BFAST: Unified and Scalable Index for NDN Forwarding Architecture

Huichen Dai, Jianyuan Lu, Yi Wang, Bin Liu
Tsinghua National Laboratory for Information Science and Technology (TNList)
Dept. of Computer Science and Technology, Tsinghua University
{dhc10, lu-jy11}@mails.tsinghua.edu.cn, wy@ieee.org, liub@tsinghua.edu.cn

Abstract—Named Data Networking (NDN) as an instantiation of the Content-Centric Networking (CCN) approach, embraces the major shift of the network function – from host-to-host conversation to content dissemination. The NDN forwarding architecture consists of three tables – Content Store (CS), Pending Interest Table (PIT) and Forwarding Information Base (FIB), as well as two lookup rules – Longest Prefix Match (LPM) and Exact Match (EM). A software-based implementation for this forwarding architecture would be cost-effective and scalable in memory resource usage. The pipelining technique is not readily applicable to table lookups. Therefore, forwarding a packet would go through multiple tables sequentially without pipelining, leading to high latency and low throughput. In order to take advantage of the software-based implementation and overcome its shortcomings, we find that, a single unified index that supports all the three tables and both LPM and EM lookup rules would benefit the forwarding performance. In this paper, we present such an index data structure called BFAST (Bloom Filter-Aided hash Table). BFAST employs a Counting Bloom Filter to balance the load among hash table buckets, making the number of prefixes in each non-empty bucket close to 1, thus enabling high lookup throughput and low latency. Evaluation results show that, for solely LMP lookup, BFAST can arrive at 36.41 million lookups per second (M/s) using 24 threads, and the latency is around 0.46 μs. When utilized to build the NDN forwarding architecture, BFAST obtains remarkable performance promotion under various request composition, e.g., BFAST achieves a lookup speed of 81.32 M/s with a synthetic request trace where 30% of the requests hit CS, another 30% hit PIT and the rest 40% hit FIB, while the lookup latency is only 0.29 μs.

I. INTRODUCTION

As network communication evolves from host-to-host communication to multi-party content distribution and retrieval, Named Data Networking (NDN) [1], [2] is proposed to embrace this paradigmatic shift. While NDN enables a plethora of new opportunities, it also introduces significant technical challenges, such as fast and scalable packet forwarding.

The forwarding architecture of NDN is much more complicated than that of IP. Conceptually it consists of three tables: Content Store (CS), Pending Interest Table (PIT) and Forwarding Information Base (FIB), as well as two lookup rules: Longest Prefix Match (LPM) and Exact Match (EM). CS is the index of cached contents on a router so that they can serve identical future requests immediately. PIT keeps track of all the requests that have been forwarded to the upstream but have not been responded with data contents. The incoming interface IDs of the requests are kept in PIT as well so that when data returns, the router can forward the data further downstream to the requesters through these interfaces. FIB is similar to the conventional forwarding table in that it contains a collection of prefixes and their corresponding next-hop interface(s), but the difference is that the prefixes are name prefixes and the lookup is name-based rather than address-based. NDN has two types of packets: Interest and Data: Interest packet is actually the request and Data packet is the response. Forwarding an Interest packet may involve looking up CS, PIT and FIB sequentially, while forwarding a Data packet involves looking up PIT and CS, FIB lookup is always LPM, CS lookup and PIT lookup are mostly EM.

While implementing this forwarding architecture, a software method is preferable since it is low-cost, flexible and has abundant memory resource, but it also makes the pipelining technique not readily applicable to serial table lookups. In this paper, we are in favor of the software implementation and aim to overcome this短coming. Existing works [3]–[7] treat CS, PIT and FIB as three separate indexes, and usually customize data structures and lookup algorithms for each index. Separate indexes mean multiple table lookups when forwarding a packet (usually three lookups for an Interest packet and two lookups for a Data packet), making it challenging to meet the goal of high-speed and low-latency lookup. In summary, we are faced with the following challenges when implementing a high-speed NDN forwarding architecture in software:

1) **High-speed name lookup.** NDN’s content names are hierarchically structured and have variable and unbounded lengths. Typically, a name can contain tens or even hundreds of characters (much longer than IPv4/v6 addresses), making name lookup against a single table a time- and resource-consuming process. Therefore, achieving wire-speed (e.g., OC-192) name lookup for multiple tables in NDN is definitely a big challenge.

2) **Low latency.** Latency has strong impacts on user experience, especially for real-time and delay-sensitive applications such as voice over Internet. Therefore, low latency is an important criteria for a forwarding architecture.

