applied sciences

MDPI

Article

Scalable Name Lookup for NDN Using Hierarchical Hashing and Patricia Trie

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: 19 December 2019; Accepted: 1 February 2020; Published: 4 February 2020

Abstract: Named data networking (NDN) is a content-centric network for the future of the internet. An NDN a packet is delivered based on a content name instead of a destination IP address. The name lookup for packet forwarding is challenging because the content name is variable, and its space is unbounded. This paper proposes a novel name lookup scheme that employs a hashing technique combined with Patricia tries. In this scheme, hash tables are dynamically maintained according to the hierarchical structure of the name, so the name can effectively accommodate variable and unbounded content names. Unlike chaining, a Patricia trie compares a key string only once at a leaf node, so it provides a fast name lookup. The proposed lookup scheme is implemented and evaluated by using an Intel Core i7-2600 CPU and 4GB of memory. Experimental results show that our scheme gives good scalability and a high throughput of name lookup with a controlled memory consumption.

Keywords: named data networking; name lookup; hash table; Patricia trie; longest prefix matching

1. Introduction

Named data networking (NDN) is a promising structure for future internet architecture [1–3]. The packet delivery in NDN is not based on the destination address but the content name. Each NDN router forwards an incoming packet based on the name in the packet. It should look up a forwarding information base (FIB) to find the matching name prefix with a given content name. However, there can be more than one matching prefix. The longest one among them should be selected as the final result, and such a process is called longest prefix matching (LPM). For example, if an FIB contains both prefixes/com and /com/google, the best matching result for a given name/com/google/people is /com/google/.

The name in an NDN packet is human-readable and has a hierarchical structure that consists of several components that are separated by the slashes. The name is variable-length, while the length of the IP address is fixed as 32-bit. Moreover, the name space is unbounded. Such characteristics make the name lookup more complex and challengeable than the IP address lookup in [4].

Many articles have focused on high-speed name lookup schemes to meet the requirement of packet forwarding in wire speed [5–7]. Name lookup schemes are usually based on ternary content-addressable memory (TCAM), hashing, or trie structures. With a TCAM, it is hard to utilize the name lookup because the name may be quite long and its length may not be fixed. A trie-based name lookup may endure many memory accesses due to unbounded name space. Most hash-based name lookup schemes are based on flat-style full name prefixes, even though the name itself is hierarchically constructed with components.

The flat-style keys may require redundant storage when those are stored in a hash table. For example, /com/google and /com/google/people are individually stored even though they have a common prefix /com/google. If each name component is separately hashed, such redundancy can be

avoided. Most hashing schemes use Bloom filters to find the key length because it is not possible to know the length in advance before hashing. A component-based hierarchical hashing scheme can resolve the above issues.

This paper proposes an efficient name lookup scheme that is based on a hierarchical hashing technique and a Patricia trie. The proposed scheme deals with the hash collision by using a Patricia trie instead of chaining, which reduces the search time significantly.

The rest of this paper is organized as follows. Section 2 describes background knowledge about name-based forwarding and Patricia tries. In Section 3, we study previous works related to name-based lookup. Section 4 explains the proposed hybrid NDN lookup engine to employ both the hash table and the Patricia trie. In Section 5, our scheme is implemented and evaluated by an FIB and a random trace. Experimental results are discussed as well. Finally, Section 6 concludes the paper.

2. Background

2.1. Forwarding in NDN

In NDN, a packet is forwarded according to the name in the packet. The name is represented as a sequence of components each of which is separated by a slash [8]. For example, a name, /com/google/www/drive contains four components: /com, /google, /www, and /drive.

There are two types of NDN packets: an interest packet and a data packet [3]. Figure 1 shows how these packets are forwarded. Each NDN router contains three tables: a content store (CS), a pending interest table (PIT), and a forwarding information base (FIB). The interest packet is generated by consumers who want to obtain some content from a producer. When more than one consumer is interested in the same content and sends the same interest packet, the router delivers only one interest packet. The data packet is replied by the producer that has the content associated with the interest packet. It is forwarded along the reverse direction to which the corresponding interest packet is sent.

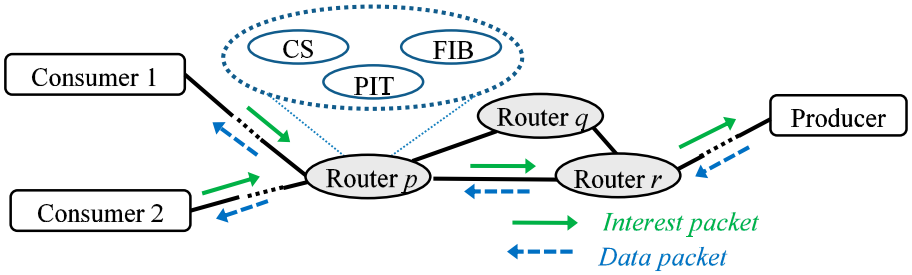

Figure 1. Overview of packet forwarding in named data networking (NDN). CS: content store. PIT: pending interest table. FIB: forwarding information base.

Figure 2 shows the forwarding processes for interest and data packets in an NDN router by using a CS, a PIT, and an FIB. A CS is a caching storage that stores the content of data packets for reuse when the same interest arrives. A PIT maintains the received interest names along with their incoming interfaces. If a received interest name is the same as one of names already stored in a PIT, then its incoming interface is appended to that entry in the PIT and the received packet is not forwarded further. An FIB is a lookup structure to find name prefixes.

