NameFilter: Achieving fast name lookup with low memory cost via applying two-stage Bloom filters

Yi Wang, Tian Pan, Zhian Mi, Huichen Dai, Xiaoyu Guo, Ting Zhang, Bin Liu[†]
Tsinghua National Laboratory for Information Science and Technology,
Department of Computer Science and Technology, Tsinghua University, Beijing, China, 100084
{yiwang09, pant09, mza11, dhc10, gxy10, zt10}@mails.tsinghua.edu.cn, liub@tsinghua.edu.cn

Qunfeng Dong
USTC
Hefei, China, 230027
qunfeng@ustc.edu.cn

Abstract—**In this paper we design, implement and evaluate NameFilter, a two-stage Bloom filter-based scheme for Named Data Networking name lookup, in which the first stage determines the length of a name prefix, and the second stage looks up the prefix in a narrowed group of Bloom filters based on the results from the first stage. Moreover, we optimize the hash value calculation of name strings, as well as the data structure to store multiple Bloom filters, which significantly reduces the memory access times compared with that of non-optimized Bloom filters. We conduct extensive experiments on a commodity server to test NameFilter's throughput, memory occupation, name update as well as scalability. Evaluation results on a name prefix table with 10M entries show that our proposed scheme achieves lookup throughput of 37 million searches per second at low memory cost of only 234.27 MB, which means 12 times speedup and 77% memory savings compared to the traditional character trie structure. The results also demonstrate that NameFilter can achieve 3M per second incremental updates and exhibit good scalability to large-scale prefix tables.**

I. Introduction

The research community have started shifting their focus to Content-Centric Networking (CCN) [1], a clean slate future Internet architecture using names instead of IP addresses to identify and locate contents. One of the representative CCN proposals is the Named Data Networking (NDN) [2] project, which has recently attracted significant attention. NDN aims to develop a new Internet architecture that can naturally accommodate emerging communication patterns, and comes with potential for a wide range of benefits such as content caching to reduce congestion and improve delivery performance, better support for multicast and mobile communications, etc.

By naming data objects instead of addressing devices as in IP networks, an NDN router routes and forwards packets by using multi-component URL-similar names rather than IP addresses. This brings great challenges of name-based lookup: 1) Name lookup against Forwarding Information Base (FIB) is longest prefix match. Unlike fixed length IP addresses, NDN names have variable and unbounded length. One search of name lookup needs to scan tens or even hundreds of characters before finding the exactly matched prefix. 2) FIB can be several orders of magnitude larger than IP forwarding tables in today's Internet routers, in terms of the number of table entries (i.e., name/IP prefixes); 3) NDN routers need to handle route updates due to content publishing/deletion, topology as well as policy changes. This makes name prefix table update much more frequent than in today's Internet IP routing tables.

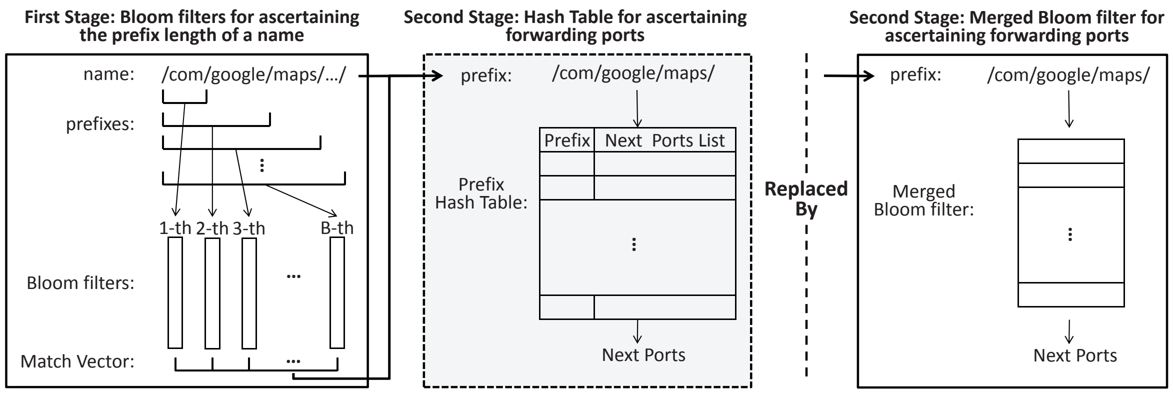
To conquer the above challenges, we investigate constructing efficient name lookup engines using Bloom filter [3], which has the following two advantages: 1) *High memory efficiency.* Bloom filter can significantly reduce memory consumption, compared with storing all the name prefixes directly (like the trie-based structure). 2) *Fast querying speed.* Bloom filter accurately locates stored information via fast hash calculations, which makes the lookup process much faster.