This work is supported by 863 project (2013AA013502), NSFC (61373143, 61432009, 61402254), the Specialized Research Fund for the Doctoral Program of Higher Education of China (20131019172), Tsinghua University Initiative Scientific Research Program (20121080068), CISCO Award Fund, China Postdoctoral Science Foundation (No. 2014M550734, No.2013M540949), and Jiangsu Future Networks Innovation Institute: Prospective Research Project on Future Networks (No. BY2013095-1-03). Corresponding Author: Bin Liu (liub@tsinghua.edu.cn)

There may be cases that different lookup rules are needed in CS and PIT lookup, e.g., when we want an incoming Data packet to satisfy multiple, different pending Interest packets. Since these cases are not clearly defined yet, in this paper we assume EM for both CS and PIT lookup.
3) *Incremental update*. The forwarding architecture should support incremental update so that normal lookups on the forwarding plane will not be impeded by updates.

4) *Scalability*. The index should be able to maintain good performance as the table sizes extend to large scale.

To address these challenges, we realize that a *unified index* which can simultaneously accommodate CS, PIT and FIB and two lookup rules will potentially help improve the lookup performance: it requires only one index lookup when forwarding a packet, which shortens the lookup delay and improves the throughput. This is where the performance benefit comes from. To this end, we design, implement and evaluate a novel data structure called BFAST (Bloom Filter-Aided Hash Table) for the unified index. This index utilizes a Counting Bloom Filter to balance the load among hash table buckets, making the number of prefixes in each non-empty bucket close to 1, and thus achieving high lookup, insertion and deletion performance. Especially, we make the following contributions:

1) An elaborate data structure called BFAST that implements the unified index for CS, PIT and FIB simultaneously;

2) BFAST achieves high throughput and low latency for lookup, insertion and deletion operations; the improvements are remarkable compared with previous works;

3) BFAST supports incremental update and can sustain a stable lookup performance as the number of table entries grows.

Theoretical analysis on BFAST reveals that 1) the expected number of entries in a hash table bucket is only 1.017 (similar to ball placing problem [8]), 2) BFAST achieves a very low lookup error rate of $2.36 \times 10^{-20}$. Experiments show that, using 24 threads, BFAST arrives at 36.41 million lookups per second (M/s) for solely LPM lookup, and the latency is around 0.46 $\mu$s. Additionally, BFAST achieves 1.56 M/s insertion speed and 3.01 M/s deletion speed. When utilized to build the NDN forwarding architecture, BFAST obtains remarkable throughput promotion under various request composition, e.g., BFAST achieves a speed of 81.32 M/s with a synthetic request trace where 30% requests hit CS, another 30% hit PIT and the rest 40% hit FIB, while the lookup latency is only 0.29 $\mu$s.

In the rest of this paper, Section II describes the NDN background. Section III presents the detailed design for BFAST, and Section IV analyzes it from the theoretical viewpoint. Section V evaluates BFAST on lookup throughput, insertion and deletion speed, as well as scalability. Section VI surveys related work and Section VII concludes the paper.

**II. NDN Overview**

NDN is a novel network architecture proposed by [1], it follows the Content-Centric Networking (CCN) [2] paradigm. Different from the current Internet, it makes content (“what”) as its central role, rather than “where” content is located. NDN has a vast background that we cannot fully cover, this section briefly introduces NDN with a focus on its forwarding architecture.

**A. Naming**

A critical distinction from IP is that, every piece of content in NDN network has an assigned name to uniquely identify it, and packets are routed/forwarded by names. NDN names are application-dependent and opaque to the network, but they all share some common characteristics – hierarchically structured and composed of explicitly delimited *components*. A typical example of an NDN name is a reversed domain name followed by a directory-style path, e.g., `org/ieee-conference/2015/cfp.html` where `org/ieee-conference/` is the reversed domain name of `ieee-conference.org`, and `2015/cfp.html` is the content’s directory path on the website server. ‘/’ is the component boundary delimiter; `org`, `ieee-conference`, `2015` and `cfp.html` are the 4 components of the name. We define prefix level as the number of components in a prefix. Name’s hierarchical structure enables name aggregation and allows LPM name lookup.

**B. NDN Communication**

NDN adopts a requester-driven communication mechanism. To retrieve data, a consumer sends out an Interest packet with the name of the desired content in its header. Routers use this name to route and forward the Interest towards data sources, and a Data packet that carries both the name and the desired content is returned to the consumer, following the reverse path of the Interest packet. Unlike IP, an Interest packet and its matching Data packet always take symmetric paths in NDN.

