Statistical Optimal Hash-based Longest Prefix Match

Yi Wang
Huawei Future Network Theory Lab
Hong Kong
wy@ieee.org

Zhuyun Qi
Shenzhen Key Lab for Cloud Computing Technology & Applications (SPCCTA), School of Electronics and Computer Engineering, Peking University
Shenzhen, China
qizhuyun@gmail.com

Huichen Dai
Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University
Beijing, China
dhconly@gmail.com

Hao Wu
Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University
Beijing, China
wuhao.thu@gmail.com

Kai Lei[*]
Shenzhen Key Lab for Cloud Computing Technology & Applications (SPCCTA), School of Electronics and Computer Engineering, Peking University
Shenzhen, China
leik@pkusz.edu.cn

Bin Liu
Tsinghua National Laboratory for Information Science and Technology, Department of Computer Science and Technology, Tsinghua University
Beijing, China
liub@tsinghua.edu.cn

ABSTRACT

Longest Prefix Match (LPM) is a basic and important function for current network devices. Hash-based approaches appear to be excellent candidate solutions for LPM with the capability of fast lookup speed and low latency. The number of hash table probes, i.e. the search path of a hash-based LPM algorithm, directly determines the lookup performance. In this paper, we propose Ω-LPM to improve the lookup performance by optimizing the search path of the hash-based LPM. Ω-LPM first reconstructs the forwarding table to support random search [19], then it applies a dynamic programming algorithm to find the shortest search path based on the statistics of the matching probabilities. Ω-LPM concretely reduces the number of hash table probes via searching most of the packets in optimal search paths. Even in the worst case, the upper bound of the average search path of Ω-LPM is $1 + \log_2(N)$, here N is the length of the longest prefix in the routing table. The case studies of the name lookup in Named Data Networking and the IP lookup in current Internet demonstrate that Ω-LPM can shorten 61.04% and 86.88% search paths compared with the basic hash-based methods of name lookup [22] and IP lookup [12], respectively; furthermore Ω-LPM reduces 32.3% probes of the name lookup and 73.55% probes of the IP lookup compared with the optimal linear search. The experimental results conducted on extensional name tables and IP tables also show that Ω-LPM has both low memory overhead and excellent scalability.

[*]Corresponding author: Kai Lei, leik@pkusz.edu.cn.

CCS Concepts

•**Networks** → *Packet scheduling; Network experimentation;*

Keywords

Named Data Networking, Router, Name Lookup, Forwarding

1. INTRODUCTION

Packet transmission in current IP network is based on store-and-forward mechanism. The network devices, e.g. routers, first store the arriving packets in the input queue, and then forward these packets according to the next-hop information by looking up the destination addresses carried in packets' headers against the forwarding table. To forward packets both correctly and effectively, a router should maintain the topology of the network in a routing table located in the control plane, and meanwhile converts its routing table to forwarding table which is optimized for fast address lookup and is downloaded into the packet forwarding engine in the data plane. Generally, prefixes in a routing table are aggregated to reduce the number of entries in the forwarding table. Though prefixes aggregation could significantly reduce the memory consumption of Forwarding Information Base (FIB), the longest prefix match (LPM) operations, with which the destination address lookup should comply, will potentially slow down the lookup speed.

978-1-5090-6386-4/17 $31.00 © 2017 IEEE
DOI 10.1109/ANCS.2017.29

For improving the lookup performance to satisfy the requirements of ever-increasing wire-speed packet forwarding, a lot of LPM algorithms have been proposed. Generally, LPM algorithms can be classified into two categories: trie-based and hash-based. Trie-based LPM algorithms construct the FIB in the trie data structure, and look up each destination address by traversing the trie from its root to its descendant nodes until finding the longest prefix or fail. The advantages of trie-based algorithms are memory efficient, easy to be organized, and supporting fast incremental update. On the other side, trie-based mechanisms have low lookup performance as each transition from one node to its child node needs at least one memory access. For example, while an IPv4 address has 32 bits and each transition consumes 1 bit, the IPv4 address lookup against a trie-based FIB needs 25 memory accesses at most[1].

Hash-based LPM algorithms divide prefixes into different sets according to their lengths[2]. And the prefixes in the same set have the equal prefix length. The address lookup process is conducted as the following three steps: (1) the lookup engine generates all the possible prefixes (named candidate prefixes) of an address; (2) then the lookup engine matches these candidate prefixes in the different prefix sets to probe whether a candidate prefix is in a particular set; (3) finally, the longest matching prefix is the one looked for. Compared to trie-based LPM algorithms, hash-based LPM approaches can achieve higher lookup speed and are more suitable for the variable and unbounded length addresses, e.g. the URL similar naming mechanism in Content-Centric Networking [16].

Therefore, we focus on optimizing hash-based LPM in this paper. The essence of hash-based LPM search process is to find a particular search path corresponding to the longest matching prefix. Matching all the candidate prefixes in the sets is the basic and simplest LPM algorithm [12], which costs a lot of needless operations and slows down the lookup speed. An improved LPM algorithm sorts the candidate prefixes in a descending order of their lengths, and matches candidate prefixes from the longest one to the shortest one [5]. The LPM search process is terminated when one prefix is matched or all prefixes are dismatched.

The search path from the longest prefix to the shortest one, working in linear search mode, is the optimal search path without reconstructing the prefix sets. In the normal scheme of prefixes partition, the events — available candidate prefixes matching, are independent of each other. In other words, whether a candidate prefix is matched, the result does not affect other prefixes' matching results. Consequently, the longer prefixes still need to be checked if one shorter prefix is dismatched; or the shorter prefixes should be tested while one longer prefix is dismatched.

Inspired by the algorithm [19] of reconstructing the IP FIB to support binary search, we break through the linear search mode via reconstructing the FIB data structure to increase correlation among different prefixes matching events: ***a prefix is available if and only if all shorter prefixes are found in their corresponding sets***. That means (1) only the longer prefixes need to be checked when current prefix is matched; (2) or only the shorter prefixes need to be checked when current prefix is dismatched. Accordingly, the new prefixes partition scheme stands by random search mode that facilitates to improve the lookup speed by shortening the search path of hash-based LPM.

Our paper makes the following contributions:

1. The design of the dynamic programming algorithm for finding the optimal search paths of hash-based LPM. The dynamic programming algorithm based on the statistics of the matching probabilities has $O(N^3)$ time complexity, where N is the maximal prefix length. The optimal search paths, presented as a matrix, can be precalculated during the FIB reconstructing process. Besides that, the optimal search paths will be quickly recalculated along with the FIB's updates[3].

2. The analysis of the worst-case upper bound of the average search path found by the dynamic programming algorithm. Under the indication of the optimal search paths provided by Ω-LPM, most of the packets can achieve approximately optimal search path. Even in the worst case, the upper bound of the average search path of Ω-LPM is $1 + \log_2(N)$, which is the search path of binary search [28].

3. The case studies on the name lookup in Named Data Networking (NDN) [31] and the IP lookup in current Internet. The experimental results demonstrate that Ω-LPM effectively shortens the search path and improves the lookup speed. Ω-LPM achieves 2.94$\times$ and 5.97$\times$ speedups compared to the basic hash-based methods of name lookup [22] and IP lookup [12], respectively. Meanwhile, the experimental results also show that Ω-LPM has good scalability.

The rest of this paper is organized as follows. The framework of hash-based LPM approaches is introduced in Section 2. In Section 3, we first propose the dynamical programming algorithm to find the optimal search path, then we analyze the worst-case upper bound of the average search path of Ω-LPM. Two cases of the name lookup and the IP lookup are studied in Section 4, and the extensive experimental results conducted on prefix tables and IP tables are demonstrated in Section 5. After reviewing related work in Section 6, we conclude this paper in Section 7.