Our experiments revealed that simply applying Bloom filter to longest name prefix match cannot satisfy the ever-increasing demand for lookup speed. Therefore, we propose *NameFilter*, a two-stage Bloom filter-based name lookup approach. In the first stage, name prefixes are mapped into Bloom filters based on their lengths, and the longest prefix of a name is determined by inquiring the Bloom filters at this stage. The second stage divides name prefixes into groups according to their associated next-hop port(s), with each group stored into a Bloom filter. The destination port(s) corresponding to that particular longest prefix will be reported by searching the second stage Bloom filters. We optimize the Bloom filters in both stages according to their actual operations. What we adopt in the first stage is called One Memory Access Bloom filter, which reduces a query's memory access times from k (k is the number of hash functions in each Bloom filter) to one. While in the second stage, we use Merged Bloom filter to decrease the memory access times by a factor of N (N is the amount of ports). Moreover, the hash functions are improved for character strings to reduce the time complexity.

Experiments on a commodity PC server with 10M name prefix entries show that NameFilter achieves 37 million searches per second (MSPS) using only 234.27 MB memory, which is 12× speedup and 4× memory saving over the classic character-trie method [4]. Meanwhile, our proposed scheme is scalable to larger prefix tables and supports 3M/3.4M insertions/deletions per second.

[†]Corresponding author: liub@tsinghua.edu.cn.

This paper is supported by 863 project (2013AA013502), NSFC (61073171, 61073184), Tsinghua University Initiative Scientific Research Program(20121080068), the Specialized Research Fund for the Doctoral Program of Higher Education of China(20100002110051), the Ministry of Education (MOE) Program for New Century Excellent Talents (NCET) in University, the Science and Technological Fund of Anhui Province for Outstanding Youth (10040606Y05), by the Fundamental Research Funds for the Central Universities (WK0110000007, WK0110000019), and Jiangsu Provincial Science Foundation (BK2011360).



First Stage: Bloom filters for ascertaining the prefix length of a name
name: /com/google/maps/…/
prefixes:
1-th 2-th 3-th B-th
Bloom filters:
Match Vector:
Second Stage: Hash Table for ascertaining forwarding ports
prefix: /com/google/maps/
Prefix Hash Table:
Prefix
Next Ports List
Next Ports
Replaced By
Second Stage: Merged Bloom filter for ascertaining forwarding ports
prefix: /com/google/maps/
Merged Bloom filter:
Next Ports

Fig. 1. The framework of NameFilter: replace hash table by Merged Bloom filter.

II. Challenges of Applying Standard Bloom Filter and Hash Table for Name Lookup

Bloom filter assisted IP lookup was proposed by Sarang [5], where IP prefixes of the same length are stored into the same Bloom filter. IP addresses are thus classified into different subsets according to their prefix length, and each subset is organized has a hash table. In the first stage, the Bloom filter corresponding to each prefix length is queried to determine whether there is a matching IP prefix of that length (for the IP address being looked up). If yes, during the second stage, exact match is performed within that subset of IP prefixes using hash table.