**C. Forwarding Architecture**

NDN has a complicated forwarding architecture, though Interest packet and Data packet are both forwarded by the names in their headers, they have different forwarding rules. The NDN forwarding architecture consists of three tables – CS, PIT and FIB. CS temporarily caches Data packets that traverse this router to serve subsequent request soliciting the same content. PIT keeps track of all the “not yet satisfied” Interest packets that have been sent upstream towards data sources. Each PIT entry consists of the name in the Interest packet and its incoming interface list. FIB, or routing table, maps name prefixes to their proper next-hop interface(s), specifying the directions where the Interest packets should be forwarded to.

The packet lookup and forwarding process for Interest and Data packet are illustrated by Fig. 1(a) and Fig. 1(b), respectively. Fig. 1(a) shows that once an Interest packet $I$ arrives at interface $i$ of an NDN router $R$, $R$: 1) consults CS if the desired content is present, if so, returns a copy in Data packet via $i$, 2) if not, looks up PIT to see if PIT has an entry for $I$. If so, adds $i$ to that entry, and discards $I$, 3) otherwise, creates a PIT entry for $I$ and add $i$ to this entry, and 4) forwards $I$ to the next-hop interface(s) by looking up FIB. When Data packet $D$ returns, Fig. 1(b) shows that $R$: 1) forwards $D$ over all the requesting interfaces in the corresponding PIT entry, 2) deletes this PIT entry, 3) caches $D$ in the CS based on certain caching policy.

**III. Unified Index for NDN Forwarding Architecture**

**A. Why Unified Index?**

Generally, Interest packet looks up the three tables in the order of CS, PIT and FIB, while Data packet looks up the PIT and CS sequentially, as illustrated by Fig. 1(a) and Fig. 1(b), respectively. Previous works consider that each table has a separate index, so forwarding each packet requires more than
one table lookup. This is undesirable since multiple lookups for one packet forwarding will decrease the forwarding speed and lengthen the lookup delay.

If there is a unified index that can accommodate all the three tables, then the tables can share a common index, and multiple lookups can be reduced to only once, which can be implemented by a single lookup function like lookup(). rather than functions like CS_lookup(), PIT_lookup() and FIB_lookup(). The function lookup() can be viewed as the lookup function abstraction of the unified index. Each entry in the index has a pointer field pointing to the entry of one table among CS, PIT and FIB, and its associative type field tells which table that entry belongs to.

B. Data Structure for Unified Index – BFAST

Hereby we propose a data structure called BFAST (Bloom Filter-Aided haSh Table) that is suitable to implement the unified index. Hash table as a basic data structure can meet the requirements like EM lookup, convenient insertion and deletion if chaining (linked list) is used to resolve hash collisions. Chaining has better performance than open addressing schemes and is therefore widely-adopted [9]. Hash table can implement LPM using multiple EM lookups with prefixes of different lengths as keys, from the longest prefix to the shortest one.

However, hash table also has the shortcoming of unbalanced load among its buckets, i.e., the length of the linked list in each bucket may differ a lot. The longer the linked list, the longer the lookup time. Meanwhile, non-uniform lengths of the linked lists can lead to jitter of packet delays, imposing negative impacts on delay-sensitive applications. Therefore, we propose to use multiple hash functions to balance the load among all the buckets. So each item is hashed to multiple candidate buckets by multiple hash functions, and among all the candidate buckets, the item is placed into the least loaded one, i.e., the one with the fewest items in it. In case of a tie, the bucket with the smallest subscript is chosen. (This is better than random choice [8], [10].)

The multiple-function idea has shown better performance than a single hash function [8], [11], [12]. However, this method requires traversing all the candidate buckets to obtain the least loaded one, resulting in many memory accesses. Faced with this problem, we assign a counter \( C_i \) to each bucket \( B_i \) in the hash table to record the number of items in it, as illustrated by Fig. 2. In this figure, three hash functions and four name prefixes are used for illustration. The prefixes are inserted in the order of com/google, com/live, com/yahoo, com/amazon (the same order for subsequent figures).

Generally, assume that we use \( k \) functions \( f_1(), f_2(), \ldots, f_k() \), and the hash table has \( N \) buckets. (So there are \( N \) counters in total.) With the help of these counters, inserting an item into the hash table needs only picking up the counter with the smallest value among the \( k \) candidate ones, and then inserting the item into that counter’s associated bucket. This is a simple but effective improvement since checking \( k \) counters is much faster than traversing \( k \) buckets, especially when the counters (or part of them) can be placed in on-chip memories whereas the hash table itself is placed in slower memories due to its larger size.