2. THE BACKGROUND OF HASH-BASED LONGEST PREFIX MATCH

2.1 Address naming mechanism

An address is a numerical/string label assigned to each object (e.g., a hardware device, a content, a service) to participate in the computer network which utilizes addresses for communication. The role of an address has been characterized as follows [2]: (1) A name indicates what we seek; (2) An address indicates where it is; (3) A route indicates how to get there.

The fixed-length IP addresses, i.e. 32-bit IPv4 or 128-bit IPv6, are applied by current IP networks. An IP address serves two

[1]IPv4 prefixes at least has 8 bits, therefore the first 8 bits are usually looked up once.

[2]For IP address, the prefix length is the bit number of an IP prefix; For name prefix, the prefix length is the component number of a name prefix.

[3]Note that the dynamic programming algorithm runs in the control plane of a router, and the lookup engine only utilizes the matrix to indicate the optimal search path of an address.

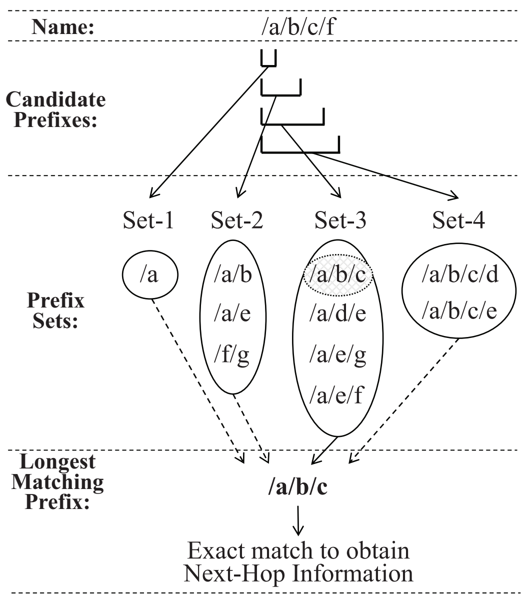

Figure 1: The general framework of a hash-based longest prefix match process.

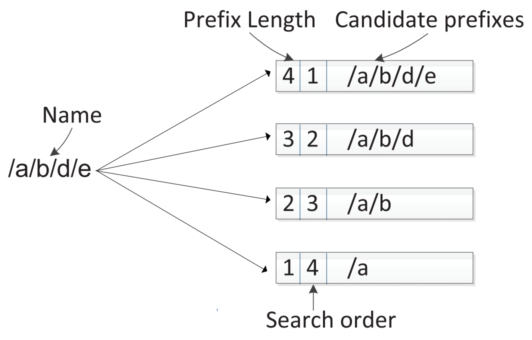

Figure 2: Probing the prefixes in a descending order based on its length.

principal functions: network interface identification and location addressing. Routers forward packets based on the destination IP address carried in a packet header. To forward packets correctly, each router maintains a routing table to store the topology of its global/local network. Since an IPv4/IPv6 prefix has at most 32-bit/128-bit, the length of an IPv4/IPv6 prefix is the number of its bits. Specifically, the aggregation granularity of IP prefixes is 1 bit. Different from IP-based network, Named Data Networking (NDN) [31], an instance of the Content-Centric Networking (CCN) [16] paradigm, employs URL-like string names to identify contents and devices. An URL-like name, composed of components, has hierarchical structure. For example, name "/org/IEEE/www" is composed of three components "org", "IEEE", and "www". Here, the character "/" is the delimiter. Given that each component in a name has unbounded length and the component number of a name is variable, an NDN name has variable and unbounded length. Specifically, the length of an NDN name is its component number and the name prefixes in a router's routing table are aggregated in the granularity of a component.

In any case of address naming mechanism (e.g., IP address, NDN name), as long as it has hierarchical structure and can be aggregated, its lookup process should comply with the longest prefix match to guarantee correctness.

2.2 The framework of hash-based longest prefix match

Hash-based LPM algorithms implement fast lookup speed by leveraging the high lookup (exact match) performance of hash table. Logically, the prefixes in the FIB are partitioned into different sets according to their lengths. The prefixes having equal length are put into a same set, and prefixes in different sets have different lengths. To transfer the longest prefix match to the exact prefix match, the lookup engine first generates all the possible prefixes of the searched address as candidate prefixes; then it checks all the candidate prefixes to find the longest matching prefix.

Figure 1 illustrates the general framework of a hash-based LPM process. 10 prefixes in the FIB are partitioned into 4 sets: *Set-1* ∼ *Set-4*. When a packet arrives at a router with a destination address (an NDN name) "/a/b/c/f", the lookup engine looks up "/a/b/c/f" against the FIB to obtain the next-hop information. Specifically, the lookup engine first generates 4 candidate prefixes of "/a/b/c/f": "/a", "/a/b", "/a/b/c", and "/a/b/c/f". After that, each candidate prefix is searched against its corresponding prefix set where all prefixes have the same length (the component number). In this example, the longest matching prefix — "/a/b/c" is the exact one looked for. Finally, the lookup engine obtains the forwarding information by searching the longest matching prefix ("/a/b/c") against the hash table that stores the next-hop information.

2.3 The prior art

2.3.1 The optimal linear search scheme

The search process of an address can be terminated when the lookup engine finds the longest matching prefix. Based on this observation, the lookup engine first sorts the candidate prefixes in a descending order of prefix length, and then it iteratively probes the candidate prefixes against their corresponding sets from the longest one to the shortest. As a result, the first matching prefix is the longest one looked for; if all candidate prefixes are dismatched, the lookup engine returns null to next module. As depicted in Figure 2, 4 candidate prefixes of the address "/a/b/d/e" are sorted in a descending order of prefix length, and the search path is: "/a/b/d/e", "/a/b/d", "/a/b", and "/a". Assuming that the FIB contains 10 prefixes as demonstrated in Figure 1. After looking up "/a/b/d/e" against *Set-4* and "/a/b/d" against *Set-3*, the lookup engine probes "/a/b" in *Set-2* and finds that "/a/b" is a matching prefix. Consequently, the LPM search process is terminated and "/a/b" is returned as the longest matching prefix.

Probing the prefixes in a descending order is simple and it is the optimal linear search path for the original FIB. In the original FIB, prefix sets are independent of each other. In other words, whether a candidate prefix is matched, the result does not affect other candidate prefixes' match results. Accordingly, when a matching pre-

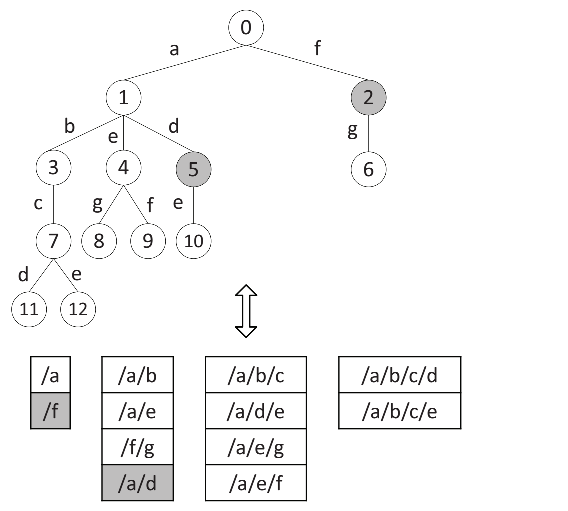

Figure 3: The trie and the reconstructed FIB of the original FIB in Figure 1.

fix is found, the lookup engine must check every candidate prefix which is longer than this matching one to confirm this one is the longest matching prefix.

2.3.2 *Reconstructing the FIB to support random search*

The search path described in Section 2.3.1 is the optimal one under the constraint of the original FIB, in which prefix sets are independent of each other. However, the lookup engine still works with linear search, and the average number of prefix probes is $O(N)$, here N is the length of an address.

