Packet Forwarding in Named Data Networking Requirements and Survey of Solutions

Zhuo Li, *Member, IEEE,* Yaping Xu, Beichuan Zhang, *Member, IEEE,* Liu Yan, and Kaihua Liu, *Member, IEEE*

***Abstract*—Named Data Networking (NDN) is the most promising paradigm recently conceived for future Internet architectures, where communications are driven by contents instead of host addresses. To realize this novel paradigm, three novel tables, namely Content Store, Pending Interest Table (PIT) and Forwarding Information Base (FIB), are utilized in NDN forwarding plane. Designing and evaluating the quick enough forwarding plane with high capacity is a major challenge within the overall NDN research area. Since NDN was proposed in 2010, there have been many efforts focusing on this challenge and a rich literature has been developed. Unfortunately, there is a lack of the comprehensive sketch about the requirements of NDN forwarding plane and the study on various schemes proposed. Focusing on the above insufficiency, this survey gives the complete requirements and compares all the schemes proposed for NDN forwarding plane based on the data structure utilized. In addition, the survey also discusses some issues, challenges and directions in future research. It is considered that designing a novel data structure to meet all requirements of the forwarding plane and studying on a better structure of Content Store play important roles, while discussing the necessity of a unified index, combining with other contents in NDN research and implementing a unified benchmark are also required in this domain.**

***Index Terms*—Named Data Networking (NDN), Forwarding Plane, Requirements, Design.**

I. Introduction

While the Internet was originally designed for host-to-host communication, it has been more inclined to be used for information dissemination and retrieval, which has significantly presented some challenges for its architecture [1]. Therefore, Information-Centric Networking (ICN) [1] becomes the most promising proposal for the future Internet architecture. In ICN, naming information at the network layer makes it feasible to deploy in-network caching and multicast mechanisms, which promotes time-efficient and resource-efficient information distribution. One of the most promising ICN paradigms is the Named Data Networking (NDN) project funded by the US Future Internet Architecture program proposed in 2010 [2], which was evolved from the Content-Centric Networking (CCN) proposed by Palo Alto Research Center [3].

Rather than forwarding packets based on their destination addresses in IP, NDN forwards packets based on named data [2], in which the names are hierarchically structured like URL to facilitate traffic demultiplexing and provide context for data consumption. All communications in NDN are performed by using two distinct types of packets: Interest and Data [4], both of which carry a name that identifies a piece of content that can be transmitted in one Data, as shown in Figure 1.

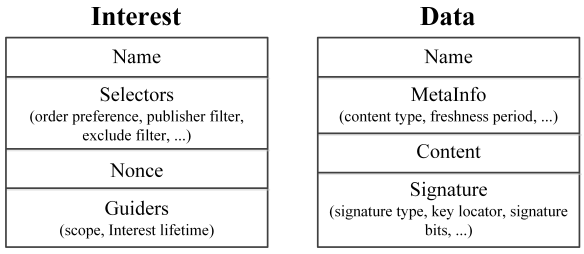


Fig. 1. Two types of packets in NDN [4].

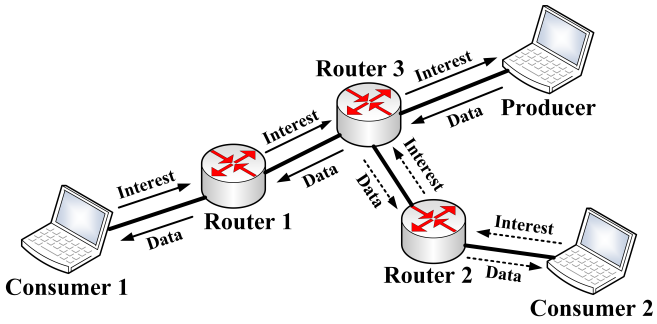


Fig. 2. Packet processing in NDN communication.

In NDN, every communication is driven by the receiving end, i.e., the data consumer. As illustrated in Figure 2, to receive the desired content, a consumer sends out an Interest with an identifying name to the network. Routers use this name to forward the Interest towards the producer(s). On the forwarding path, once the Interest reaches a node that has the requested Data, i.e., the Interest name is the same with the Data name or a prefix of the Data name [5], the node will send back the Data that contains both the name and the content, together with a signature by the producer's key. This Data follows the reversed path taken by the Interest to get back to the requesting consumer.