Taking a closer look at the counters in Fig. 2, plus the \( k \) hash functions, we find that they can form a Counting Bloom Filter (CBF) [13], as depicted in Fig. 3. Now inserting an item into the hash table consists of two steps: 1) insert the item into CBF, 2) insert the item into the least loaded bucket. This CBF brings many advantages, including: 1) fast EM lookup, 2) quick return when looking up an item that is not in the hash table, since there is no need to access the hash table itself, and these lead to 3) fast
LPM lookup because multiple lookup trials are needed and only one prefix will hit, 4) the possibility that CBF (or part of it) can be placed in the fast on-chip memory because of its relatively small size compared with the hash table, promoting the overall lookup performance. Note that the counter value here no longer reflects the number of items actually present in its associated bucket but is usually greater than that, and the bucket that an item is to insert may also be different. It is a less loaded bucket, but not necessarily the least loaded one. Experimental results show that this still achieves very good load-balancing result.

However, this data structure has difficulty dealing with lookups and deletions. Assume that when item x is inserted into the index, C_i is the counter with the smallest value. After this insertion, several other items are inserted subsequently, and these insertions may increase the value of C_i. When we look up item x, C_i may no longer be the smallest counter, and we cannot find x any more without traversing all the k candidate buckets, as is the case for the deletion operation.

Be aware that the data structure in Fig. 3 is also observed by Haoyu Song et al. [14], they are also faced with the problems of lookup and deletion. We have different ways addressing them, since this is a preliminary data structure in both designs, we eventually evolve and diverge into different solutions for lookup and deletion (refer to Section VI).

In order to pursue good support for lookup and insertion, we propose to use k assisting or auxiliary Counting Bloom Filters (aCBF_i, 1 ≤ i ≤ k) to record which hash function calculates the subscript of the smallest counter when inserting an item, as depicted in Fig. 4. (To distinguish aCBFs with the CBF introduced earlier, we call that CBF the main CBF, abbreviated as mCBF.) The aCBF works as follows when inserting an item x: 1) calculate k hash values by f_1(x), f_2(x), · · · , f_k(x), 2) insert x into the mCBF, and pick up the smallest counter (assume it is C_i) among k candidate ones, then insert x into bucket B_i, 3) assume that the subscript of C_i (i.e., i, namely the location in the mCBF) is calculated by the j-th hash function, that is i = f_j(x)%N, then insert x into aCBF_j. This means aCBF_j records all the items whose bucket subscripts in the hash table are derived from the j-th hash function (f_j(·)%N). This insertion process is illustrated by Algorithm 1, where lines 3 ~ 4 insert x into mCBF, lines 6 ~ 12 insert x into the least-loaded bucket, and lines 14 ~ 15 insert x into the corresponding aCBF. (The LengthOf(CBF) function returns the number of counters in that CBF.) Note that all the mCBF, aCBF and the hash table share the same suite of hash functions, therefore, hash values of an item need to be calculated only once.

With the help of aCBF, lookup and deletion can now be realized without traversing all the k candidate buckets. When looking up an item x, we first look up the mCBF to see if x is present in the index, then look up aCBFs to see which aCBF contains x. Assume it is aCBF_j, then the subscript i of the corresponding bucket B_i is derived from i = f_j(x)%N. In this way, we can find x in bucket B_i (Algorithm 2). The deletion operation can repeat this lookup process and remove x from bucket B_i, as well as from mCBF and aCBF, after finding it (Algorithm 3).

By far, we have introduced the complete BFAST data structure, as well as its EM lookup, insertion and deletion algorithms.

Algorithm 1 Insert item into Hash Table (with mCBF and aCBF)
1: procedure InsertItem(item x)
2: ▷ First insert x into mCBF
3: for i : 1 → k do
4: mCBF.C_i(x)%N ++
5: ▷ Then insert x into the least-loaded bucket
6: C_min ← MAX_VALUE
7: for i : 1 → k do
8: if C_min > mCBF.C_i(x)%N then
9: C_min ← mCBF.C_i(x)%N
10: j ← f_j(x)%N
11: m ← i
12: B_j ← B_j ∪ x ▷ Insert x into B_j
13: ▷ At last, insert x into the mth aCBF
14: for i : 1 → k do
15: aCBF_m.C_j(x)%LengthOf(aCBF_m) ++

Algorithm 2 Lookup item in Hash Table (with mCBF and aCBF)
1: procedure LookupItem(item x)
2: ▷ Quick check if x is in mCBF
3: for i : 1 → k do
4: if mCBF.C_i(x)%N = 0 then
5: return
6: ▷ Look up x in aCBF_i...aCBF_k
7: for i : 1 → k do
8: if CBF_query(aCBF_i, x) = TRUE then
9: j ← f_j(x)%N
10: for item y ∈ B_j do
11: if y.key = x then ▷ found
12: return y