To further shorten the search path of hash-based LPM, Waldvogel *et al.* [19] break through the linear search by reconstructing the FIB data structure to add correlation among different prefixes sets: a prefix can be available if and only if all shorter prefixes are found in their corresponding sets. That means only the longer prefixes need to be checked when one prefix is matched; or the shorter prefixes need to be checked when one prefix is dismatched. Consequently, the new prefix partition of the FIB supports random search to increase the lookup speed by shortening the search path.

Since a prefix in the FIB is composed of components (e.g., 1-bit in the IP address, one string component in the NDN name), each prefix has its ***meta-prefixes***. For example, the prefix “/a/b/c/d” has 3 meta-prefixes: “/a/b/c”, “/a/b”, and “/a”. If the all 3 meta-prefixes of “/a/b/c/d” are inserted into the FIB, the lookup engine can guarantee that no longer prefixes will be matched when the current prefix is dismatched. Note that we can distinguish the meta-prefixes with the original prefixes according to their forwarding information. Consequently, the correlation among different prefix sets is created, and the reconstructed FIB supports random search.

Intuitively, generating the meta-prefixes of a prefix in the original FIB and inserting them into the original FIB will cause the memory expansion problem of the reconstructed FIB. However, the number of prefixes including the meta-prefixes and the original prefixes in the reconstructed FIB is equal to the number of edges[4] in a trie that is applied to construct the original FIB. If we employ trie to organize the original FIB, all extra meta-prefixes of an original prefix will be stored in the trie as inner nodes. Figure 3 demonstrates the trie and the reconstructed FIB of the original FIB of Figure 1. In Figure 3, the trie and the reconstructed FIB both contain 12 edges/prefixes. In the trie, *node-2* and *node-5* corresponding to prefixes “/f” and “/a/d” are inserted as inner nodes; as well as in the reconstructed FIB, meta-prefixes “/f” and “/a/d” are inserted. Therefore, the prefix number of the reconstructed FIB is equal to the number of edges in the original FIB’s trie, and there is no memory space expansion problem in the reconstructed FIB.

[4]Note that the number of edges in a trie equals the number of nodes in a trie minus one.

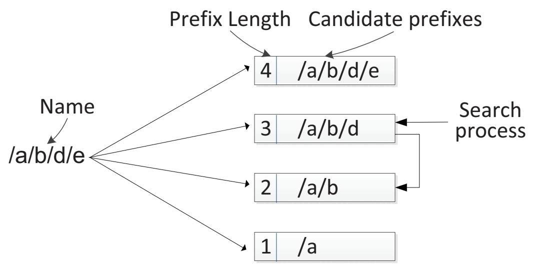

Figure 4: An example of binary search mechanism for LPM.

2.3.3 *Binary search scheme*

With the help of the reconstructed FIB, the lookup engine can search the longest matching prefix in the random mode. The hash-based LPM search process can be abstracted as the classical problem of integer search, e.g., finding the key in a sorted array. One of the most effective algorithm is the binary search algorithm that needs $O(logN)$ probes to find the longest matching prefix on average [19]. Therefore, the lookup engine can reduce the number of probes from $O(N)$ to $O(logN)$ by replacing the linear search scheme with binary search scheme [28].

Figure 4 illustrates an example of binary search scheme for hash-based LPM. The lookup engine generates and sorts the 4 candidate prefixes of the address “/a/b/d/e”. Then the candidate prefix “/a/b/d” with 3 ($\lceil(1 + 4)/2\rceil = 3$) components is checked at first. Given that “/a/b/d” cannot match any prefix in Set-3 of the reconstructed FIB presented in Figure 3, the binary search scheme chooses a shorter candidate prefix “/a/b” with 2 ($\lceil(1 + 3)/2\rceil = 2$) components as the next one to be probed. Since Set-2 contains prefix “/a/b”, the binary search scheme chooses a longer prefix to carry on searching. However, the failure of prefix “/a/b/d” reveals that there is no longer prefix than “/a/b/d” will be matched. Therefore, the lookup engine stops the search process and returns prefix “/a/b” as the longest matching prefix.

3. THE OPTIMAL SEARCH PATH FOR HASH-BASED LONGEST PREFIX MATCH

3.1 The statistical optimal search scheme

The essence of hash-based LPM search process is a search path for finding the longest matching prefix. Compared with the basic

Algorithm 1: Dynamic Programming Algorithm For Optimizing LPM

```
   Input: $N$, $P'_i$
   Output: $C_{ij}$, $L_{ij}$
1  for $i \leftarrow 1$ to $N-1$ do
2      $C_{ii} \leftarrow 1$;
3      $L_{ii} \leftarrow i$;
4  end
5  for $i \leftarrow 1$ to $N$ do
6      for $j \leftarrow 1$ to $N-i$ do
7          for $k \leftarrow j$ to $j+i$ do
8              $C'_{ij} \leftarrow 1 + (1 - P'_i) * C_{i(k-1)}$;
9              $C'_{ij} \leftarrow C'_{ij} + P'_i * C_{(k+1)j}$;
10             if $C'_{ij} < C_{ij}$ then
11                 $C_{ij} \leftarrow C'_{ij}$;
12                 $L_{ij} \leftarrow k$;
13             end
14         end
15     end
16 end
```

linear search scheme described in Section 2.3.1, the binary search scheme effectively reduces the number of probes from $O(N)$ to $O(logN)$ by reconstructing the FIB. However, we are still interested in whether there is an optimal search path for the reconstructed FIB, and how to find this optimal search path.

Since the length distribution of prefixes in the routing table is known in advance, there should be an optimal search path for each address. The number of prefix sets is no greater than N, therefore even in the worst case we can find the optimal search path of an address by applying exhaustive search that has $O(N!)$ time complexity. However, the exhaustive search is an exponential time complexity solution, and we need to design a more effective algorithm to find the optimal search path for practical implementation.

Fortunately, we design and implement a dynamic programming algorithm only with $O(N^3)$ time complexity to find the optimal search path. Assuming that the prefixes in the reconstructed FIB are partitioned into N prefix sets; C_{ij} is the number of probes in the optimal search path from Set_i to Set_j; PRE_k is the candidate prefix with k components; P_i is the matching probability of Set_i in the original FIB; and P'_i is the matching probability of Set_i in the reconstructed FIB. The optimal search path of an address is equivalent to finding the minimal C_{1M}, if $M < N$; or the minimal C_{1N}, if $M \geq N$. Here M is the length of the name or the IP address.

The problem of finding the minimal C_{ij} can be decomposed into two optimization sub-problems: $C_{i(k-1)}$ and $C_{(k+1)j}$. Here, $i \leq k \leq j$; and $C_{ij} = 0$, if $i > j$. As the reconstructed FIB supports random search, we suppose the candidate prefix PRE_k with k components will be checked at first. If PRE_k is dismatched, the length of the longest matching prefix will be less than k, and the lookup engine will search from PRE_1 to PRE_{k-1} to lookup the exact longest matching prefix, i.e., to find the minimal $C_{i(k-1)}$; otherwise PRE_k is matched, the length of the longest matching prefix will be k or greater than k, and the lookup engine will search from PRE_{k+1} to PRE_j to lookup the exact longest matching prefix, i.e., to find the minimal $C_{(k+1)j}$. Therefore, the minimal C_{ij} can be calculated in the following formula:

$$C_{ij} = MIN\{1 + (1 - P'_k) * C_{i(k-1)} + P'_k * C_{(k+1)j}\} \quad (1)$$

Here, $1 \leq i \leq N$, $1 \leq i \leq N$, $i \leq k \leq j$ and $C_{ii} = 1$ according to Formula 1. By storing the solution of sub-problem (e.g., C_{ij}), the optimal search path C_{1N} is solved bottom-up. The dynamic programming algorithm, described in Algorithm 1, reduces the time complexity of finding the optimal search path from $O(N!)$ to $O(N^3)$ compared with the exhaustive search.