As name lookup is essentially longest prefix match (based on characters), Sarang's Bloom filter assisted lookup method can also be applied to name lookup, with IP prefixes replaced by name prefixes. However, straightforward transplantation may lead to the following issues:

1) Suppose we use chaining to resolve hash collisions in the second stage The hash table stores the first name prefix corresponding to the hash value with other conflicted name prefixes chained downwards. Apparently, using this naive method to organize name prefixes leads to performance bottleneck in both lookup speed and memory occupation.
    a) Lookup speed: The lookup process sequentially scans through the hash chain until a match is found; the average number of sequential scans is $1 + \alpha/2$ when chaining is used to resolve the hash collisions (α is the load factor of the hash table) [6]. Calculating the hash value of a name prefix takes $O(w_s)$ to scan the entire name string, where w_s is the average length of the name strings. Hence, the average time complexity of searching a name prefix in the hash table is $O(w_s(1 + \alpha/2))$.
    b) Memory occupation: Each name prefix entry in the hash chains stores entire name prefix, forwarding port list and a pointer to the next entry; the total memory occupation is relatively large. Meanwhile, to decrease the hash collision rate and improve the lookup speed, the load factor should be limited, which results in an even larger memory occupation.
2) Unlike fixed length IP addresses, name prefixes have a variable number of components, which means the number of Bloom filters is undetermined. It is difficult for hardware (FPGA or ASIC chips) to dynamically allocate Bloom filters to support variable length name prefixes. In general-purpose processor, software-oriented implementation may serialize operations on the Bloom filters, which producees multiple sequential memory accesses and degrades system performance.
3) It is much more complicated to calculate the hash values of name prefixes than calculating the hash values of IPv4 address. To obtain the hash value of a 32-bit IPv4 address, only one computation and one memory access with the time complexity of $O(1)$ are involved; to obtain the hash value of a name prefix, we have to traverse the entire name string with the time complexity of $O(w_s)$.

III. NameFilter: Algorithm and Data Structure

A. Replace hash table with Bloom filters

Using hash table in the second stage as described in section II to precisely search and determine the forwarding ports leads to large overhead in either processing time or memory, which fails to meet the requirement for wire-speed processing. We need to find out a more efficient solution for name lookup.

Compared with hash table, Bloom filter is another way to represent name sets and actually a generalized form of hashing with trade-offs between memory consumption and collision probability (false positive)[1]. Bloom filter allocates a relatively small number of bits to per set element with a bounded false positive probability, decreasing the memory consumption.

Bloom filters could be used as classifiers for name prefixes to the same outgoing ports. Therefore, the number of required Bloom filters in the second stage is equal to the number of router's ports. Suppose a router has N ports and each corresponding Bloom filter requires k hash functions to calculate the final forwarding destinations. There're totally kN memory accesses for one query, which degrades lookup speed tremendously. Therefore, we need an innovative Bloom filter structure to reduce memory access and support parallel filtering.

[1]In NDN, Pending Interest Table can eliminate the Interest packet loop caused by false positive.

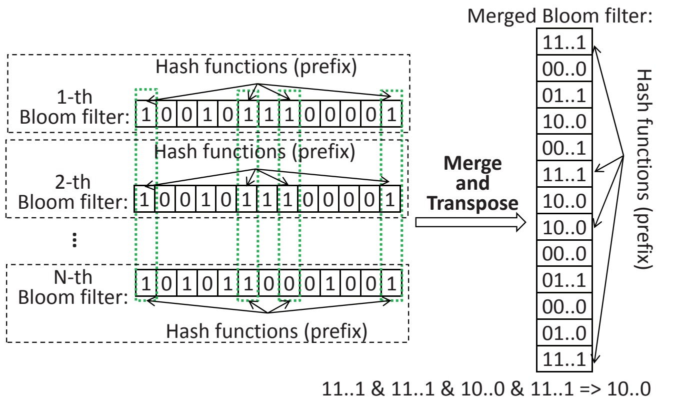


Fig. 2. Merge and transpose the traditional Bloom filters to Merged Bloom filter.

B. Merged Bloom filter

Here, we employ Merged Bloom filter to solve above port number dependent memory access problem.