Algorithm 3 Delete item from Hash Table (with mCBF and aCBF)
1: procedure DeleteItem(item x)
2: ▷ Quick check if x is in mCBF
3: for i : 1 → k do
4: if mCBF.C_i(x)%N = 0 then
5: return
6: ▷ Delete x from mCBF
7: for i : 1 → k do
8: mCBF.C_i(x)%N -=
9: ▷ Look up x in aCBF_i...aCBF_k
10: for i : 1 → k do
11: if CBF_query(aCBF_i, x) = TRUE then
12: j ← f_j(x)%N
13: for item y ∈ B_j do
14: if y.key = x then ▷ found
15: B_j ← B_j − y ▷ remove y from B_j
16: CBF_delete(aCBF_i, x) ▷ delete x from aCBF_i
17: return

Algorithm 4 Longest Prefix Match Lookup
1: procedure LPMLookup(item name)
2: n ← NumberOfComponents(name)
3: for i : n → 1 do
4: ▷ get first i components of name as a prefix
5: prefix ← FirstNComponents(name, i)
6: if LookupItem(prefix) ≠ NULL then
7: return LookupItem(prefix)

Afterwards, LPM lookup for a name can be implemented in this way: 1) generate all the possible prefixes from that name; 2) lookup the prefixes from the longest one to the shortest one by invoking LookupItem() in Algorithm 2; 3) the first matched prefix is the longest prefix. This procedure is shown in Algorithm 4.

Introducing CBF to BFAST will bring false positives when looking up an item, these false positives, however, will only lead to extremely low lookup error rate (Section IV).
probability that a certain bucket will be chosen for one time, the length \( L \) of the longest linked list is \( \sum_{i=1}^{M} P\{L = i\} \cdot i \). In our experiments, \( k \) is set to 8. Based on the fixed false positive rate (around \( 10^{-5} \)), we choose the number of prefixes \( M \) by calculating \( N \) according to Formula (7). E.g., if \( M = 1 \) million, then \( N \) is around 29.6 million. Substituting \( M, N \) and \( k \) into (4), we can get the value of \( E[L] \approx 10.17 \). This value is the same for different values of \( M \) since \( M \) and \( N \) are dependent. Let \( L_{\text{longest}} \) denote the length of the longest linked list, we also deduce a loose upper bound on \( L_{\text{longest}} \). We abstract the item inserting process by a ball placing problem. Suppose that we sequentially place \( M \) balls into \( N \) bins, for each ball we choose \( k \) bins at random and place the ball into the least loaded one. In case of a tie, the left-most one is chosen. When \( M = N \), Berthold Vo\"cking [8] has proven that the number of balls in the fullest bucket is

\[
L_{\text{fullest}} = \frac{\ln N}{k \ln \phi_k} + O(1)
\]  

where \( \phi_k = \lim_{d \to \infty} \sqrt[k]{F_k(d)} \), and \( F_k(d) \) is the \( k \)-ary Fibonacci numbers, \( F_k(d) = \sum_{i=1}^{k} F_k(d - i) \). In fact, (for \( N = 1 \) million)

\[
\frac{\ln N}{k \ln \phi_k} < \frac{\ln N}{(k-1) \ln 2} < \frac{\ln N}{\ln k} = 1.2627
\]

In our case, the names are the balls and the buckets are the bins, \( M < N \), therefore, the longest length \( L_{\text{longest}} \) of the linked list will be less than \( L_{\text{fullest}} \) (\( L_{\text{longest}} < L_{\text{fullest}} \)).

### B. Lookup Error Rate

First of all, we define the lookup error as returning a valid interface ID for a name or name prefix not present in the index. Other errors like returning an invalid interface ID (e.g., negative integer) for a name or name prefix present in the hash table will not happen in our index.

During the name lookup process, we have to check the mCBF and aCBF, and compare the key’s signature with the signature in the bucket’s linked list. The possible errors will result from the false positives introduced by the mCBF and aCBF, as well as the signature collisions (different keys mapped to the same
hash value). Assume that we have inserted $M$ items into the mCBF with $N$ counters. For each item, it has equal possibility to be inserted into $aCBF_i, (1 \leq i \leq k)$, therefore, we let each $aCBF$ consist of $N/k$ counters (due to Bloom Filter's linearly property). We take a 32-bit hash value as the name’s signature, such as CRC-32.