To present the dynamic programming algorithm more clearly, Figure 5 illustrates the matrixes of C_{ij} and ℓ_{ij} (prefix length) of the dynamic programming algorithm for finding the optimal search path of the FIB demonstrated in Figure 3.

1. The dynamic programming algorithm recalculates the matching probability P'_i of *Set-i* in the reconstructed FIB based on the matching probability P_i in the original probability. Given that the meta-prefixes of a prefix is inserted into the reconstructed FIB, there is $P'_i = \sum_{k=i}^{N} P_k$. For example, in Figure 5, the original P_i (i from 1 to 4) are {0.1, 0.3, 0.4, 0.2}[5]. Consequently, we can calculate P'_i = {1, 0.9, 0.6, 0.2}.
2. C_{ij} and ℓ_{ij} are initialized, respectively. C_{ii} is set to 1, and the other C_{ij} ($i \neq j$) is set to $N + 1$. ℓ_{ii} is set to i, and the other ℓ_{ij} ($i \neq j$) is set to 0.
3. The C_{ij} and ℓ_{ij} are searched iteratively (line 5 $\sim$ 16 in Algorithm 1) from $j - i = 1$ to $j - i = N - 1$. For example, there are two feasible ways to calculate C_{34}, *Path-1*: $< \ell_{33} \rightarrow \ell_{44} >$ or *Path-2*: $< \ell_{44} \rightarrow \ell_{33} >$. In *Path-1*, the prefix with 4 components (ℓ_{44}) will be probed if and only if the prefix with 3 components (ℓ_{33}) is matched. Therefore, $C_{path1} = 1 + P'_3 * C_{44} = 1.6$. On the other side, in *Path-2*, the prefix with 3 components (ℓ_{33}) will be probed if and only if the prefix with 4 components (ℓ_{44}) is dismatched. Hence, the $C_{path2} = 1 + (1 - P'_4) * C_{33} = 1.8$. Consequently, the *Paht-1* is chosen as the optimal search path, and there are $C_{34} = 1.6$ and $\ell_{34} = 3$.
4. The dynamic programming algorithm stops and returns the matrixes of C_{ij} and ℓ_{ij} to indicate the optimal search path. For an address with M ($M \geq N$) components, the optimal search path is illustrated in Figure 6. At first, the candidate prefix PRE_3 with 3 components is checked. If PRE_3 is matched in *Set-3*, the next prefix to be checked is PRE_4 as $\ell_{44} = 4$; If PRE_3 is dismatched, the next prefix to be checked is PRE_2 as $\ell_{12} = 2$. The lookup engine probes the prefixes one by one until finding the longest matching prefix or all prefixes in the optimal path are checked.

[5]We assume the access probability of each prefix in the original FIB is equal, i.e., $P_i = |Set_i|/N$, here $|Set_i|$ is the number of prefixes in Set_i. The measurement technology of the matching probability is another research problem, and in this paper we focus on finding the optimal search path based on the statistics of the matching probabilities. However, the worst-case upper bound of the average search path of Ω-LPM, derived in Section 3.2, is irrelevant with the statistics of the matching probabilities.

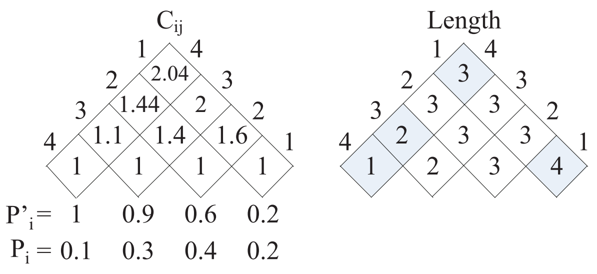

Figure 5: The C_{ij} and ℓ_{ij} matrixes of the dynamic programming algorithm.

The matrix of ℓ_{ij}, indicating the optimal search path of the FIB, can be precalculated as well as the FIB reconstructing process. Besides that, the matrix of ℓ_{ij} will be recalculated along with the FIB's updates. As the time complexity of the dynamical programming algorithm is $O(N^3)$ and N usually is small (In our experiments, N is 32 and 6 in IPv4 and NDN), the matrix of ℓ_{ij} can be recalculated quickly. Note that the dynamic programming algorithm runs in the control plane of a router, and the lookup engine only utilizes the matrix of ℓ_{ij} to indicate the optimal search path of an address.

3.2 The worst-case upper bound of the average search path of statistical optimal LPM

As described in Section 3.1, the matrix C_{ij} indicates the least upper bounds of the average search paths to find the addresses with different number of components , as well as the matrix ℓ_{ij} indicates the exact search paths. The matrixes C_{ij} and ℓ_{ij} can be calculated fast according to the routing table by the Algorithm 1 which has $O(N^3)$ computational complexity, but we are still interesting to find the worst-case upper bound when the search addresses violate the statistics of the matching probability.

Formula 1 demonstrates that C_{ij} is the minimal value in the all paths. If we choose any path, e.g., a candidate prefix PRE_m, the path length $C_{ij}^{'}$ will be greater than or equal to C_{ij}:

$$\begin{aligned} C_{ij} &= MIN\{1 + (1 - P'_k) * C_{i(k-1)} + P'_k * C_{(k+1)j}\} \\ &\le 1 + (1 - P'_m) * C_{i(m-1)} + P'_m * C_{(m+1)j} \end{aligned} \quad (2)$$

If $C_{i(m-1)} \ge C_{(m+1)j}$, we get:

$$\begin{aligned} C_{ij}^{'} &= 1 + (1 - P'_m) * C_{i(m-1)} + P'_m * C_{(m+1)j} \\ &= 1 + C_{i(m-1)} + P'_m * (C_{(m+1)j} - C_{i(m-1)}) \\ &\le 1 + C_{i(m-1)} \end{aligned} \quad (3)$$

Else if $C_{i(m-1)} \le C_{(m+1)j}$, we get:

$$\begin{aligned} C_{ij}^{'} &= 1 + (1 - P'_m) * C_{i(m-1)} + P'_m * C_{(m+1)j} \\ &= 1 + C_{(m+1)j} + (1 - P'_m) * (C_{i(m-1)} - C_{(m+1)j}) \\ &\le 1 + C_{(m+1)j} \end{aligned} \quad (4)$$

Here, we choose $m = \lfloor (i+j)/2 \rfloor$. Therefore, $C_{ij}^{'}$ can be calculated recursively according to Formula 3 or Formula 4. Formally, the recursive process of calculating $C_{ij}^{'}$ can be represented by Formula 5.

$$T(n) = T(\lfloor \frac{n}{2} \rfloor) + 1 \quad (5)$$

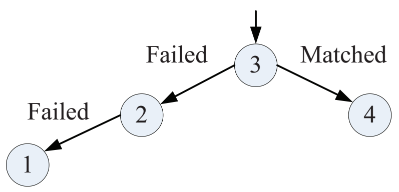

Figure 6: The optimal search path of an address with more than N components.

Here, n is the length of an address, i.e., $n = j - i + 1$; and $T(\cdot)$ is a function. According to the Master Theorem [10], we can get:

$$C_{ij}^{'} \le T(n) \le 1 + \log_2(\lfloor \frac{n}{2} \rfloor * 2) \le 1 + \log_2(n) \quad (6)$$

Formula 6 can be proofed via applying the mathematical induction, and the proof is depicted below.