Differing from IP, multiple nodes interested in the same content can share transmissions over a broadcast medium using standard multicast suppression techniques in NDN, as both Interest and Data can identify the content exchanged by name. Therefore, compared with IP, NDN has to imply a substantial re-engineering of forwarding plane in the router, in which

This work was supported by the National Natural Science Foundation of China (Grant No. 61602346).

Z. Li, Y. Xu, L. Yan and K. Liu are with the school of Microelectronics, Tianjin University, Tianjin 300072, China (email: zli@tju.edu.cn; xuyaping_tju@tju.edu.cn; yanliu@tju.edu.cn; liukaihua@tju.edu.cn).

B. Zhang is with the Computer Science Department, University of Arizona, Tucson, AZ 85721, US (email: bzhang@cs.arizona.edu).

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/COMST.2018.2880444, IEEE Communications Surveys & Tutorials

TABLE I
NUMBER OF DATA STRUCTURE SCHEMES IMPLEMENTED IN THE FIB, PIT, CONTENT STORE AND UNIFIED INDEX.

Topic		FIB	PIT	Content Store	Unified Index	Total	
2011	Trie-based	2	0	0	0	2	2
	Hash table-based	0	0	0	0	0	
	Bloom filter-based	0	0	0	0	0	
	Skip list-based	0	0	0	0	0	
2012	Trie-based	1	1	0	0	2	8
	Hash table-based	1	0	0	1	2	
	Bloom filter-based	1	2	0	0	3	
	Skip list-based	0	0	1	0	1	
2013	Trie-based	1	0	0	0	1	6
	Hash table-based	2	1	0	0	3	
	Bloom filter-based	2	0	0	0	2	
	Skip list-based	0	0	0	0	0	
2014	Trie-based	2	0	0	1	3	8
	Hash table-based	0	1	0	0	1	
	Bloom filter-based	2	1	0	1	4	
	Skip list-based	0	0	0	0	0	
2015	Trie-based	5	0	0	0	5	12
	Hash table-based	4	0	2	0	6	
	Bloom filter-based	0	0	0	1	1	
	Skip list-based	0	0	0	0	0	
2016	Trie-based	5	1	1	0	7	12
	Hash table-based	0	0	0	0	0	
	Bloom filter-based	3	1	0	0	4	
	Skip list-based	0	0	1	0	1	
2017	Trie-based	0	0	0	0	0	6
	Hash table-based	2	0	0	0	2	
	Bloom filter-based	3	0	1	0	4	
	Skip list-based	0	0	0	0	0	
2018	Trie-based	2	0	0	0	2	4
	Hash table-based	0	0	0	0	0	
	Bloom filter-based	1	0	1	0	2	
	Skip list-based	0	0	0	0	0	
Total		39	8	7	4	58	

three tables including Content Store, Pending Interest Table (PIT) and Forwarding Information Base (FIB) are deployed [2]. Among them, Content Store caches the packet buffers to support in-network caching, PIT keeps track of the Interest forwarded upstream to ensure the return of the Data, and FIB stores the forwarding information for Interests to ensure proper packet forwarding.

It is widely believed that the design of scalable NDN forwarding plane must meet the basic challenge, namely variable-length and hierarchical names. In NDN, all lookup operations in forwarding plane reply upon finding the matching name prefix for a given name, which differs from longest prefix matching of IP [6] in two substantial ways. First, NDN names are explicitly hierarchical, consisting of a series of delimited components, while IP addresses can match a prefix at any bit position. Second, IP addresses are fixed length on the contrary, the NDN name lengths are variable, with no externally imposed upper bound [2].

This challenge in NDN brings a significant cost, both in router storage and packet processing. As mentioned before, the structure of name is so more complex than the IP address that the NDN forwarding plane calls for much more memory [2]. If data has to be stored in large and slow memories, it is really a heavy burden for the routers to complete fast forwarding. Consequently, there are two open questions: how to reduce the size of three tables in forwarding plane to store the packet information efficiently on the routers, and how to fast operate on them [7].