Figure 2a shows the process to forward an interest packet in an NDN router. When receiving an interest packet, the router first checks whether the name in the packet exists in a CS or not. If the matched name is in a CS, the corresponding data packet is sent back as a reply. Otherwise, it is checked whether the name in the packet is already located in the PIT or not. If there is the matched name in the PIT, the incoming interface of the packet is added to the PIT. Otherwise, the packet is finally forwarded according to the result in the FIB. The matching method in the FIB is the longest prefix matching, while that in the CS or the PIT is the exact matching.

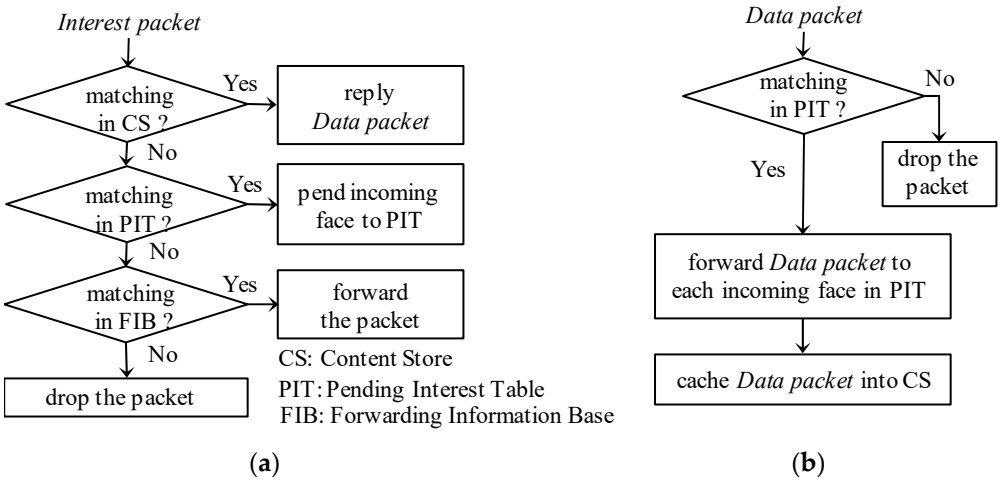

Figure 2. Forwarding processes in NDN router for: (**a**) interest packet and (**b**) data packet.

When an interest packet arrives in the producer, the corresponding data packet is constructed and sent back. Figure 2b shows the forwarding process for the data packet. For an incoming data packet, a router searches for its name in the PIT and sends the packet to each incoming interface that is pending for the name in the PIT. The packet is cached in the CS for the future use as well.

2.2. Patricia Trie

A trie is a kind of tree structure and has been used in several IP address lookup schemes [9,10]. A binary trie is the most basic trie whose degree is two. Each node in a binary trie is expressed by a bit string to represent the path from the root to that node. Figure 3a shows an example of a binary trie where all prefixes (*a*–*d*) are indicated by filled circles. As shown in this figure, the binary trie needs many node accesses. Note that a node with a single child induces unnecessary node access. The path compression to eliminate such a node can reduce the number of node accesses.

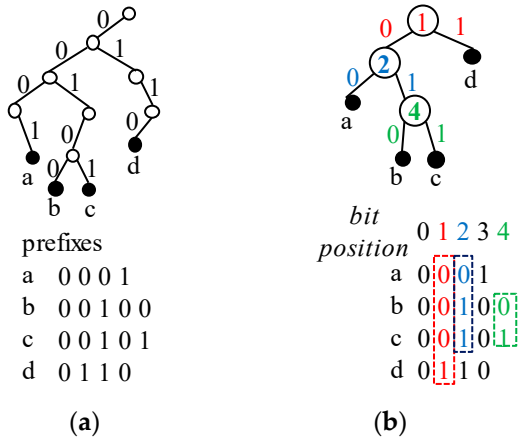

Figure 3. Two trie structures to contain four prefixes: (**a**) binary trie and (**b**) Patricia trie.

A Patricia trie is a binary trie in which any node without its sibling is merged into its parent [4]. Figure 3b shows a Patricia trie corresponding to the binary trie in Figure 3a. Note that the degree of each node is 0 or 2 in a Patricia trie. In a binary trie, each bit position is checked in turn. However, in a Patricia trie, only specific bit positions are checked. In Figure 3b, the number in a circle indicates the bit position to be checked. If the incoming key is 00100, the bit positions 1, 2 and 4 are checked in turn. Thus, the corresponding values are 0, 1 and 0 so it reaches the prefix *b*.

3. Related Works

Most previously proposed name lookup schemes can be classified into TCAM-based [11], trie-based [12–15], and hash-based [16–21] ones.

A TCAM is an associative memory that searches all the entries in parallel. Each cell can store a value of * (do not care) as well as 0 or 1, so it is fairly suitable to store and lookup variable-sized prefixes. Therefore, many IP address lookup schemes have utilized a TCAM [22,23]. However, it is hard for a TCAM to accommodate long prefixes and a large number of prefixes, so it is restrictively used in name lookup. Sun et al. proposed a name-based LPM scheme that uses content-addressable memory (CAM) [11]. After dividing every prefix into several fixed length blocks and one variable length block, it hashes the former fixed blocks and stores them in binary CAMs. The latter variable block is stored into a TCAM.