PROOF. Since $C_{ij} = 1$ when $i = j$, there is $T(1) = 1$. The process of the mathematical induction is that:

1. When $n = 1$, there are $T(1)$=C_{ii}=1 and $1 + \log_2(1)$=1, so $T(1) \le 1 + \log_2(1)$;
2. Assuming $T(n) \le 1 + \log_2(n)$ is correct, then
3. we get:

$$\begin{aligned} T(2n) &= T(\lfloor \tfrac{2n}{2} \rfloor) + 1 \\ &= T(n) + 1 \\ &\le 1 + 1 + \log_2(n) \\ &\le 1 + \log_2(2n) \end{aligned}$$

Or

$$\begin{aligned} T(2n+1) &= T(\lfloor \tfrac{2n+1}{2} \rfloor) + 1 \\ &= T(n) + 1 \\ &\le 1 + 1 + \log_2(n) \\ &\le 1 + \log_2(2n) \\ &\le 1 + \log_2(2n+1) \end{aligned}$$

4. Therefore, the inequality $T(n) \le 1 + \log_2(n)$ is proofed to be correct.

□

Since $C_{ij} < C_{ij}^{'}$, we get $C_{ij} < 1 + \log_2(n)$, i.e., the upper bound of C_{ij} is $1 + \log_2(n)$, here $n = j - i + 1$.

4. CASE STUDIES

4.1 Case study 1: Name Lookup

Named Data Networking (NDN) was proposed to embrace Internet's functionality transition: from host-centric communication to content-centric information dissemination. Different from IP-based network routers, NDN routers forward packets by content names, which have variable and unbounded length. Further, an NDN name routing table could be several orders of magnitude larger than a current IP routing table. This kind of complex name constitution plus the huge-sized name routing table makes wire speed NDN name lookup an extremely challenging task. Practical name lookup mechanism design and implementation, therefore, requires substantial innovation and re-engineering.

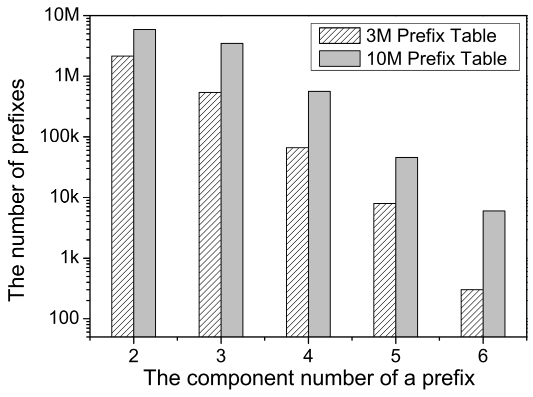

Figure 7: The distribution of name prefixes in the name tables.

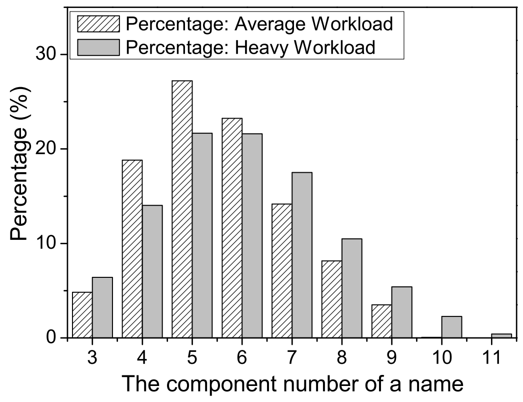

Figure 8: The percentages of names with different component numbers in the traces.

Hash-based algorithms for name lookup perform high lookup speed, low memory consumption, and good scalability [22, 23, 25]. However, the search paths of the earlier works on hash-based name lookup are not optimized and still have space to improve. Therefore, in this case study, we apply the dynamic programming algorithm (introduced in Section 3.1) to calculate the optimal search path to reduce the number of probes and speed up the lookup process.

Prefix tables and name traces: Both prefix tables and name traces used in our experiments are downloaded from the webpage [6]. The two prefix tables, "3M prefix table" and "10M prefix table", contain 2,544,794 entries and 9,552,363 entries, respectively. Each prefix table entry is composed of an NDN-style name and a next-hop port number. The distribution of the number of components and the average length of prefixes are shown in Figure 7.

The name traces simulate the destination names carried in NDN packets. There are two types of name traces, simulating average lookup workload and heavy lookup workload, respectively. Each name trace contains 200M names. The percentage of names with different number of components and the average length of names in each trace are illustrated in Figure 8.

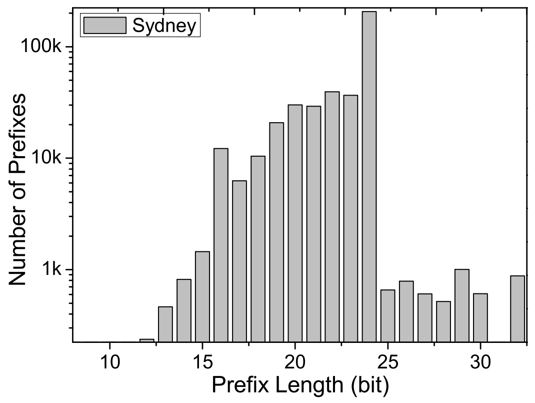

Figure 9: The distribution of IP prefixes in the Sydney.

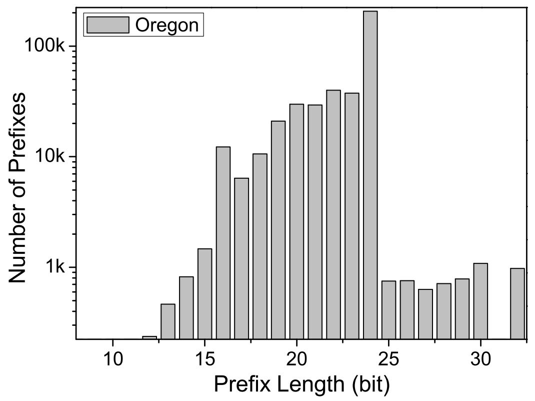

Figure 10: The distribution of IP prefixes in the Oregen.

4.2 Case study 2: IP Lookup

IP lookup is a basic but important function for a IP router. A lot of works on IP lookup including hardware circuits (e.g., TCAM), architectures, algorithms, and data structures, have been done to improve the IP lookup performance. Hash-based algorithms [19, 12, 27] are one of the main branches of IP lookup. Waldvogel *et al.* propose a binary search scheme to shorten the search path and improve the lookup performance. But the binary search scheme cannot obtain the optimal search path. Therefore, to further improve the lookup performance, we apply our programming algorithm on IP lookup to get the optimal search path.

IP tables and IP traces: The IP tables (named Sydney and Oregon) used in our experiments are downloaded from the RIPE [7] and Oregon [8], respectively. Sydney contains 400,010 IP prefixes and Oregon contains 402,411 IP prefixes. The prefix distributions of Sydney and Oregon are illustrated in Figure 9 and Figure 10.

The IP trace used to test the performance is downloaded from the router of Chicago in CAIDA [4]. The IP trace sustains 5 minutes and contains 91,568,588 packets.

Table 1: The matrix C_{ij} of the 3M prefix table.

i\j	*6*	*5*	*4*	*3*	*2*	*1*
1	2.005991	2.005974	2.000000	1.778248	1.000000	1.000000
2	2.005991	2.005974	2.000000	1.778248	1.000000	
3	1.227743	1.227725	1.221752	1.000000		
4	1.027018	1.026938	1.000000			
5	1.002989	1.000000				
6	1.000000					

Table 2: The matrix ℓ_{ij} of the 3M prefix table.

i\j	*6*	*5*	*4*	*3*	*2*	*1*
1	3	3	3	3	2	1
2	3	3	3	3	2	
3	3	3	3	3		
4	4	4	4			
5	5	5				
6	6					

Table 3: The matrix C_{ij} of the 10M prefix table.

i\j	*6*	*5*	*4*	*3*	*2*	*1*
1	2.025358	2.025229	2.000000	1.589941	1.000000	1.000000
2	2.025358	2.025229	2.000000	1.589941	1.000000	
3	1.435417	1.435288	1.410059	1.000000		
4	1.061841	1.061525	1.000000			
5	1.005141	1.000000				
6	1.000000					

Table 4: The matrix ℓ_{ij} of the 10M prefix table.

i\j	*6*	*5*	*4*	*3*	*2*	*1*
1	4	3	3	3	2	1
2	4	3	3	3	2	
3	4	3	3	3		
4	4	4	4			
5	5	5				
6	6					

5. EXPERIMENTAL RESULTS

In this section, we evaluate the lookup performance improved by optimizing the search path of hash-based LPM. Totally 4 methods are implemented: the basic method used in the literatures [22, 12] that probes all the sets to find the longest matching prefix; the optimal linear search scheme introduced in Section 2.3.1; the binary search scheme used by Waldvogel *et al.* [19]; and the Ω-LMP proposed by us.