The number of name prefixes in the Bloom filter for each port is close to one another, therefore we set the Bloom filters to be equal size and apply the same group of hash functions to all the N Bloom filters. The Merged Bloom filter combines the N Bloom filters bound to N forwarding ports, with N bits in the same location aggregated into one bit string. In memory, we store this bit string within a word or a few continuous words. As shown in Figure 2, N Bloom filters on the left are transformed into one Merged Bloom filter on the right. In Merged Bloom filter, each entry stores a bit string with machine word-aligned. From the most significant bit to the least significant bit, the N-th bit store the corresponding hash results from the N-th Bloom filter and the remaining unused bits are padded with 0. In Figure 2, the 1st location of each Bloom filter is 1 (corresponding to the results from the 1-st hash function), which turns into a bit string of 111...1 in the first array of the Merged Bloom filter via merging and permutation.

In this newly constructed data structure, there are only k hash computations required when a prefix looks for its forwarding port by obtaining the k bit strings corresponding to the k hash functions with an $'$AND$'$ operation finally implemented on them. The locations of $'1'$ in the results represent the final forwarding decision. For example, in the Figure 2, the bit strings are 11...1, 11...1, 10...0, 11...1, and the $'$AND$'$ result of them is 10...0, which denotes that the forwarding port is 1.

Merged Bloom filter transforms the serialized Bloom filter operations into parallel ones, which reduces the hash computation and memory access times from kN to k effectively, hence increasing the lookup speed tremendously.

C. String-based One Memory Access Bloom filter

Merged Bloom filter is inadequate for the first stage Bloom filters which are utilized to determine the prefix length of a name key since the number of prefixes in each Bloom filter differs from others substantially. This is because the name prefixes are assigned to Bloom filters according to their lengths, and the amounts of prefixes of the same length differ a lot.

Therefore, we need to improve the Bloom filter lookup speed of a single prefix to accelerate the name lookup speed. One memory access Bloom filter is first proposed by Yan Qian et al. [7]. The main idea is to map k hash function results into one single word, which will be obtained in one memory operation for all the k hash values, thus reducing the memory access times from k to one.

In traditional IP-based network tasks, the hashed object is mostly the IP address. The hash operations consume a relatively low computation resources while the memory access becomes the performance bottleneck. In NDN scenario, it will require at least one traverse of the whole string to make one hash computation and k hash functions lead to k times of string traverse, which makes the hash computation the dominant time consuming task in the matching process when using Bloom filters.

In our paper, we improve the performance of the string based hash to speed up the Bloom filter operations in the first stage. First, we calculate the first hash value of the prefix via DJB ('times 33') hash function [8]. Next, we obtain the location of the next hash value from the previous hash result in the prefix. After $k-1$ loops, the $k-1$ hash values are mapped into one word. The returned hash[0] is used to calculate the addresses in Bloom filters and the rest bits of the word hashed from corresponding addresses are calculated via one $'$AND$'$ operation. If all of the k-1 bits are 1s, the match is successful, in other cases the match is failed.

D. Update

As standard Bloom filter does not support dynamic deletion, we allocate a counting Bloom filter for each Bloom filter. First, a counting Bloom filter is constructed for each prefix set. Second, the counting Bloom filter is mapped into boolean Bloom filter. Finally, Bloom filters in the second stage are transformed into Merged Bloom filter.

The changes of name prefix table first modify the counting Bloom filters whose results will be mapped into Bloom filters in NameFilter afterwards, making dynamic insertions, updates and deletions of NameFilter incremental.

IV. Experimental Evaluation

In this section, we evaluate the performance of NameFilter and compare it with the other name lookup mechanisms in terms of lookup throughput, memory consumption, update rate as well as scalability.