Trie-based name lookup schemes may suffer overhead due to a number of memory accesses. Because a name may be a long string of characters, name lookup may need more memory accesses in a trie than an IP lookup. Lee et al. tried to decrease that by path compression [12]. In [13], Li et al. proposed port information assist trie (P-trie) to prune subtries based on output interface information.

Wang et al. presented a high-speed name lookup scheme by mapping states in a logical trie for an FIB into physically parallel modules [14]. They also presented the name component encoding (NCE) scheme to provide a memory saving code allocation mechanism that employs two types of tries called the component character trie (CCT) and the encoded name prefix trie (ENPT) [15]. Before the name lookup, each component's code in a name can be obtained by the CCT. The ENPT is a prefix trie that uses the encoded components to represent a prefix, and it is used for the longest prefix lookup. The state transition array was developed to implement the CCT and the ENPT for saving the required memory space.

Most hash-based name lookup schemes compare the name with candidate prefixes as a section of components for their effectiveness, and some of them utilize Bloom filters for efficiency. Wang et al. proposed a name lookup scheme employing a two-stage Bloom filter called the NameFilter [16]. In the first stage, the length of the probing name prefix is determined by means of Bloom filters. The second stage can find the output interface by searching the merged Bloom filters that are classified by the number of output interfaces. They also proposed a hash-based LPM scheme to reduce the search space by the lookup in the order that was generated by using dynamic programming [17].

Yuan et al. proposed a name lookup scheme that employed the binary search of hash tables according to the number of name components based on not the real name components but their fingerprints [18]. Lee et al. presented two name lookup schemes that employed on-chip Bloom filters and off-chip hash tables [19]. Recently, Dai et al. presented an index data structure called the Bloom filter-aided hash table (BFAST) for NDN name lookup [20]. The BFAST can reduce the number of memory accesses by achieving the load balance among hash slots by using a counting Bloom filter. An auxiliary data structure is used to indicate the smallest counter at insertion time. They proposed a dynamic adaptation for lookup and deletion to adjust the load values, and they also proposed an indexing scheme to reduce the memory consumption.

Some schemes have dealt with LPM for name lookup by employing both a hash and a trie [24,25]. Quan et al. presented a hybrid structure by using both fixed-sized counting Bloom filters for upper components and variable-sized tries for lower components to realize efficient NDN lookup [24]. After defining the exact string differentiation problem based on the speculative forwarding, Yuan et al. proposed a fingerprint-based Patricia trie and a fingerprint-based hash table for name prefix lookup [25]. A collision-free fingerprint-based hash table was also developed by supplementing the collision table to contain the collided name strings.

4. Proposed Name Lookup Scheme

Scalable name lookup is challenging for several reasons. First, the name space is unbounded, and the length of a name is variable. An FIB should accommodate variable-length name prefixes. Second,

it is expected that the number of name prefixes is much larger than network prefixes in an IP network. Lastly, the lookup process should find the longest matching name prefix.

Hashing usually provides fast key lookup at a constant time if the key length is not very large. However, key length is variable, and we need to find the longest matching name prefix during NDN lookup instead of the exact matching name prefix. Thus, hashing needs to be modified or enhanced if used.

The set of the name prefixes is hierarchically structured in NDN, so the upper-level component strings may be stored redundantly if each full name prefix string is stored in the flat-style structure. For instance, if two name prefixes /com/google and /com/facebook are stored in the flat structure, the first level component string/com would be stored twice. Additionally, hashing time is redundant while finding the longest matching name prefix. For a given search name/com/google, we need to compute hash codes, h("/com") and h("/com/google") because we do not know which one is the longest matching prefix in advance. Thus, the hashing time for "/com" is doubled in this instance. To eliminate component redundancy and support the longest matching efficiently, we adopted a hierarchical structure of hash tables, each of which was basically used for a component.

4.1. Overall Structure of FIB

The proposed lookup structure consisted of hierarchical hash tables in which each level supported the search structure of name components. Figure 4 depicts the two-level name hierarchy. Each level consisted of several hash tables and their overflow areas. Patricia tries were used for the overflow areas instead of linked lists.

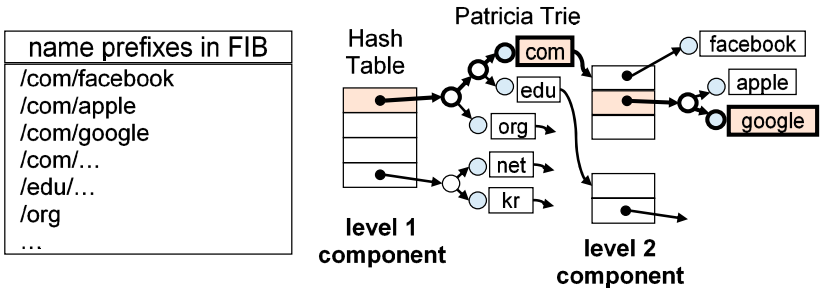

Figure 4. Proposed structure of the forwarding information base (FIB).

For a given interest name/com/google/people, the hash value of the first component /com is computed, and then a bucket is determined in the root-level hash table. In this example, unfortunately, several components, which were /com, /edu, and /org, collided in the same bucket and were stored in an overflow area. By using a Patricia trie, the component /com could be found in a leaf node, and it connected to a next level hash table. In the same way, it searched for /google, which was the longest result for the interest name.