5.1 Experimental results of name lookup

5.1.1 Experimental Setup

The name lookup engine is implemented and run on a commodity PC with two 6-core CPUs. The PC runs Linux Operating System in the version 2.6.41.9-1.fc15.x86_64. The entire program consists of about 2,800 lines of code, developed by C++ programming language. The part of multi-core parallel processing is developed using OpenMP API [1] in the version 2.5.

We combine the NameFilter [22] with the 3 LPM search schemes introduced in Section 3 to increase the name lookup speed in NDN. Totally 4 methods are implemented. The baseline is the original NameFilter, which is improved by applying the basic linear search path to shorten the search path (Named NF-Linear). After reconstructing the data structure of the FIB, the binary search scheme [28] is used to further shorten the search path (Named NF-Binary). Finally, the optimal search scheme is used to generate the shortest search path (Named NF-Optimal).

5.1.2 Search path length

First of all, the lookup engine calculates the optimal search paths of the prefix tables according to the Algorithm 1. The matrixes C_{ij}, storing the lengths of the optimal search paths of the 3M prefix table and the 10M prefix table, are listed in the Table 1 and Table 3, respectively. And the matrixes ℓ_{ij}, containing the indications of the optimal search paths of the 3M and the 10M prefix tables, are presented in the Table 2 and Table 4, respectively. In the average case, the lengths of the optimal search paths on the 3M and the 10M prefix tables for the name with more than 6 components are 2.0061 and 2.0254. As illustrated in Figure 7, the maximal length prefixes in the 3M prefix table and the 10M prefix table consist of 6 components. Therefore, the worst-case upper bound of the search path of the statistical optimal LPM is 3.585.

Then, we compare the search path lengths of the 4 methods conducting on the 3M and 10M prefix tables with the average and heavy workloads. The experimental results are shown in Table 5. The basic method probes all candidate prefixes of a name to find the longest matching prefix, so on average it probes 5.21 and 5.30 prefixes against 3M prefix table under average workload and heavy workload, respectively. By checking the prefixes from the longest one to the shortest, the optimal linear method effectively reduces the number of probes down to 3. The binary search method, probing 3.27 prefixes on average, has longer search path length than the linear method, as a name in the traces at most has 11 components and the most of names are shorter than 9. The final scheme, Ω-LPM, only needs to check 2.03 prefixes on average to find the longest matching prefix, which shortens 61.04%, 32.33%, and 37.91% search paths of the basic method, the linear method and the binary method, respectively. Meanwhile, the experimental results on 10M prefix table show the same conclusion.

Beyond that, we are still interested in foreseeing the performance trend as prefix table size grows. Toward this endčňbased on 10M prefix table, we generate ten prefix tables with 1 $\sim$ 10 million prefixes, respectively. The experiments are then conducted on these ten prefix tables. Figure 11 illustrates the measured results of the search path length under average workload. The search path length of the binary method depends on the specific trace, and the length jitter of NF-Binary is larger than the other 3 methods. Whatever, Ω-LPM has the shortest search path length. The better experimental phenomenon shown in Figure 11 is that the search path length of Ω-LPM decreases gradually as the prefix table size grows. Consequently, Ω-LPM has good scalability and is suitable for larger prefix tables.

5.1.3 Memory space

The FIB is reconstructed to support random search by inserting

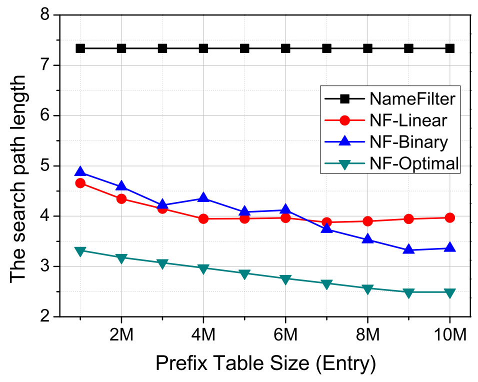

Figure 11: The scalability of search path length.

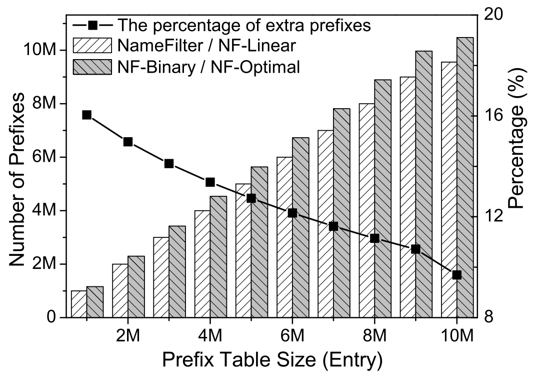

Figure 12: The scalability of memory space and extra memory percentage.

Table 5: The average lengths of the search paths with different methods.

Prefix Table	Trace	The average search path length			
		Basic Method	Linear	Binary Search	Ω-LPM
3M	Average	5.21	3.00	3.27	2.03
3M	Heavy	5.30	3.90	4.19	2.31
10M	Average	5.33	3.76	3.36	2.49
10M	Heavy	5.41	3.95	4.81	2.77
Sydney	Chicago	25	12.47	4.03	3.29
Oregon	Chicago	25	12.40	4.02	3.28

Table 6: The number of prefixes of different methods on the name prefix tables and IP prefix tables

Prefix Table	The number of prefixes	
	Basic Method / Linear	Binary Search / Ω-LPM
3M	2,544,794	2,806,907
10M	9,552,363	10,478,119
Sydney	400,010	971,593
Oregon	402,411	981,457

the meta-prefixes of each prefix to the FIB. Obviously, the reconstructed FIB has more prefixes than the original one. Fortunately, the extra memory space is acceptable according to the measurement results demonstrated in Table 6. On 10M prefix table, the original FIB contains 9,552,363 prefixes, and the reconstructed FIB has 10,478,119 prefixes which costs extra 9.69% memory.

Figure 12 illustrates the performance trend of memory space and the extra memory percentage as the prefix table size grows. It is consistent with our expectation that the numbers of prefixes of different methods gradually increase along with the growing of the prefix table size. However, the percentage of extra prefixes caused by reconstructing the FIB decreases gradually, as the most of meta-prefixes of the incremental prefixes have been inserted into the reconstructed FIB. With the scale of prefix table expanding, less and less meta-prefixes need to be inserted into the reconstructed FIB. Therefore, we can predict that the extra memory overhead can be negligible on a FIB with tens, or even hundreds million of prefixes.

5.1.4 Lookup speed

We first compare the lookup speed of the 4 methods in the single thread work mode. The experimental results are listed in Table 7. Both experimental results conducted on 3M prefix table and 10M prefix table demonstrate that NF-Optimal can effectively improve the name lookup speed. On 10M prefix table, NF-Optimal achieves 2.03 MSPS (Million Searches Per Second) and 1.88 MSPS under average workload and heavy workload, respectively. Compared with NameFilter, NF-Optimal achieves 1.43× speedup.

Table 8 further presents the lookup speed of 4 methods running with 24 threads. Under average workload, NF-Optimal achieves 35.75 MSPS and 34.82 MSPS on 3M prefix table and 10M prefix table, respectively, which is 16× speedup compared with the performance conducted in the single thread work mode. Assuming the average packet length in NDN is 250 bytes, the name lookup engine, applying NF-Optimal mechanism, can achieve about 70Gbps. If the transmission speed of an interface card equipped on a software router is 10Gbps, this kind of name lookup engine can deal with 7 interface cards' traffic at the same time.