Since NDN was proposed in 2010, designing a scalable NDN forwarding plane has received considerable attention. Many data structures and algorithms are proposed to support efficient lookup mechanisms of name prefix in forwarding plane. Unfortunately, there is a lack of the comprehensive sketch about the requirements of NDN forwarding plane and the study on various schemes proposed. This lack leads to inefficiency in the research, even many schemes proposed have some serious mistakes.

Thus, more accurate requirements of NDN forwarding plane

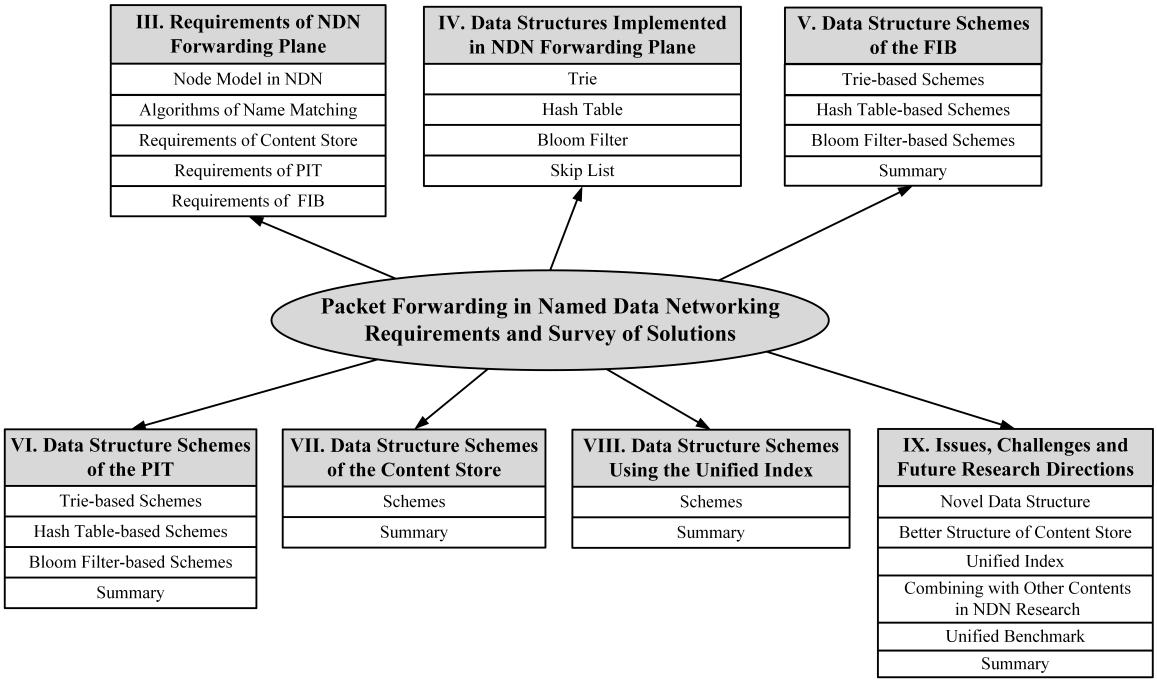


Fig. 3. Road map of packet forwarding in Named Data Networking requirements and survey of solutions.

are presented in this survey. Meanwhile, all the schemes proposed are discussed based on the data structure utilized. In addition, some issues, challenges and future research directions are also discussed. Especially, we make the following contributions:

1) The requirements of NDN forwarding plane are accurately presented in terms of the name matching algorithms performed in NDN router and the operation flow of name lookup, access frequency, table size and special property about Content Store, PIT and FIB. It is considered that four different algorithms have to be implemented in NDN forwarding plane to find the matching name prefix. For the access frequency, Content Store and PIT have high frequency for read operations and write operations, while FIB requires lots of read operations and few write operations. In addition, the size of FIB is larger than 10 million, but Content Store and PIT have different sizes in edge and core routers. Moreover, the special properties among three tables are also different. For example, Content Store must support the cache replacement policy, PIT has to perform timeout operation, and FIB has to support forwarding strategy.
2) All data structures used in NDN forwarding plane are analyzed in detail. Usually, trie [8], hash table [9], Bloom filter [10] and skip list [11] are utilized as the index in all schemes proposed. Trie is logical, so it can further reduce the memory consumption of the hierarchical names stored in NDN router, and naturally support finding the matching name prefix. Hash table can implement fast lookup, but the need that storing all name strings guarantees the forwarding correctness consumes more memory. Compared with the above two data structures, Bloom filter can not only achieve fast retrieval, but also reduce the memory consumption. However, the feature that Bloom filter cannot directly map the data in memory makes it not act as index alone. Moreover, skip list can usually support the replacement policy. But obviously, the lookup speed of skip list is relatively slow.
3) All schemes proposed for NDN forwarding plane are classified and introduced. The number of data structure schemes implemented in the FIB, PIT, Content Store and unified index in recent years is shown in Table I. Currently, there are 58 schemes aiming at the NDN forwarding plane, where 7, 8 and 39 schemes are respectively proposed for Content Store, PIT and FIB. Besides, 4 schemes implement the unified index which organizes some tables by using a single index. Moreover, the deployment of the data structure is different among the three tables. For FIB, trie is widely utilized to reduce the memory consumption, as FIB should support finding the longest matching name prefix. Meanwhile, Bloom filter is more often used in PIT, as a higher memory access frequency is required by PIT. As for Content Store, skip list is more suitable, because Content Store has to support not only fast name lookup but also the cache replacement policy.
4) Some issues, challenges and future research directions are also discussed at last. It is considered that designing a novel data structure to meet all requirements of the forwarding plane and studying on a better structure of Content Store play important roles, while discussing the necessity of a unified index, combining with other contents in NDN research and implementing a unified benchmark are also required in this domain.

The rest of the paper is organized as illustrated in Figure 3. Section II presents the related works and the differences of this survey. Section III interprets the node model and name matching algorithms of NDN forwarding plane in detail.

Meanwhile, a series of requirements of NDN forwarding plane are provided clearly. Section IV discusses the data structures implemented in NDN forwarding plane. From Section V to Section VIII, all the schemes since NDN was proposed are analyzed. Section IX discusses some issues, challenges and directions in future research. Finally, a brief conclusion is provided in Section X.

II. Related Work

Recently, some excellent survey works have been done on ICN. So we give this section to explain the differences and significance on all of them. In this section, the surveys from the whole field of ICN [1], [12], [13] and NDN [14] are firstly introduced, then the surveys of a particular domain in ICN, like caching [15]–[17], mobility [18]–[20], security [21], [22], energy efficiency [23]–[25] and others [26]–[30], are presented. Finally, we made a brief summary and emphasize the importance of this survey.

From the whole field of ICN, advantages of ICN and features of proposed ICN architectures are discussed in [1], [12], [13]. Among the three above, the survey proposed by Xylomenos et al. [1] identifies the core functionalities of ICN and describes in more detail and depth the most representative ICN architectures, including NDN. However, since NDN is only one of the described ICN architectures, the survey hardly dedicates more pages to describe the features and design details of NDN. Specifically, Section III (B) in [1] is the description of NDN, which includes a comprehensive diagram of NDN architecture and an explanation of the operation of its tables, but does not give a more accurate description of the name matching algorithms in different scenarios of name retrieval.

From the whole field of NDN, Saxena et al. [14], 2016 present in-depth research on NDN architecture, services and applications. In terms of services, the survey provides a comprehensive introduction to the major routing, caching, forwarding, security and mobility techniques proposed for NDN in recent years. However, the survey describes the features and schemes of NDN forwarding plane incompletely. Specifically, Section 4.3 in [14] is the description of NDN forwarding, including the data structure schemes of NDN forwarding plane and forwarding strategies. Thereinto, Table 3 and Table 4 discuss and compare the 4 data structure schemes used for PIT and the 9 used for FIB respectively, but do not contain all the representative schemes proposed so far, and do not give a more complete analysis of all the mentioned schemes.

In the domain of caching, the features and mechanisms of ICN caching are described in [15] and [16], while the simulation results and performance comparison of typical caching mechanisms are presented in [17]. However, these surveys only discuss various caching mechanisms, and do not give a more complete description of the data structure schemes that support the caching mechanisms.