4.2. Hash Table

All hash tables are organized in the same format regardless of their component levels. Figure 5 shows the structure of the hash table and two types of hashing schemes. As shown in Figure 5a, the hash table consisted of three fields: *htab_size*, *tot_no_keys*, and *htab* [*htab_size*]. The former two fields, *htab_size* and *tot_no_keys*, were the hash table size and the total number of stored keys, respectively. Those were supplementary fields to dynamically adjust the hash table size. For the adjustment purpose, we exploited a load factor that was calculated by *tot_no_keys*/*htab_size*. The value of *htab_size* was always a power of two. The field *htab* was an array of pointers to the Patricia tries or the linked lists, and it was actually the hash table itself.

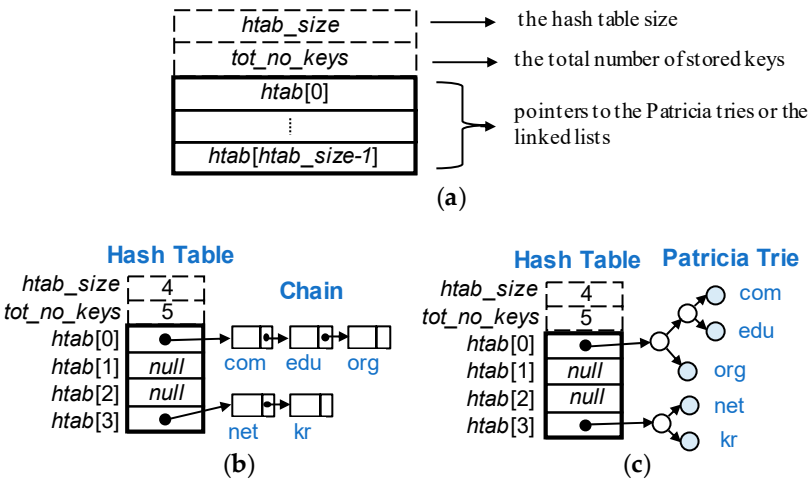

Figure 5. The structure of the hash table and two types of hashing schemes: (**a**) hash table format, (**b**) separate chaining, and (**c**) Patricia trie-based.

A collision problem may occur when the same hashing value is produced for different keys. Our hashing scheme used a Patricia trie for the overflow area instead of a linked list to resolve the collision problem. Figure 5b,c compare separate chaining with our scheme by an example where five keys were stored: *com*, *edu*, *org*, *net* and *kr*. Suppose h("*com*") = h("*edu*") = h("*org*") = 0, and h("*net*") = h("*kr*") = 3. Thus, {*com*, *edu*, *org*} and {*net*, *kr*} are stored in the 0-th and the 3rd buckets, respectively. In separate chaining, all the keys that are hashed in a bucket are maintained in a linked list, as shown in Figure 5b. On the other hand, in our scheme, the keys in a bucket were the leaves of a Patricia trie, as shown in Figure 5c. In this example, *htab_size* is 4 and *tot_no_keys* was 5, so the actual load factor was 1.25 (=5/4).

For a given search key K, the hashing time is usually $O(k)$ where k is the length of K. The selective bit extraction of a key reduces hashing time, but it may cause an undesirable result. Upon a successful search, every key in a linked list should be compared with a search key until the matched key is found in separate chaining. If there are m keys in a bucket, the total comparison time is $O(mk)$. On the other hand, in a Patricia trie, key comparison is only taken once in a leaf node. Only bit inspection is needed while traversing intermediate nodes in a Patricia trie. Overall comparison time is $O(h+k)$, where h is the height of a Patricia trie, and h is $\log_2 m$ in the best case and m in the worst case, so the comparison time is $O(m+k)$ in the worst. In practice, m is not so large, and k can be large.

Each hash table size is varied and can even be dynamically changed by means of a key insertion induced by an update of the FIB. This means one should not fix hash table size by using a static array. For scalability, we used dynamic arrays for hash tables in which its size can be doubled. The amortized cost of insertion and deletion for a dynamic array is $O(1)$ [26]. Given a limit of load factor β, we controlled the size of hash table to meet the restriction, $\alpha \leq \beta$ where α is the actual load factor. α is defined as d/N, where d is the number of stored keys and N is the number of buckets in the hash table. Whenever $\alpha > \beta$ by insertion of a new key, the hash table size is doubled. α represents the average number of keys in a bucket or the average number of leaves in Patricia tries. By controlling the hash table size, α satisfies the following restriction.

$$\alpha \leq \beta \text{ if } N = 1 \tag{1}$$

$$\beta/2 \leq \alpha \leq \beta \text{ if } N \geq 2 \tag{2}$$

4.3. Construction of Patricia Trie

Since a key is a character string, a Patricia trie should be constructed for characters. However, a Patricia trie node has only two branches, so a character string is represented by a binary character

code. In a Patricia trie, all keys are placed in leaves. However, a key can be a prefix of another key, and that key has to be placed in an internal node. To resolve this issue, we appended a terminating character '\0' to every key. Since no key had '\0' in the middle of the string, there was no proper prefix of another key. Figure 6 shows an example of a Patricia trie for a set of keys, {a, ab, ac, bb, bn}. All keys were placed in leaves even though "a" was a prefix of "ab." The number inside a Patricia node indicates a bit position to be inspected. The most significant bit position for distinguishing these keys was 6, as shown in Figure 6a. Keys "a", "ab", and "ac" had 0 in the 6th bit, while keys "bb" and "bn" had 0. Thus, the root node of the Patricia trie in Figure 6b had a value of 6. Additionally, the left subtrie of the root node had the keys "a", "ab", and "ac", while the right subtrie had "bb" and "bn." Since a component is a variable-length string, it could not be stored in a fixed-size leaf node. The key string was stored in an auxiliary structure, as shown in Figure 6b. Key string comparison occurred once only when it reaches a leaf node.