The name lookup throughput of NameFilter and NF-Optimal, as demonstrated in Figure 13, tend to stabilize on average workload along with the prefix table size grows, while the lookup speeds of NF-Linear and NF-Binary gradually decrease along with the prefix table size grows. NF-Optimal is more stable than NF-Linear and NF-Binary, since optimal search path is dynamically adjusted according to the FIB in real time. Another reason is that the optimal search path is not only effective for the name with more than N components, but also effective for the short names with less than N components. In summary NF-Optimal has good scalability of lookup speed.

5.1.5 Update

To support random search, the meta-prefixes of a prefix in the original FIB should be inserted into the reconstructed FIB. Therefore, the update of the reconstructed FIB is more complex than the update of the original FIB. 1) To insert a prefix into the reconstructed FIB, all its meta-prefixes should be generated and inserted into the reconstructed FIB. Each meta-prefix needs to be searched in the FIB to check whether it has been inserted into the FIB. Only

Table 7: The lookup speed of different methods with 1 work thread.

Prefix Table	Trace	The lookup speed (MSPS)			
		NameFilter	NF-Linear	NF-Binary	NF-Optimal
3M	Average	1.68	2.04	1.99	2.13
3M	Heavy	1.49	1.90	1.84	1.93
10M	Average	1.42	1.78	1.91	2.03
10M	Heavy	1.33	1.84	1.66	1.88

Table 8: The lookup speed of different methods with 24 work threads.

Prefix Table	Trace	The lookup speed (MSPS)			
		NameFilter	NF-Linear	NF-Binary	NF-Optimal
3M	Average	26.49	31.67	36.14	35.75
3M	Heavy	25.22	28.12	29.89	32.48
10M	Average	24.88	25.76	30.45	34.82
10M	Heavy	22.34	27.89	28.65	31.22

the new sub-prefix will be inserted into the FIB. 2) To delete a prefix, only the prefix itself will be removed from the reconstructed FIB, since the sub-prefixes of the deleted prefix may be the sub-prefixes of other prefixes.

The update performance of the 4 methods are illustrated in Figure 14. Given NameFilter only needs one exact match operation to insert a prefix or delete a prefix, its insertion/deletion throughput is higher than NF-Optimal. NF-Optimal can still achieve 0.9 million insertions per second or 1.0 million deletions per second. Meanwhile, NF-Optimal trends stable along with the prefix table size grows.

5.2 Experimental results of IP lookup

5.2.1 *Search path length*

The average search path lengths of the 4 methods on the IP tables of Sydney and Oregon under the Chicago trace are listed in Table 5. The basic method probes all possible candidate IP prefixes to find the longest matching one, therefore it needs 25 probes (The length of an IP prefix must be greater than 7). On average, the optimal linear search scheme probes 12.47 times and 12.40 times on the IP tables of Sydney and Oregon, respectively. The binary search method achieves better performance than the linear scheme, it only needs around 4 probes. Ω-LPM further improves the lookup performance. It only needs to check 2.38 prefixes on average to find the longest matching prefix. Consequently, on the IP tables, Ω-LPM shortens 86.88%, 73.55%, and 18.41% search paths of the basic method, the linear method and the binary method, respectively.

5.2.2 *Memory space*

Given the IP prefixes has 25 prefix sets, the reconstructed FIBs of IP tables cost more memory than the reconstructed FIBs of name tables. The measurement results of the reconstructed IP FIBs are

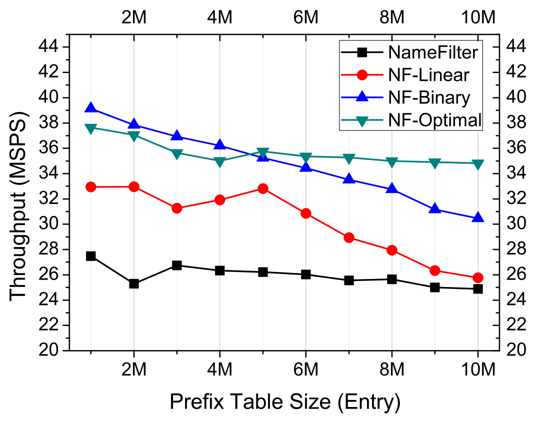

Figure 13: The scalability of throughput with average workload.

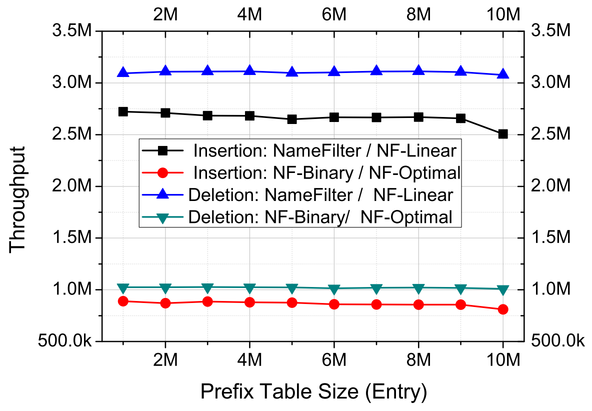

Figure 14: The update and deletion performance.

demonstrated in Table 6. The reconstructed IP FIBs of Sydney and Oregon are 2.42× and 2.43× times larger than the original FIBs. Given the number of IP prefixes in the reconstructed FIB is equal to the number of edges in the trie that organizes the FIB, the memory cost is acceptable for the real system.

6. RELATED WORK

As a basic function of a router, various LPM algorithms have been studied for more than 20 years.

TCAM-based Algorithms: Ternary content addressable memory (TCAM), as a fully associative memory which stores wildcards (don't care states) in a memory cell in addition to 0s and 1s compared with conventional binary memory, matches the data in parallel by providing a comparator in each cell. Despite TCAM returns the search result within a single memory access and achieves high speed lookup, the high cost and the large power consumption of TCAM make it unattractive for high-end routers. Numerous TCAM-based approaches [30, 32, 18] are proposed to reduce the power consumption, but TCAM's range of use is greatly limited because of its disadvantages that stem from the small capacity and the word-width.

Trie-based Algorithms: Trie [14] structure has been widely used to construct the forwarding table in IP network to address the LPM problem. PATRICIA trie algorithm [17], first used in file system and then used in BSD kernel [3], applies the path-compressed trie structure to reduce the memory accesses of the original trie. Lulea algorithm [11] further speeds up the lookup speed by utilizing Bitmap technology to compress the forwarding table which is small enough to fit in the cache of a conventional general purpose processor. Multibit-tire architectures, such as Tree Bitmap [13], can improve the lookup speed as one memory access will consume multiple bits in the IP address. Since one lookup in Tree Bitmap still requires multiple off-chip memory accesses, FlashTrie [9] overcomes the shortcomings of the multibit-trie based approaches by using a hash-based membership query to limit off-chip memory accesses per lookup and to balance memory utilization among the memory modules.

However, the trie-based approaches are appropriate for the short and fixed-length naming mechanism, such as IPv4/IPv6 addresses. For the URL-similar naming mechanism, such as NDN names, the lookup speed will dramatically slow down and the memory consumption will greatly inflate because of the high depth of the trie [21, 20, 26].

Hash-based Algorithms: Hash-based approaches appear to be an excellent candidate for LPM with the possibility of fast lookup speed and low latency [25, 22, 24]. DIR-21-3-8 algorithm [15] proposed by Gupta *et al.* can achieve one route lookup every memory access by implementing in a pipelined fashion in hardware. In this scheme, the IP prefixes in the FIB are “leaf push”[6] and partitioned into three sets: a 21-bit prefix set, a 24-bit prefix set, and a 32-bit prefix set, and therefore only three memory accesses are needed in the worst case. However, DIR-21-3-8 algorithm, based on “leaf push” technology, is useless for the NDN FIB with URL-similar prefixes, since the granularity of a URL-similar prefix is a component not a bit and the NDN FIB cannot support for “leaf push” consequently.