In other particular domains in ICN, such as mobility, Feng et al. [18] present the problem statement for mobility in NDN and discuss NDN mobility support schemes based on the degree of separation of the identifier and locator, while Zhang et al. [19] focus on producer mobility solutions and articulate design tradeoffs of different approaches. The works that apply ICN communication paradigm into mobile wireless networks are summarized in [20] from the perspectives of architecture design and enabling technologies. In terms of security, the taxonomy and severity of ICN attacks as well as the relation between ICN attacks and ICN attributes are presented in [21], while the various issues and proposed solutions are analyzed from the three areas of security, privacy and access control in more detail in [22]. In terms of energy efficiency, enabling technologies for green ICN and some broader perspectives are discussed and explored in [23], while energy-efficient caching issues and techniques are mainly concerned in [24] and [25]. As for other domains, the naming and routing mechanisms in some ICN architectures are analyzed and compared in [26]. Liu et al. [27] focus on Information-Centric Mobile Ad Hoc Networks (ICMANET), describe the conceptual model and categorizing of content routing, and then summarize the representative routing schemes. Shang et al. [28] discuss the advantages of applying the NDN architecture to the Internet of Things (IoT) and how to achieve IoT framework functionality. Chen et al. [29] provide an introduction of NDN data transport process, indicate the differences with IP transport control, and then present recent proposals for NDN transport control. Ren et al. [30] point out congestion control issues of NDN and analyze the design methods of congestion control mechanisms as well as the existing typical algorithms and protocols. These above are all excellent surveys of a particular domain in ICN, but the requirements and research achievements of the forwarding plane, as well as the different design features and application performance of it in different network scenarios, are not described clearly or accurately.

In summary, the related surveys, including the surveys from the whole field of ICN and NDN and the surveys of a particular domain in ICN, are presented in detail. Through the analysis above, the comprehensive sketch about NDN forwarding plane is still imperfect. Firstly, the surveys from the whole field of ICN and NDN always describe the forwarding plane sketchily and broadly. For example, [1] does not give a more accurate description of the name matching algorithms in different scenarios of name retrieval, and [14] does not cover all the representative schemes of the forwarding plane proposed so far. Secondly, the various surveys of a particular domain in ICN lack attention to the domain of the forwarding plane. For example, they focus on some domains like caching [15]–[17] and mobility [18]–[20], but the requirements and research achievements of the forwarding plane are not presented clearly so far. Thus, more accurate requirements of NDN forwarding plane, including the name matching algorithms performed in NDN router and the operation flow of name lookup, access frequency, table size and special property about Content Store, PIT and FIB, are presented in this survey. Meanwhile, all the data structure schemes proposed are discussed in more detail, and some issues, challenges and future research directions are also indicated. Specifically, compared with [1], we describe and illustrate the four name matching algorithms in five scenarios of name retrieval more accurately. Compared with

TABLE II
REQUIREMENTS OF NDN FORWARDING PLANE.

Table	Matching Algorithm		Access Frequency	Special Property
	Interest	Data		
Content Store	ASNM	ENM	Lots Read/Write	Supporting Cache Replacement Policy
PIT	ENM	ANPM	Lots Read/Write	Supporting Timeout Operation
FIB	LNPM	N/A	Lots Read, Few Write	Supporting Forwarding Strategy

N/A: Not applied

[14], data structure schemes of the PIT and FIB presented in our survey are 2 and 4 times as many as those presented in [14] respectively. We also present the data structure schemes of the Content Store and using the unified index, and analyze all the schemes in more detail and depth.

III. REQUIREMENTS OF NDN FORWARDING PLANE

Logically, NDN forwarding plane contains three tables: Content Store, PIT and FIB to support the forwarding based on name and sharing transmissions when multiple nodes are interested in the same content, which is so different with IP. Reviewing researches on the requirements of NDN forwarding plane [7], [31]–[40] since NDN was proposed, traditionally, there are two basic requirements in NDN forwarding plane. First, it has to implement fast update operation. When Interest/Data arrive at NDN routers and the routing protocols recompute FIB, the three tables must support fast insertions, deletions and modifications. Second, NDN forwarding plane must support very high capacity. Suppose a system is provisioned with packet buffers 2 KB in size with 2 GB of total buffer space, which means Content Store capacity for 1 million distinct packet names. If an NDN name is 200 bytes in length, it needs a further 200 MB to store the names. In this example, the requirements of an NDN forwarding device perhaps include 10s of gigabytes of main memory and a prefix table several gigabytes in size [7].