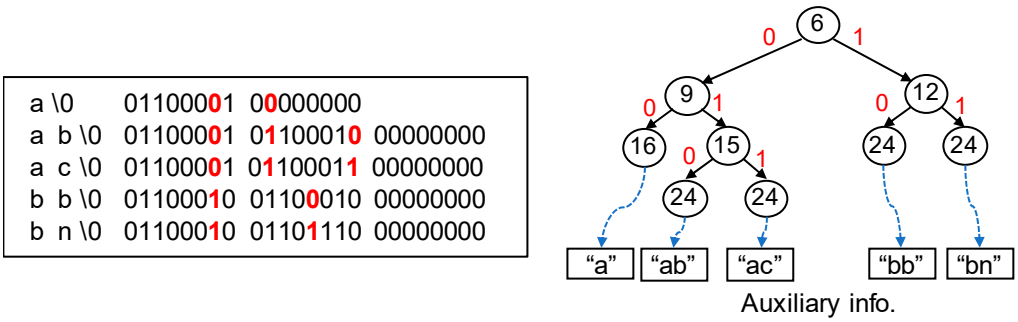

Figure 6. An example of key strings and its Patricia trie: (**a**) five key strings and their bit representations and (**b**) the corresponding Patricia trie to (**a**).

Figure 7 shows the structures of a Patricia node and its auxiliary information. A Patricia node consists of four fields: *bitpos*, *aux*, *child*[0], and *child*[1]. The first field *bitpos* represents bit position to be checked in order to determine which one to be searched—the left or the right trie. If the size of the *bitpos* field is n byte, the component consisting of up to $2^{8n}/8$ characters can be handled. Even though *bitpos* is 2 bytes long, it is possible to deal with any component with up to 8192 characters, which is practically sufficient length of a component string. The second field *aux* is pointer to auxiliary information. A leaf node and its ancestors share this auxiliary information. The remaining fields *child*[0] and *child*[1] represent the left and the right child, respectively.

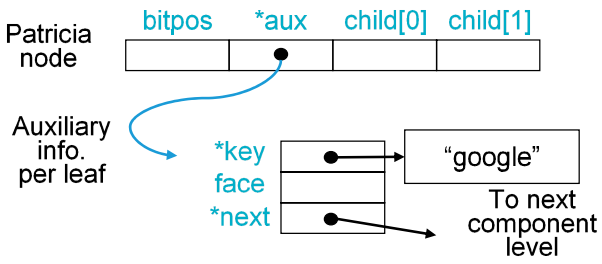

Figure 7. Patricia node and auxiliary information.

Auxiliary information has three fields: *key*, *face*, and *next*, as shown in Figure 7. Since the store for a key is dynamically allocated and pointed to by *key*, and there is no restriction on the length of a key string. *face* represents an output face number that a given packet is forwarded to. Whenever a key string is matched at leaf node, the face number *face* is reserved as a candidate for the matching result. At each level the candidate face number is updated, and lookup continues to perform at next component level that is directed by pointer *next*.

4.4. Lookup Algorithm

Whenever an interest name is given for lookup, the name lookup engine extracts and processes each component as starting from the first level. The hash code of an extracted component locates an entry in the hash table. Each hash table entry has the pointer to a Patricia trie, which is provided for overflow area. Thus, it needs to search the Patricia trie to find a matching result. The search result consists of a face number and a pointer to the next level hash table. If the Patricia search succeeds, the found face is recorded as the potentially best result, and lookup proceeds with a next level component. The lookup process stops at the level where the search fails, and the recorded last result becomes the final result, i.e., the longest matching result. As an example, suppose /a1/b1/c1 is given as an interest name, and the given set of prefixes is {/a1/b1, a1/b1/c2}. Then, the lookup stops at the third level, because c1 is not matched with c2 at this level. Consequently, the best result becomes /a1/b1, which is the last matching prefix.

The whole lookup procedure is described in Algorithm 1. Additionally, Algorithm 2 shows how to search a Patricia trie. When a component string is given, a specific bit that is designated by *bitpos* is extracted. According to the bit, the left or the right child is traversed, and the procedure is repeated until a leaf node is reached.

Algorithm 1: Algorithm for Name Lookup in FIB

Input: *name* that is an Interest name
1: *htab* = *root_hash_table*
2: *best_result* = *default_face*
3: **forever do**
4: *c* = next_component(*name*)
5: *h* = hash(*c*)
6: *aux* = Search_Patricia(*htab*[*h*], *c*)
7: **if** *aux* is **not** null **and** *c* == *aux*→*key* **then**
8: *best_result* = *aux*→*face*
9: *htab* = *aux*→*next*
10: **else**
11: **return** *best_result*

Algorithm 2: Search Algorithm for Patricia Trie

Input: a Patricia node *t* and a component *c*
1: **if** *t* is null then
2: **return** null
3: **for** *t* is not a leaf do
4: *b* = GET_BIT(*c*, *t*→*bitpos*)
5: *t* = *t*→*child*[*b*]
6: **return** *t*→*aux*

5. Experiment

5.1. Experiment Environments

The proposed scheme was implemented in C++ code and evaluated by an FIB and a random trace. Since there is no real FIB for NDN, we used a name set in an IP network instead, which was a dmoz name set [27]. It contained 2,488,260 names, and the maximum component level was 8. An input trace was randomly generated as a collection of ten million names that followed Zipf's law.