A. Experimental setup

a) Prefix Table: Two Internet domain collections are used as prefix tables in our experiments. "3M prefix table", obtained from the paper [9], contains 2,763,780 entries; "10M prefix table", crawled from Internet, contains 10,000,000 entries. Each name table entry is composed of an URL-similar name and a pointer to the list of next-hop ports.

b) Name Trace: The name traces are used to test the lookup performance of our proposed methods. Two types of name traces are generated to measure the lookup performance under average workload and heavy workload, respectively. A name in name traces is formed by concatenating a prefix chosen from prefix tables with a randomly generated suffix. The prefixes of names in average workload traces are randomly selected from prefix tables; the prefixes of names in heavy workload traces are randomly chosen from the top 10% longest prefixes in the prefix tables.

c) Computational platform: The name lookup engine is implemented and run on a commodity PC with two 6-core CPUs (Intel Xeon E5645*2) Table I lists the details of relevant hardware configuration. The entire NameFilter program, developed using C++ programming language, consists of about 4,500 lines of codes. The part of multi-core parallel processing is developed using OpenMP API [10] in version 2.5.

TABLE I
HARDWARE CONFIGURATION.

Item	Specification
CPU	Intel Xeon E5645×2 (6 cores, 2 threads, 1.6GHz)
RAM	DDR3 ECC 48GB (1333MHz)
Motherboard	ASUS Z8PE-D12X (INTEL S5520)
Operating System	Fedora Linux 2.6.41.9-1.fc15.x86_64

B. Experimental results

We implement NameFilter and compare it with other four methods to expose the advantages of NameFilter. The performance of the traditional character trie (CharacterTrie) [4] is the baseline, and is improved by Name Component Encoding (NCE) [9] via encoding the components of a name. The other two methods are described in Section II and Section III-A, which are named as BloomHash and BloomFilter, respectively. By default, the system false positive of NameFilter and other methods in all experiments is set to 10^{-8}, with the load factor (α) of the hash table in BloomHash set to 0.5.

1) Throughput: First of all, we compare the name lookup speed on the 3M and 10M prefix table of five different methods using the average and heavy workload traces. Table II demonstrates the name lookup throughput of the methods running in one thread. On 10M prefix table, the name lookup speeds of CharacterTrie are 0.124 MSPS and 0.102 MSPS in average workload and heavy workload, while NCE achieves 0.256 MSPS and 0.235 MSPS, respectively. BloomHash effectively accelerates the name lookup speed to 1.024 MSPS and 0.834 MSPS, about 8 times speedup of CharacterTrie. Since BloomFilter needs to access all the Bloom filters of the ports, it can only achieve 0.185 MSPS and 0.178 MSPS. By applying One Memory Access Bloom filter and Merged Bloom filter, NameFilter boosts the name lookup speed to 1.895 MSPS and 1.834 MSPS, which is almost 17 times speedup of CharaterTrie and 1.8 times speedup of BloomHash.

The methods obtain the best name lookup performance with 24 parallel threads as shown in Table III. NameFilter achieves 37 MSPS and 32.59 MSPS in average workload and heavy workload, respectively, which means 12, 2.5 and 26 times speedup than CharaterTrie, BloomHash and BloomFilter respectively.

TABLE II
THE LOOKUP SPEED OF DIFFERENT METHODS WITH ONE PROCESSING THREAD.

Prefix Table	Trace	Lookup Speed (MSPS)				
		CharacterTrie	NCE	BloomHash	BloomFilter	NameFilter
3M	Average	0.188	0.254	1.043	0.203	2.033
3M	Heavy	0.156	0.247	0.977	0.185	1.903
10M	Average	0.124	0.256	1.024	0.185	1.895
10M	Heavy	0.102	0.235	0.834	0.178	1.834

TABLE III
THE LOOKUP SPEED OF DIFFERENT METHODS WITH 24 PARALLEL PROCESSING THREADS.