To the best of our knowledge, this section firstly presents the NDN node model and four different algorithms of name matching performed in NDN router. Then the requirements of Content Store, PIT and FIB are introduced in detail by four aspects, which include the operation flow of name lookup, access frequency, table size and special property. The main requirements of NDN forwarding plane are shown in Table II.

A. Node Model in NDN

In NDN, the communication is driven by consumer via the exchange of Interest and Data [4]. The consumer puts the name of the desired content into an Interest and sends it to the network. Routers forward the Interest towards the data producer(s) with this name. On the forwarding path, once the Interest reaches a node that has the requested Data, i.e., the Interest name is the same with the Data name or a prefix of the Data name [5], the Data which contains both the name and the content is sent back following the reversed path created by the Interest back to the consumer, as illustrated in Figure 2.

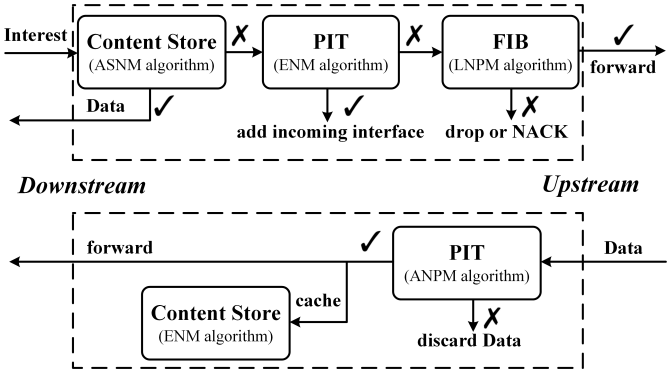


Fig. 4. NDN packet operational flow in the forwarding plane.

Given that there are no source or destination addresses in an NDN packet, the router of NDN makes forwarding decisions by the name rulesets stored in Content Store, PIT and FIB, which are deployed on the router of NDN. Content Store essentially stores packet buffers, which is equipped with a lookup structure and supports cache replacement policies. PIT, which keeps track of the Interest forwarded upstream, provides two major functions, namely Interest aggregation and Data multicast. FIB stores the forwarding information for Interests, and forwards Interests toward potential source(s) of matching Data. Figure 4 illustrates the NDN packet operational flow in the forwarding plane.

When an Interest is received by the NDN router, the router first checks whether there is a matching Data in its Content Store. If a match is found, the Data is sent back to the incoming interface of the Interest. If not, the Interest name is checked against the entries in PIT. If the name has existed in PIT already, the router simply adds the incoming interface of this newly received Interest to the existing PIT entry. If the name does not exist in PIT, the Interest is added into PIT and further forwarded based on the forwarding information stored in FIB [2].

When a Data arrives at the NDN router, its name is used to query PIT. If a matching PIT entry is found, the router sends the Data to the interfaces from which the Interest was received, caches the Data, and removes PIT entry. Otherwise, the Data is unsolicited and discarded [2].

B. Algorithms of Name Matching

In NDN, the networks should retrieve data by name prefixes [2]. For example, when a consumer application starts, it may not know the complete data name, as some parts of the name cannot be guessed, computed, or inferred beforehand. In these

cases, a consumer can send an Interest carrying a prefix of the data name, and the network will retrieve one data under the prefix. For an incoming Interest which only carries a prefix of the data name, e.g., */ndn/UA/videos/today.mp4*, the network can return any Data whose name is covered by the Interest name, such as */ndn/UA/videos/today.mp4/segment1* and */ndn/UA/videos/today.mp4/segment2*. This is a name discovery mechanism by which consumers can learn more about the names of available data in order to construct complete names for future Interests.

