functional Bloom filters: one Bloom filter for returning

a matching level and the other Bloom filter for returning an output face. The number of elements programmed to the Bloom filters in our proposed architecture is minimized by applying a level-priority trie. Since the proposed architecture is sufficiently small to be stored in on-chip memories, the FIB lookup performance is greatly improved by only querying the on-chip Bloom filters without accessing the off-chip hash table. The off-chip hash table is only accessed for the case where *indeterminable* results are returned at the second phase, and the simulation results show that more than 99.99% of the inputs obtain output faces by only querying Bloom filters stored in on-chip memories.

Author Contributions: conceptualization, H.B. and H.L.; methodology, H.L.; software, H.B.; validation, H.B.; formal analysis, H.B.; investigation, H.L.; resources, H.B.; data curation, H.B.; writing—original draft preparation, H.B.; writing—review and editing, H.L.; visualization, H.B.; supervision, H.L.; project administration, H.L.; funding acquisition, H.L.

Funding: This research was funded by the National Research Foundation of Korea (NRF), NRF-2017R1A2B4011254.

Conflicts of Interest: The authors declare no conflict of interest. The funders had no role in the design of the study; in the collection, analyses, or interpretation of data; in the writing of the manuscript, or in the decision to publish the results.

Abbreviations

NDN	Named Data Networking
CS	Content Store
PIT	Pending Interest Table
FIB	Forwarding Information Base
NPT	Name Prefix Trie
LPT	Level-Priority Trie
BF	Bloom Filter
LML	Last Matching Level
CRC	Cyclic Redundancy Check

References

1. Jacobson, V.; Smetters, D.K.; Thornton, J.D.; Plass, M.F.; Briggs, N.H.; Braynard, R.L. Networking Named Content. In Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies (CoNEXT '09), Rome, Italy, 1–4 December 2009.
2. 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 IEEE International Conference on Distributed Computing Systems (ICDCS), Macau, China, 18–21 June 2012; pp. 688–697.
3. Vasilakos, A.V.; Li, Z.; Simon, G.; You, W. Information Centric Network: Research Challenges and Opportunities. *J. Netw. Comput. Appl.* **2015**, *52*, 1–10. [[CrossRef](#)]
4. Mun, J.; Lim, H. Cache sharing using Bloom filters in named data networking. *J. Netw. Comput. Appl.* **2017**, *90*, 74–82. [[CrossRef](#)]
5. Saxena, D.; Raychoudhury, V. Radiant: Scalable, Memory Efficient Name Lookup Algorithm for Named Data Networking. *J. Netw. Comput. Appl.* **2016**, *63*, 1–13. [[CrossRef](#)]
6. Saxena, D.; Raychoudhury, V. N-FIB: Scalable, Memory Efficient Name-Based Forwarding. *J. Netw. Comput. Appl.* **2016**, *76*, 101–109. [[CrossRef](#)]
7. Zhang, L.; Claffy, K.; Crowley, P.; Papadopoulos, C.; Wang, L.; Zhang, B. Named Data Networking. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 66–73. [[CrossRef](#)]
8. Yi, C.; Afanasyev, A.; Wang, L.; Zhang, B.; Zhang, L. Adaptive Forwarding in Named Data Networking. *ACM SIGCOMM Comput. Commun. Rev.* **2012**, *42*, 62–67. [[CrossRef](#)]
9. 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. [[CrossRef](#)]
10. 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), Kathmandu, Nepal, 5–9 December 2011; pp. 1–5.