Sarang*et al.* [12] proposed the first algorithm that employs Bloom filters for LPM. This algorithm performs parallel queries on Bloom filters in order to determine address prefix membership in sets of prefixes sorted by prefix length. Similarly, NameFilter [22] exploits Bloom filters to determine the longest matching prefix of a searched name in the first stage. Compared with Sarang’s algorithm, NameFilter’s Bloom filters in the first stage are too large to fit in the on-chip memory, hence the look speed of NameFilter is dramatically decreased even using one memory access Bloom filters to organize the prefix sets.

CCNx [5], as the current prototype implementation of CCN [16], achieves LPM based on the linear search scheme introduced in Section 2.3.1. Given a hierarchical name with a number of components, CCNx retrieves all possible prefixes from the name, conducts exact-match search of the prefixes, from the longest one to the shortest one, in the FIB, and stops when the first match is found. The overall forwarding performance, however, is at least an order of magnitude below wire speed [29].

[6]“Leaf push” means that a prefix node is pushed down to its children nodes for keeping the same length with other prefixes. For example, a 3-bit length IP prefix *111** can be “leaf push” to *1110* and *1111* for 4-bit length prefixes.

In summary, the aforementioned hash-based LPM algorithms cannot support for random search. On the contrary, Waldvogel *et al.* [19] and Patrick *et al.* [28] improve the lookup performance by reconstructing the original FIB to support random search. Beyond that, Ω-LPM designs a dynamic programming algorithm to find the optimal search path to further increase the lookup speed.

7. CONCLUSION

In this paper, we proposed the Ω-LPM scheme to improve the lookup performance of hash-based LPM. By inserting the meta-prefixes of the original prefix into the FIB, Ω-LPM can support for random search which is the foundation for optimizing the search path. Fortunately, the number of prefixes including the meta-prefixes and the original prefixes in the reconstructed FIB is equal to the number of edges in a trie which is used to construct the original FIB. To further increase the lookup speed, Ω-LPM employs a dynamic programming algorithm to find the optimal search path to reduce the number of hash table probes. The case studies of the name lookup and the IP lookup demonstrate that Ω-LPM can effectively shorten the search path. The experimental results also show the Ω-LPM has good performance of memory consumption and scalability.

8. ACKNOWLEDGMENTS

The work is supported by the Shenzhen Key Fundamental Research Projects (No. JCYJ20160330095313861), Huawei Innovation Research Program (HIRP), NSFC (61602271, 61373143, 61432009), China Postdoctoral Science Foundation (No. 2015T80089, 2016M591182), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20130002110084).

9. REFERENCES

[1] The openmp api specification for parallel programming. http://openmp.org/wp/.

[2] RFC791, Internet Protocol − DARPA Internet Program Protocol Specification (September 1981).

[3] Berkeley Software Distribution. http://en.wikipedia.org/wiki/Berkeley_Software_Distribution, 2014. [Online].

[4] CAIDA Anonymized Internet Trace. http://www.caida.org/data/monitors/passive-equinix-sanjose.xml, 2014. [Online].

[5] CCNx project. http://www.ccnx.org/, 2014. [Online].

[6] Name Lookup Project. http://s-router.cs.tsinghua.edu.cn/namelookup.org/index.htm, 2014. [Online].

[7] RIPE. http://www.ripe.net/, 2014. [Online].

[8] University of Oregon Route Views Archive Project. http://archive.routeviews.org/, 2014. [Online].

[9] M. Bando and H. Chao. FlashTrie: Hash-based Prefix-Compressed Trie for IP Route Lookup Beyond 100Gbps. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–9, 2010.

[10] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms, Second Edition*. MIT Press and

McGrawHill, 2001.

[11] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink. Small forwarding tables for fast routing lookups. In *SIGCOMM'97*, pages 3–14, 1997.

[12] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor. Longest prefix matching using bloom filters. *IEEE/ACM Transactions on Networking*, 14(2):397–409, 2006.

[13] W. Eatherton, G. Varghese, and Z. Dittia. Tree bitmap: hardware/software ip lookups with incremental updates. *SIGCOMM Comput. Commun. Rev.*, 34(2):97–122, Apr. 2004.

[14] E. Fredkin. Trie memory. *Commun. ACM*, 3(9):490–499, Sept. 1960.

[15] P. Gupta, S. Lin, and N. McKeown. Routing lookups in hardware at memory access speeds. In *INFOCOM'98*, pages 1240–1247, 1998.

[16] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard. Networking named content. In *CoNEXT'09*, pages 1–12. ACM, 2009.

[17] D. R. Morrison. PATRICIA - Practical Algorithm To Retrieve Information Coded in Alphanumeric. *J. ACM*, 15(4):514–534, Oct. 1968.

[18] V. C. Ravikumar, R. Mahapatra, and L. Bhuyan. EaseCAM: an energy and storage efficient TCAM-based router architecture for IP lookup. *Computers, IEEE Transactions on*, 54(5):521–533, 2005.

[19] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner. Scalable high speed ip routing lookups. In *SIGCOMM'97*, pages 25–36, 1997.

[20] Y. Wang, H. Dai, J. Jiang, K. He, W. Meng, and B. Liu. Parallel name lookup for named data networking. In *IEEE Global Telecommunications Conference (GLOBECOM)*, pages 1 –5, dec. 2011.

[21] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen. Scalable name lookup in ndn using effective name component encoding. In *IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, pages 688–697, june 2012.

[22] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong. NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters. In *Infocom mini-conference'13*, 2013.

[23] Y. Wang, D. Tai, T. Zhang, J. Lu, B. Xu, H. Dai, and B. Liu. Greedy name lookup for named data networking. In *ACM SIGMETRICS*, pages 359–360. ACM, 2013.

[24] Y. Wang, D. Tai, T. Zhang, J. Lu, B. Xu, H. Dai, and B. Liu. Greedy name lookup for named data networking. In *Proceedings of the ACM SIGMETRICS/International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '13, pages 359–360, New York, NY, USA, 2013. ACM.

[25] Y. Wang, B. Xu, D. Tai, J. Lu, T. Zhang, H. Dai, B. Zhang, and B. Liu. Fast Name Lookup for Named Data Networking. In *IWQoS*. ACM, 2014.

[26] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, et al. Wire speed name lookup: A gpu-based approach. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 199–212, 2013.

[27] H. Yu, R. Mahapatra, and L. Bhuyan. A hash-based scalable IP lookup using Bloom and fingerprint filters. In *ICNP*, pages 264–273. IEEE, 2009.

[28] H. Yuan and P. Crowley. Reliably scalable name prefix lookup. In *Architectures for Networking and Communications Systems (ANCS), 2015 ACM/IEEE Symposium on*, pages 111–121. IEEE, 2015.

[29] H. Yuan, T. Song, and P. Crowley. Scalable ndn forwarding: Concepts, issues and principles. In *International Conference on Computer Communications and Networks (ICCCN)*, pages 1–9, 2012.

[30] F. Zane, G. Narlikar, and A. Basu. CoolCAMs: Power-Efficient TCAMs for Forwarding Engines. In *INFOCOM'03*, pages 42–52, 2003.

[31] L. Zhang, D. Estrin, V. Jacobson, and B. Zhang. Named Data Networking (NDN) Project. http://www.named-data.net/, 2014. [Online].

[32] K. Zheng, C. Hu, H. Lu, and B. Liu. A TCAM-based distributed parallel IP lookup scheme and performance analysis. *Networking, IEEE/ACM Transactions on*, 14(4):863–875, 2006.