Prefix Table	Trace	Lookup Speed (MSPS)				
		CharacterTrie	NCE	BloomHash	BloomFilter	NameFilter
3M	Average	3.505	5.489	16.787	1.857	36.976
3M	Heavy	2.737	4.949	13.427	1.698	35.638
10M	Average	3.172	4.017	14.738	1.411	37.003
10M	Heavy	2.022	3.985	10.516	1.273	32.591

2) Memory: The memory consumption of the different methods are illustrated in Table IV. CharacterTrie needs 283.21 MB and 1,026.34 MB on 3M prefix table and 10M prefix table, respectively. Compared to CharacterTrie, NCE reduces about 30% memory consumption to 192.58 MB and 718.44 MB. While BloomHash achieves 34% memory saving and demands 184.35 MB and 680.49 MB. Furthermore, BloomFilter compresses the memory size to 52.63 MB and 190.10 MB which economizes more than 80% memory consumption. To keep the false positive probability less than 10^{-8}, NameFilter requires a little more memory than BloomFilter, 64.73 MB and 234.27 MB. Given it can boost 26 speedup even compared to CharacterTrie and BloomHash, NameFilter still saves more than 77% and 65% memory consumption, respectively.

3) Scalability: Figure 3 and Figure 4 demonstrate the lookup throughput of the 5 methods on different prefix table sizes. The name lookup speed of NameFilter maintains between 30 MSPS to 40 MSPS on different prefix table sizes, which means NameFilter has good scalability on different prefix table sizes.

The memory consumption of the methods on different prefix table sizes are illustrated in Figure 5. With the increased number of prefixes, all the five methods' memory sizes inflate. BloomFilter has the best memory efficiency and NameFilter's curve is close to BloomFilter's. Meanwhile, NameFilter's curve slope is less than BloomHash's, which means NameFilter is more suitable for larger prefix tables.

4) Update: As described in Section III-D, NameFilter supports incremental insertion and deletion. Figure 6 demonstrates that NameFilter achieves 3.0M insertions and 3.4M deletions per second, respectively. Meanwhile, it also shows that the

TABLE IV
THE MEMORY CONSUMPTION OF DIFFERENT METHODS.

Prefix Table	Memory Cosumption (MByte)				
	CharacterTrie	NCE	BloomHash	BloomFilter	NameFilter
3M	283.21	192.58	184.35	52.63	64.73
10M	1,026.34	718.44	680.49	190.10	234.27

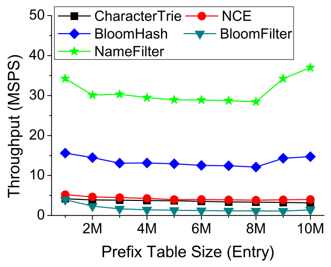


Fig. 3. The lookup throughput of the methods on different prefix table sizes (10M, Average workload, 24 Threads, k=8).

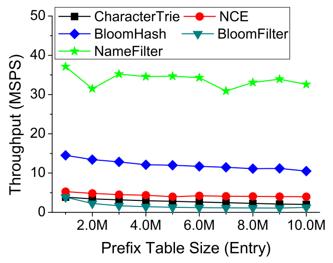


Fig. 4. The lookup throughput of the methods on different prefix table sizes (10M, Heavy workload, 24 Threads, k=8).

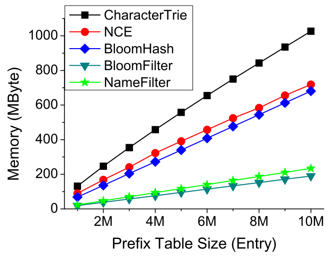


Fig. 5. The memory consumption of the methods on different prefix table sizes (10M, Average workload, 24 Threads, k=8).

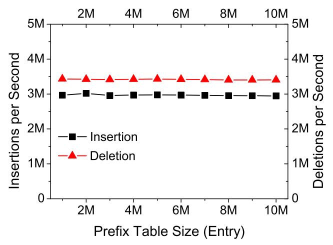


Fig. 6. The insertion and deletion performance of NameFilter on different prefix table sizes.

insertion and deletion performance is independent of the prefix table size.