11. Seo, J.; Lim, H. Bitmap-based Priority-NPT for Packet Forwarding at Named Data Network. *Comput. Commun.* **2018**, *130*, 101–112. [[CrossRef](#)]
12. Muñoz, C.; Wang, L.; Solana, E.; Crowcroft, J. I(FIB)F: Iterated bloom filters for routing in named data networks. *arXiv* **2017**, arXiv:1609.02532.
13. Panda, P.; Dutt, N.; Nicolau, A. On-Chip vs. Off-Chip Memory: The Data Partitioning Problem in Embedded Processor-Based Systems. *ACM Trans. Des. Autom. Electron. Syst.* **2000**, *5*, 682–704. [[CrossRef](#)]
14. Lim, H.; Kim, S. Tuple Pruning Using Bloom Filters for Packet Classification. *IEEE Micro* **2010**, *30*, 48–59. [[CrossRef](#)]
15. Lim, H.; Lim, K.; Lee, N.; Park, K. On Adding Bloom Filters to Longest Prefix Matching Algorithms. *IEEE Trans. Comput.* **2014**, *63*, 411–423. [[CrossRef](#)]
16. Mun, J.; Lim, H. New Approach for Efficient IP Address Lookup Using a Bloom Filter in Trie-Based Algorithms. *IEEE Trans. Comput.* **2016**, *65*, 1558–1565. [[CrossRef](#)]
17. Lim, H.; Yim, C.; Swartzlander, E.E., Jr. Priority Tries for IP Address Lookup. *IEEE Trans. Comput.* **2010**, *59*, 784–794. [[CrossRef](#)]
18. Vagionas, C.; Maniotis, P.; Pitris, S.; Miliou, A.; Pleros, N. Integrated Optical Content Addressable Memories (CAM) and Optical Random Access Memories (RAM) for Ultra-Fast Address Look-Up Operations. *Appl. Sci.* **2017**, *7*, 700. [[CrossRef](#)]
19. Bloom, B. Space/time tradeoffs in in hash coding with allowable errors. *Commun. ACM* **1970**, *13*, 422–426. [[CrossRef](#)]
20. Broder, A.Z.; Mitzenmacher, M. Network Applications of Bloom Filters: A Survey. *Internet Math.* **2004**, *1*, 485–509. [[CrossRef](#)]
21. Tarkoma, S.; Rothenberg, C.E.; Lagerspetz, E. Theory and practice of Bloom filters for distributed systems. *IEEE Commun. Surv. Tutor.* **2012**, *14*, 131–155. [[CrossRef](#)]
22. Lim, H.; Lee, J.; Yim, C. Ternary Bloom Filter Replacing Counting Bloom Filter. *IEEE Commun. Lett.* **2017**, *21*, 278–281. [[CrossRef](#)]
23. Lim, H.; Lee, J.; Yim, C. Complement Bloom Filter for Identifying True Positiveness of a Bloom Filter. *IEEE Commun. Lett.* **2015**, *19*, 1905–1908. [[CrossRef](#)]
24. Bonomi, F.; Mitzenmacher, M.; Panigraha, R.; Singh, S.; Varghese, G. Beyond Bloom filters: From approximate membership checks to approximate state machines. In Proceedings of the 2006 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '06), Pisa, Italy, 11–15 September 2006; pp. 315–326.
25. Byun, H.; Lim, H. IP Address Lookup Algorithm Using a Vectored Bloom Filter. *Trans. Korean Inst. Electr. Eng.* **2016**, *65*, 2061–2068. [[CrossRef](#)]
26. Zhang, R.; Liu, J.; Huang, T.; Pan, T.; Wu, L. Adaptive Compression Trie Based Bloom Filter: Request Filter for NDN Content Store. *IEEE Access* **2017**, *5*, 23647–23656. [[CrossRef](#)]
27. Zhang, W.; Xu, Y.; Li, Y.; Zhang, Y.; Li, D. FlameDB: A Key-Value Store With Grouped Level Structure and Heterogeneous Bloom Filter. *IEEE Access* **2018**, *6*, 24962–24972. [[CrossRef](#)]
28. Moralis-Pegios, M.; Terzenidis, N.; Mourgias-Alexandris, G.; Vysokinos, K. Silicon Photonics towards Disaggregation of Resources in Data Centers. *Appl. Sci.* **2018**, *8*, 83. [[CrossRef](#)]
29. Muñoz, C.; Leone, P. Fragmented-Iterated Bloom Filters for Routing in Distributed Event-Based Sensor Networks. *Internet Distrib. Comput. Syst.* **2015**, *9258*, 248–261. [[CrossRef](#)]
30. Alexa the Web Information Company. Available online: <http://www.alexa.com> (accessed on 17 January 2019).