The name lookup process needs three operations: 1) look up mCBF, 2) look up aCBF, and 3) signature comparison. Only false positives in mCBF and aCBF, as well as signature collision happens simultaneously can lead to lookup error; a single CBF false positive, or a single signature collision will not result in any lookup error. This observation implies that the overall error rate is the product of multiplying the error rate in each step. The false positives for mCBF and each aCBF are set to $1.0 \times 10^{-5}$, which means

$$P_{fp} = (1 - (1 - \frac{1}{N})^{kM})^k \approx (1 - e^{-kM/N})^k = 1.0 \times 10^{-5} \quad (7)$$

Since the aCBF group has $k$ CBFs, the overall false positive rate of $k$ CBFs is $1 - (1 - P_{fp})^k$. The collision probability among CRC-32 hashes of $H$ given names is $(H - 1)/2^{32}$, under the assumption that CRC-32 is perfectly unbiased. In our case, $H = E[L] + 1$, where $E[L]$ denotes the average number of names in the bucket, and the additional 1 stands for the input name (the key). Therefore, the lookup error rate, or for an item not present in the index, the probability that it is found in the index is

$$P_e = P_{fp} \cdot (1 - (1 - P_{fp})^k) \cdot \frac{E[L]}{2^{32}} = 2.36 \times 10^{-20} \quad (8)$$

$P_e$ is the error rate of the Exact Match lookup of our index. Obviously, it is so small and can be neglected.

C. Memory Access Complexity of BFAST Lookup

Since the lookup performance is generally bounded by the number of memory accesses, next we analyze the memory access complexity of a successful LPM operation of BFAST from the theoretical viewpoint.

For BFAST, each exact match lookup consumes $k$ memory accesses to query the mCBF. If hit, another $k + L$ memory accesses are required to query the aCBF and a specific hash table bucket. (The aCBFs are implemented in an interleaved manner, which means the cells at the same position in each aCBF are placed at contiguous addresses in the memory [15]. Therefore, accessing $k$ aCBFs requires $k$ memory accesses.) Be $L$ the maximum prefix length in the FIB in terms of components, i.e., the maximum prefix level. Therefore, one name lookup requires $(L - L_p + 1) \cdot k + k + L = (L - L_p + 2) \cdot k + L$ memory accesses, where $L_p$ is the prefix length of the searched name in terms of components. Following the big-O notation in the time complexity representation, this memory access complexity is $O((L - L_p + 2) \cdot k + L)$. (Note that $E[L] \approx 1.017$.) As we shall see, it is smaller than NameFilter’s memory access complexity.

NameFilter maintains two stages of Bloom Filters. The first stage has $L$ Bloom Filters, each of them records the prefixes of the same level, while the second stage has $P$ ($P$ is the number of ports/interfaces on a router) Bloom Filters, prefixes with the same next-hop interface are recorded in the same Bloom Filter. NameFilter firstly identifies name’s prefix level by sequentially querying first-stage Bloom Filters (from the Bloom Filter recording the longest prefixes to the one recording the shortest ones), leading to $(L - L_p + 1) \cdot k$ memory accesses. Secondly, NameFilter checks all the second-stage Bloom Filters to obtain the port number, requiring $P \cdot k$ memory accesses. In total, NameFilter’s complexity is $O((L - L_p + 1) \cdot k + P \cdot k)$.

V. Evaluation

This section evaluates the performance of the NDN forwarding architecture based on BFAST (refer to Fig. 1 for lookup logic), and compares it with other lookup methods. BFAST is implemented using $k = 8$ hash functions. For comparison, other representative methods including NameFilter [6], NCE [4], [5] and CCNx [16] are also implemented. Refer to Section VI for the logics of NameFilter, NCE and CCNx.

A. Experimental Setup

1) Prefix Sets and Name Traces: We collect two domain name sets to compose the NDN FIBs, the first set contains 2,762,737 domain names (around 3 Million) and the second one contains 9,998,795 ones (around 10 Million). The domain names in each set are hierarchically reversed to NDN-style name prefixes. These prefixes, plus a next-hop interface ID for each, compose the NDN FIB. Therefore, we have two FIBs, and they are referred to as 3M FIB and 10M FIB, respectively. As aforementioned, we define prefix level as the number of components in a prefix. The distribution of prefix levels and average prefix length are shown in Fig. 6. Using these two FIBs to build BFAST, the average length $L_{average}$ of the linked list in each bucket is 1.0002 for the 3M FIB and 1.0512 for the 10M FIB, which are consistent with the analysis in Section IV.

We then generate name traces based on the two FIBs to mimic the content names in NDN packet headers. Because the lookup processes are different for Interest and Data packet, we take the traces as Interest packets, whose lookup process is more complicated. The names in the traces are formed by appending randomly generated suffixes to name prefixes in the FIBs. We generate two types of name traces for each FIB to simulate average and heavy [6] workload of lookup. The name length and level distributions of each trace are illustrated in Fig. 7.