V. Related Work

Until recently, three techniques, namely, hashing, trie and Bloom filter with their variations or hybrids are mostly discussed with trade-offs under different application constraints. Z. Genova *et al.* [11] and X. Li *et al.* [12] hash URLs to fixed-length signatures, and look up these signatures in the hash table, achieving a desirable lookup performance. However, these methods consider an URL as an indivisible entity, which fails to support longest prefix match. Some other methods try to decompose an URL into components using its hierarchical semantics. Our previous work [9] proposes a name encoding solution to compress the name characters with a state transition array to represent the prefix trie structure and optimize the traverse operations. With the trie structure, the lookup time complexity of these methods is bounded by the trie node traversing. Hashing needs to resolve collisions via chaining, leading to sequential memory access. Additional compressing or encoding subroutines occupy a considerable processor's computation resources.

Compared with hashing, Bloom filter is another way to represent name sets and actually a generalized form of hashing with trade-offs among memory occupation, memory access times and collision probability. Bloom filter assisted IP lookup is proposed by Sarang *et al.* [5] with IP addresses classified into different subsets according to their prefix lengths in the first stage, transforming the longest prefix match problem into exact match in multiple subsets. Within each subset, the exact match is performed using hashing in the second stage. M. Yu *et al.* [13] design a Bloom filter based router for flat name lookup at a low cost of on-chip memory. While the memory occupation is well controlled, only a 10% speed increase is reported comparing with the conventional methods. By contrast, our proposed NameFilter can achieve much larger speed acceleration when running on the multi-thread scenario. One of the drawbacks of Bloom filter is requiring large memory bandwidth when operating multiple hash functions. Y. Kanizo *et al.* [7] design a One Memory Access Bloom filter via organizing multiple hash result bits in the same machine word and reading them as a whole. This memory bandwidth reduction is conducted within one Bloom filter operation which is totally different from our port number independent approach. In our observation, as router's port number grows, corresponding increased number of Bloom filters devotes most of the memory access times.The proposed NameFilter solves the scalability issue via merging the second stage Bloom filters.

VI. Conclusion

In this paper, we propose NameFilter, a fast name lookup method with low memory cost via leveraging the two-stage Bloom filters for longest name prefix match in NDN. Merged Bloom filter and One Memory Access Bloom filter employed by NameFilter effectively reduce the memory access times, and the improved string-based hash functions further decrease the computation time. Extensive experiments on large scale name prefix tables demonstrate that NameFilter can greatly reduce the memory cost, improve the scalability and support incremental updates while achieving high-speed name lookup.

References

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking Named Content," in *Proc. of ACM CoNEXT*, 2009.

[2] L. Zhang, D. Estrin, V. Jacobson, and B. Zhang, "Named Data Networking (NDN) project," in *Technical Report, NDN-0001*, 2010.

[3] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM*, vol. 13, no. 7, pp. 422–426, July 1970.

[4] E. Fredkin, "Trie memory," *Commun. ACM*, vol. 3, no. 9, pp. 490–499, Sep. 1960.

[5] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest Prefix Matching using Bloom Filters," in *Proceedings of ACM SIMCOMM'03*, 2003.

[6] D. E. Knuth, *The Art of Computer Programming, volume 1/Fundamental Algorithms*. Addison-Wesley, 1968.

[7] Y. Qiao, T. Li, and S. Chen, "One memory access bloom filters and their generalization," in *INFOCOM, 2011 Proceedings IEEE*. IEEE, 2011, pp. 1745–1753.

[8] D. J. Bernstein, "DJB hash." [Online]. Available: http://www.partow.net/programming/hashfunctions/#DJBHashFunction

[9] 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 *Proceedings of IEEE ICDCS*, 2012.

[10] "The OpenMP API specification for parallel programming." [Online]. Available: http://openmp.org/wp/

[11] Z. Prodanoff and K. Christensen, "Managing routing tables for url routers in content distribution networks," *International Journal of Network Management*, vol. 14, no. 3, pp. 177–192, 2004.

[12] L. Xiao-Ming and F. Wang-Sen, "Two Effective Functions on Hashing URL [j]," *Journal of Software*, vol. 2, 2004.

[13] M. Yu, A. Fabrikant, and J. Rexford, "BUFFALO: Bloom filter forwarding architecture for large organizations," in *Proc. of ACM CoNEXT*. ACM, 2009, pp. 313–324.