The computing equipment to perform the name lookup consisted of an Intel Core i7-2600 CPU (3.40GHz) with 8192KB cache and 4GB main memory. The lookup throughput was measured for a 10M trace by using a single thread.

5.2. Experiment Results

All name prefixes were obtained from a dmoz name set. The original data set was in the form of URLs such as "http://www.google.com/words", but we converted it into the form of an NDN name like "/com/google/www/words." The number of components in a name ranges from 1 to 8, as shown in Figure 8. The dominant component level was 3, and the number of components of that level was 1,883,555, which was 75.7% of the total. This implied that hierarchical hashing from the first level was more favorable than flat-style hashing starting from the last level.

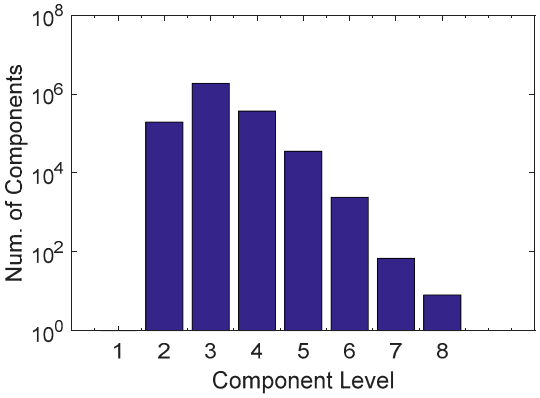

Figure 8. Distribution of component level in a dmoz name set.

The performance of our scheme was evaluated with throughput that was measured in packets per second (pps), as shown in Figure 9. The throughput is inversely proportional to the maximum load factor β. As explained in Section 4.2, β is the upper bound of the actual load factor of a hash table. The higher β value allows more keys to be accommodated in a hash table. This implies there can be more chances of hash collision.

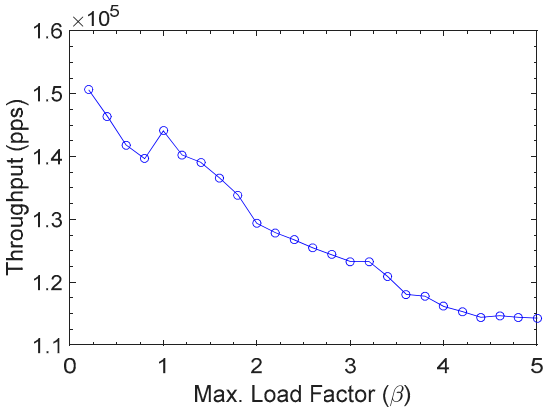

Figure 9. Throughput according to load factor.

There was a growing tendency for performance as β decreased, but a high memory usage was expected when β was set to low value. Figure 10 shows a trade-off between memory usage and lookup latency. The lookup latency was calculated by dividing the elapsed time by the number of processed packets. The memory usage was drastically decreased till β reached 1.0, whereas it kept stable after that point. Thus, latency and memory usage could compromise at around β = 1.0.

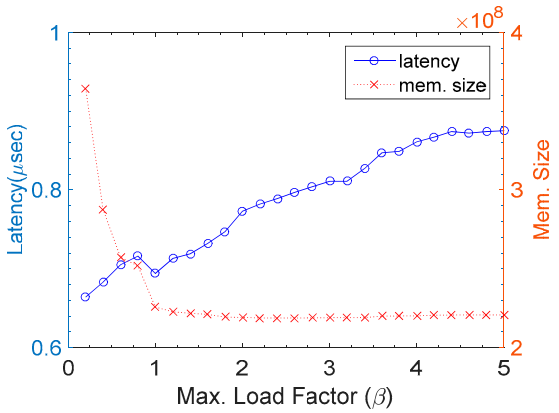

Figure 10. The lookup latency and required memory size.

For $\beta < 1.0$, the size of hash table should be higher than the number of stored keys. For example, if $\beta = 0.5$ and the number of stored keys becomes 5, then the hash table size should be set as higher than or equal to 10. This implies that half of the hash buckets were likely unoccupied. Figure 11 shows that the number of hash buckets gradually approached the number of occupied buckets as β went higher.

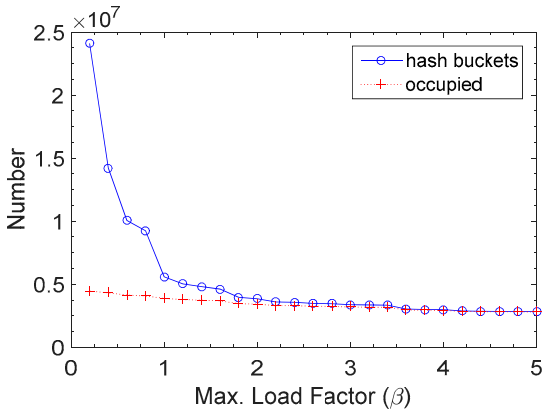

Figure 11. Comparison on the hash table size and the number of occupied buckets.

Figure 12 compares the Patricia trie with a linked list in maximum depth. The depth of the Patricia trie was much lower than that of the linked list. The depth of the linked list was approximately proportional to β, whereas that of the Patricia trie very slowly increased as β went higher. Since the depth value accounted for lookup time in the worst case, it was expected that a worst-case lookup takes less time in the Patricia trie than a linked list.