2) Platform: The experiments are conducted on a commodity server platform, running OS Linux 2.6.43. Platform hardware configuration is listed in Table I. The forwarding architecture is implemented by the C++ programming language, using OpenMP [17] to support multi-thread programming.

<table>
<thead>
<tr>
<th>Item</th>
<th>Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU</td>
<td>Intel Xeon E5645×2 (6 cores×2 threads, 1.6GHz)</td>
</tr>
<tr>
<td>RAM</td>
<td>DDR3 ECC 48GB (1.333MHz)</td>
</tr>
<tr>
<td>Motherboard</td>
<td>ASUS Z8PE-D12X (INTEL S5520)</td>
</tr>
</tbody>
</table>

B. Longest Prefix Match Lookup performance

As the first step of our experiments, we measure and compare the LPM lookup throughput on the 3M and 10M FIB, for
outperforms NameFilter, and gains 6.72 of NameFilter, NCE and CCNx, respectively. This speed is 1.22, 4.15 and 11.1 times the highest throughput average workload of 10M FIB – 36.41 M/s (with 24 threads).

Fig. 6: Distribution of prefix levels and average prefix length for the 3M FIB and 10M FIB. Percentage numbers indicate the ratio among all the prefixes in each FIB.

The four methods of BFAST, NameFilter, NCE and CCNx. The measurements are under average and heavy workload, from single thread to multi-thread environment. The single thread results are shown in Table II, which reveals that the BFAST index achieves the highest lookup throughput under both average and heavy workload. The highest speed reaches 2.33 M/s under the average workload of 10M FIB, obtaining 1.27×, 4.49× and 12.40× speedup over NameFilter, NCE and CCNx, respectively. Under heavy workload, BFAST slightly outperforms NameFilter, and gains 6.72× and 19.52× speedup over NCE and CCNx (10M FIB heavy workload), respectively.

Fig. 8 shows the results of multi-thread name lookup, with four methods under 1 to 48 parallel threads. Obviously, the lookup speed increases monotonically as the thread number grows from 1 to 24, and then remains relatively stable from 25 to 48 threads. BFAST still outperforms other methods with multiple threads. The highest throughput is achieved by BFAST under average workload of 10M FIB — 36.41 M/s (with 24 threads). This speed is 1.22, 4.15 and 11.1 times the highest throughput of NameFilter, NCE and CCNx, respectively.

C. Scalability

1) LPM Lookup performance: BFAST has already shown its better lookup performance over other methods. We are interested in the trend of its lookup performance as FIB size grows. To this end, we partition each FIB into 10 equal-sized subsets, and compose 10 sub-FIBs for each FIB: the i-th sub-FIB consists of the first i subsets. We conduct lookup experiments based on the 20 generated sub-FIBs with 24 parallel threads. Experimental results are presented in Fig. 9. For all the four methods, the lookup performance slightly degrades as the FIB sizes gradually grows. BFAST remains relatively stable under 3M FIB and tends to stabilize after 7-th or 8-th sub-FIB under the 10M FIB.

2) Insertion and deletion performance: We measure the insertion and deletion performance of BFAST based on the sub-FIBs too. Experimental results are presented in Fig. 10, which shows that the insertion and deletion speeds keep relatively stable as the FIB size grows. BFAST achieves around 1.56 M/s and 3.01 M/s for 3M FIB insertion and deletion, 1.41 M/s and 2.86 M/s for 10M FIB insertion and deletion. Note that these updates are performed directly on the forwarding plane, rather than on the control plane.

3) LPM Lookup Latency: The name lookup latencies are shown in Fig. 11 and Fig. 12, which reveal that BFAST achieves the lowest delay. (0.49 μs and 0.46 μs for 3M FIB and 10M FIB respectively).

D. Memory Consumption

The memory consumption of the four methods are listed in Table III. All the Bloom Filters have a false positive rate of 10^-5, and each counter is assigned 4 bits. BFAST consumes 419.32 MB and 1,517.60 MB for the 3M and 10M FIB, respectively. Compared with other methods, memory consumption is a drawback of BFAST. The reason why BFAST consumes so much memory is it uses aCBF to do the bookkeeping work and the false positive rate is extremely low. Therefore, one of our future work aims to reduce the memory cost of BFAST. Note that though two-stage Bloom Filters in NameFilter only consumes 64.73 MB and 234.27 MB for two FIBs respectively, it requires an extra CBF (with size 104.89 MB and 379.60 MB) on the control plane to handle FIB updates. Moreover, each time the FIB gets updated, the latest version must be downloaded to the forwarding plane, which not only consumes enormous time and memory bandwidth, but also interrupts the lookup process frequently.