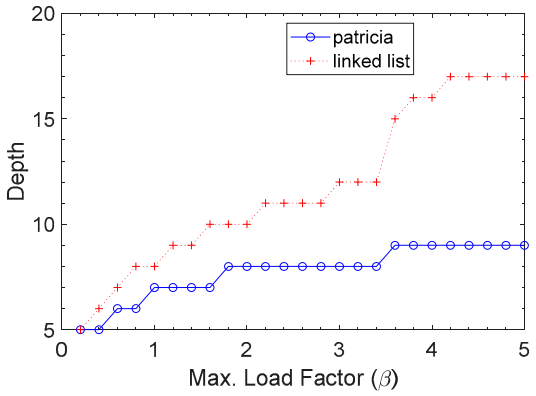

Figure 12. Maximum depth of the Patricia trie and a linked list.

Figure 13 compares the Patricia trie with the linked list in regards to the average number of node accesses and the memory usage. The lookup time highly depended on the number of memory accesses. Thus, it took more time when the average number of node accesses was higher. It is shown in Figure 13a that the Patricia trie was better than the linked list as β increased. However, there was a trade-off between the number of node accesses and the memory usage. The Patricia trie used more memory than the linked list because the Patricia trie required intermediate nodes, unlike the linked list. The memory usage of both is compared in Figure 13b.

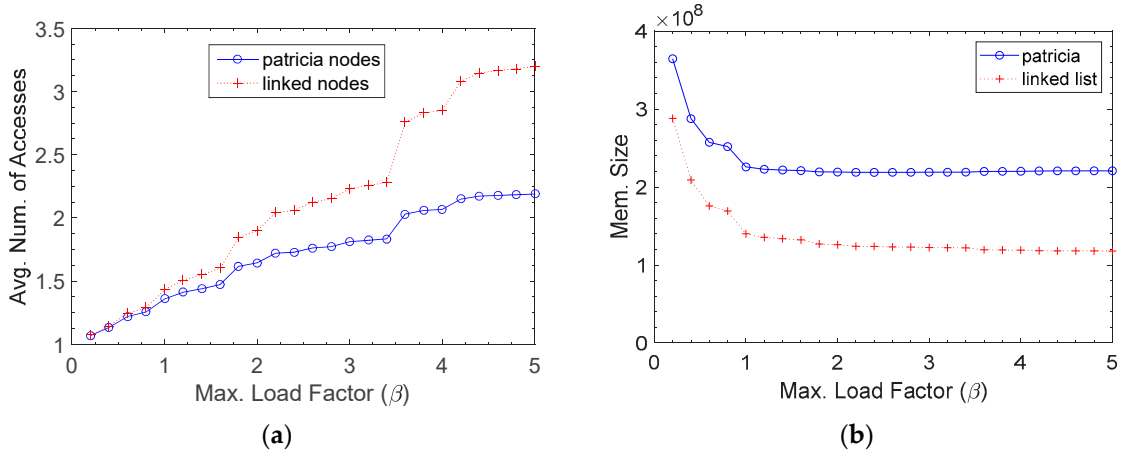

Figure 13. Comparison of a Patricia and a linked list: (**a**) average number of node accesses and (**b**) required memory size.

6. Conclusions

In NDN, packet delivery is based on the content name rather than the destination address. In a forwarding engine, name lookup is faced with some challenges. The content name is variable-length, and its space is unbounded. Additionally, the number of name prefixes is expected to be large compared to that of network prefixes in an IP network.

In this paper, we proposed a new name lookup scheme that is fast and scalable. Since there is no limit on the number of components and the length of a name string, multiple hash tables are hierarchically constructed in each component level for scalability. Each hash table size can dynamically change so that the scheme is not limited when new name prefixes are inserted. Upon hash collision, a Patricia trie enables a quick search in our scheme, while in separate chaining, every node should be visited in the worst case. Moreover, the key comparison in a Patricia trie occurs only once at a leaf node, whereas a key should be compared on every visited node in separate chaining. The experiment results showed that it provides not only high throughput but controlled memory consumption.

Author Contributions: Conceptualization, J.K.; Formal analysis, J.K. and J.K.; Investigation, M.-C.K. and M.S.S.; Methodology, J.K.; Validation, M.-C.K. and M.S.S.; Writing—original draft, J.K.; Writing—review and editing, J.K. 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. Ahlgren, B.; Dannewitz, C.; Imbrenda, C.; Kutscher, D.; Ohlman, B. A survey of information-centric networking. *IEEE Commun. Mag.* **2012**, *50*, 26–36. [CrossRef]
2. Pan, J.; Paul, S.; Jain, R. A survey of the research on future internet architectures. *IEEE Commun. Mag.* **2011**, *49*, 26–36. [CrossRef]
3. Zhang, L.; Afanasyev, A.; Burke, J.; Jacobson, V.; Crowley, P.; Papadopoulos, C.; Wang, L.; Zhang, B. Named data networking. *ACM SIGCOMM Comput. Commun. Rev.* **2014**, *44*, 66–73. [CrossRef]

4. Ruiz-Sanchez, M.A.; Biersack, E.W.; Dabbous, W. Survey and taxonomy of IP address lookup algorithms. *IEEE Netw.* **2001**, *15*, 8–23. [CrossRef]
5. Li, F.; Chen, F.; Wu, J.; Xie, H. Longest prefix lookup in named data networking: How fast can it be? In Proceedings of the 9th IEEE International Conference on Networking, Architecture, and Storage (NAS), Tianjin, China, 6–8 August 2014; pp. 186–190. [CrossRef]
6. Yuan, H.; Song, T.; Crowley, P. Scalable NDN forwarding: Concepts, issues and principles. In Proceedings of the 21st International Conference on Computer Communications and Networks (ICCCN), Munich, Germany, 30 July–2 August 2012; pp. 1–9. [CrossRef]
7. Kim, J.; Kim, J. An Efficient Prefix Caching Scheme for Fast Forwarding in Named Data Networking. *Stud. Inf. Control* **2018**, *27*, 175–182. [CrossRef]
8. Bari, M.F.; Chowdhury, S.R.; Ahmed, R.; Boutaba, R.; Mathieu, B. A survey of naming and routing in information-centric networks. *IEEE Commun. Mag.* **2012**, *50*, 44–53. [CrossRef]
9. Nilsson, S.; Karlsson, G. IP-address lookup using LC-tries. *IEEE J. Sel. Areas Commun.* **1999**, *17*, 1083–1092. [CrossRef]
10. Wu, Y.; Nong, G.; Hamdi, M. Scalable pipelined IP lookup with prefix tries. *Comput. Netw.* **2017**, *120*, 1–11. [CrossRef]
11. Sun, Y.; Egi, N.; Shi, G.; Wu, J. Content-based route lookup using CAMs. In Proceedings of the 2012 IEEE Global Communications Conference (GLOBECOM), Anaheim, CA, USA, 3–7 December 2012; pp. 2677–2682. [CrossRef]
12. Lee, J.; Lim, H. A new name prefix trie with path compression. In Proceedings of the 2016 IEEE International Conference on Consumer Electronics-Asia (ICCE-Asia), Seoul, Korea, 26–28 October 2016; pp. 1–4. [CrossRef]
13. Li, D.; Li, J.; Du, Z. An improved trie-based name lookup scheme for named data networking. In Proceedings of the 2016 IEEE Symposium on Computers and Communication (ISCC), Messina, Italy, 27–30 June 2016; pp. 1294–1296. [CrossRef]
14. Wang, Y.; Dai, H.; Jiang, J.; He, K.; Meng, W.; Liu, B. Parallel name lookup for named data networking. In Proceedings of the 2011 IEEE Global Telecommunications Conference (GLOBECOM 2011), Houston, TX, USA, 5–9 December 2011; pp. 1–5. [CrossRef]
15. Wang, Y.; He, K.; Dai, H.; Meng, W.; Jiang, J.; Liu, B.; Chen, Y. Scalable name lookup in NDN using effective name component encoding. In Proceedings of the 32nd International Conference on Distributed Computing Systems (ICDCS), Macau, China, 18–21 June 2012; pp. 688–697. [CrossRef]
16. Wang, Y.; Pan, T.; Mi, Z.; Dai, H.; Guo, X.; Zhang, T.; Liu, B.; Dong, Q. Namefilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters. In Proceedings of the 2013 INFOCOM, Turin, Italy, 14–19 April 2013; pp. 95–99. [CrossRef]
17. Wang, Y.; Qi, Z.; Dai, H.; Wu, H.; Lei, K.; Liu, B. Statistical Optimal Hash-based Longest Prefix Match. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Beijing, China, 18–19 May 2017; pp. 153–164. [CrossRef]
18. Yuan, H.; Crowley, P. Reliably scalable name prefix lookup. In Proceedings of the 2015 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Oakland, CA, USA, 7–8 May 2015; pp. 111–121. [CrossRef]
19. 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]
20. Dai, H.; Lu, J.; Wang, Y.; Pan, T.; Liu, B. BFAST: High-Speed and Memory-Efficient Approach for NDN Forwarding Engine. *IEEE/ACM TON* **2017**, *25*, 1235–1248. [CrossRef]
21. So, W.; Narayanan, A.; Oran, D.; Wang, Y. Toward fast NDN software forwarding lookup engine based on hash tables. In Proceedings of the 2012 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Austin, TX, USA, 29–30 October 2012; pp. 85–86. [CrossRef]
22. Ravikumar, V.C.; Mahapatra, R.N. TCAM architecture for IP lookup using prefix properties. *IEEE Micro* **2004**, *24*, 60–69. [CrossRef]
23. Zheng, K.; Hu, C.; Lu, H.; Liu, B. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *IEEE/ACM TON* **2006**, *14*, 863–875. [CrossRef]
24. Quan, W.; Xu, C.; Guan, J.; Zhang, H.; Grieco, L.A. Scalable name lookup with adaptive prefix bloom filter for named data networking. *IEEE Commun. Lett.* **2014**, *18*, 102–105. [CrossRef]

25. Yuan, H.; Crowley, P.; Song, T. Enhancing Scalable Name-Based Forwarding. In Proceedings of the 2017 ACM/IEEE Symposium on Architectures for Networking and Communications Systems, Beijing, China, 18–19 May 2017; pp. 60–69. [CrossRef]
26. Cormen, T.H.; Leiserson, C.E.; Rivest, R.L.; Stein, C. Hash tables. In *Introduction to Algorithms*, 3rd ed.; MIT Press: Cambridge, MA, USA, 2009; pp. 253–285.
27. Directory of World Wide Web. Available online: http://www.dmoz.org/ (accessed on 5 November 2016).