<table>
<thead>
<tr>
<th>TABLE III: Memory consumption (MB)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
</tr>
<tr>
<td>BFAST</td>
</tr>
<tr>
<td>NameFilter</td>
</tr>
<tr>
<td>NCE</td>
</tr>
<tr>
<td>CCNx</td>
</tr>
</tbody>
</table>

E. Forwarding Architecture Lookup Performance

Previous experiments focus on the LPM lookup throughput for the four respective methods. In the following, we experiment on the lookup performance of the whole NDN forwarding architecture based on the four methods. For NameFilter, NCE and CCNx, we place two hash tables before the FIB to mimic
CS and PIT. Among all the Interest packets in the trace, some will hit CS, others will hit PIT and the rest will hit FIB. Therefore, 4 compositions of the Interest trace are considered: <0%,0%,100%>, <10%,10%,80%>, <20%,20%,60%> and <30%,30%,40%>. E.g., <10%,10%,80%> means 10% Interest packets hit CS, another 10% hit PIT and the rest 80% hit FIB, which further indicates that 10% Interest packets require one EM lookup, another 10% need two EM lookups, and the rest 80% incur two EM lookups and one LPM lookup. Experimental results are presented in Fig. 13. (For clarity, the CCNx is not presented due to its low throughput.) These figures elucidate that when used for the entire NDN forwarding engine, BFAST has absolute performance advantage over other methods! With <30%,30%,40%> Interest composition, BFAST achieves the highest speed of 81.32 M/s under average workload of 10M FIB, while NameFilter and NCE obtains 15.59 M/s and 8.31 M/s respectively, meaning 5.21 and 9.78 speedups for BFAST over NameFilter and NCE.

The average latencies of the forwarding plane under different request compositions are presented in Table IV, which reveals that BFAST has the lowest delay to forward an Interest packet.

VI. RELATED WORK

We survey related work in two aspects: 1) fast hashing techniques, 2) studies on data structure for NDN table indexes. Looking up a hash table involves hash value computation followed by memory accesses. A basic hash table cannot avoid hash collisions and will lead to more memory accesses, which takes much more time than hash value computation. Multiple hash functions have been proven to perform better than a single hash function [8], [10]–[12]. Balanced Allocations [11] uses only one hash table but more than one hash function to show that multiple hash functions achieve much more balanced results than a single hash function. Vöcking [8] takes one step further to prove that in case of tie due to multiple hash functions, asymmetric tie-breaking rules will again help load balance, obtaining better load-balancing results than [11]. These studies are similar to the starting phase of our BFAST data structure.

Multilevel Adaptive Hashing [12] is another method, which uses multiple hash tables each with different hash functions.
The items colliding in one table are hashed into other tables. This method is employed to boost fast IP lookup [10].

Haoyu Song et al. also propose a fast hash table structure by combining a CBF and an ordinary hash table [14], from which we observe a similar but preliminary data structure as in Fig. 3. We are both aware that this data structure has low efficiency dealing with lookups and deletions. To support fast lookup and deletion, our solutions diverge into different designs at last; Haoyu Song et al. address this problem by sharing items among multiple hash table buckets, while we address this problem using auxiliary CBFs.

There have been several works on index data structure design for NDN name lookup. NCE [5] employs a component-grained trie structure to organize name prefixes, where each edge stands for a name component. Afterwards, the name components are represented and replaced by a unique code (integer), making name lookup faster, but NCE has poor support for EM lookup. Moreover, as more and more prefixes are inserted, the nodes in the trie will have numerous outgoing edges, slowing down the lookup process. DiPIT [3] proposes a Bloom Filter-based data structure to compose PIT, while Varvello et al. [18] implement PIT using both linear-chaining hash table and open-addressed d-left hash table. In [19], hash table is employed to index all the three tables; improvements include an optimization method for LPM lookup, integrating CS and PIT into the same index. This reveals that the authors have also seen the benefit of combining some tables together using a suitable data structure. NameFilter [6] employs Bloom Filters to achieve LPM lookup for FIB. Though ingenious in design and fast in speed, NameFilter employs Bloom Filters to achieve LPM lookup performance. Meanwhile, it also supports incremental updates. Evaluation results indicate that BFAST has better LPM lookup performance over other methods, and its throughput sustains as the table sizes grow. Additionally, it also achieves high-speed incremental insertions and deletions. When utilized to build the NDN forwarding architecture, it achieves remarkable throughput promotion under various request compositions, outperforming other methods with significant performance advantage.

**References**