To support the mechanism that retrieves data by name prefixes, four different algorithms of name matching should be implemented in five scenarios of name retrieval on NDN router, where the five scenarios include Interest matching in Content Store, Interest matching in PIT, Interest matching in FIB, Data matching in PIT and Data matching in Content Store, as illustrated in Figure 4. In the following, the four algorithms of name matching are presented in detail based on the different scenarios.

1) Interest matching in Content Store: After NDN router receives an Interest, the scenario firstly occurs in Content Store, which is responsible for querying whether there is a matching Data. Here, the Data may match with exact content requested by this Interest, or a part of requested content. That is, the name of the matching Data may be same as the Interest name, or start with this Interest name. Therefore, the name matching algorithm named All Sub-Name Matching (**ASNM**), which is defined as querying any Data whose name starts with this Interest name, is implemented in this scenario. Owing to utilizing the ASNM, the router can return any eligible Data related with this Interest as soon as possible. Meanwhile, this algorithm also helps the consumer to learn more information about the available data names and construct complete names for future Interests.

2) Interest matching in PIT: This scenario occurs in PIT to determine whether this Interest needs to be aggregated. The Interest aggregation is performed only if the name of incoming Interest is same as the name stored in PIT entry. Hence, in the scenario, the name matching algorithm named Exact Name Matching (**ENM**), which is defined as matching an entry with the exact name of Interest, is applied. Because of using the ENM, PIT can aggregate Interests exactly thus reducing the number of Interests forwarded by the router.

3) Interest matching in FIB: If the forwarding information of next hop for an incoming Interest needs to be retrieved, the scenario occurs in FIB. In common with IP, FIB in NDN needs to find out the forwarding information of the longest matching name prefix. Thus, the name matching algorithm, namely Longest Name Prefix Matching (**LNPM**) whose definition is just same as the longest prefix matching of IP, is performed with the requested names as the lookup keys. By utilizing the LNPM, FIB can find the most accurate forwarding information for the Interest as far as possible.

4) Data matching in PIT: After NDN router receives the Data returned, the scenario firstly occurs in PIT, in which the interface(s) of next hop for the Data can be found. Given that the Data can satisfy all pending Interests whose names are the same with the prefixes of this Data name, the name matching algorithm called All Name Prefix Matching (**ANPM**), which is defined as querying any pending Interests whose name is a prefix of Data name, is implemented in this scenario. Through applying the ANPM, PIT can find the incoming interfaces of corresponding pending Interests to achieve Data multicast.

5) Data matching in Content Store: After the incoming Data has been forwarded to corresponding interfaces based on PIT, the scenario occurs in Content Store to verify if this Data needs to be cached. If the Data with the same name as incoming Data does not exist in Content Store, this Data will be cached. Therefore, **ENM** is utilized with the Data name as the lookup keys, which is same as the name matching algorithm used in the scenario of Interest matching in PIT. By using the ENM, Content Store can cache the incoming Data to satisfy the future Interests requesting same content as far as possible.

C. Requirements of Content Store

Content Store is the router's buffer memory, which caches arriving Data to satisfy future Interests that request same Data. In general, Content Store consists of the index and packet buffer, where every index entry records the Data name and address pointing to the packet buffer, as well as the packet buffer is applied to store the cached Data.

For the name lookup, when an Interest with name */ndn/UA/videos/today.mp4* arrives at some interface of NDN router, the name matching algorithm **ASNM** is performed in Content Store for any Data whose name starts with this Interest name, and the lookup could match Content Store entries with the name such as */ndn/UA/videos/today.mp4* and */ndn/UA/videos/today.mp4/segment1*. If one or more matches are found in Content Store, the optimal match is determined by a selection procedure. Given that the Data with the name */ndn/UA/videos/today.mp4* is the "best" match, this Data will be sent out to the interface which the Interest arrived at. On the arrival of a Data with name */ndn/UA/videos/today.mp4*, the name matching algorithm **ENM** for the name */ndn/UA/videos/today.mp4* and the cache replacement policy determine together if this Data should be cached. During the insertion, a packet needs to be evicted to make space once Content Store is full.

Virtually, every incoming Interest or Data has to access Content Store, so frequent memory accesses occur in Content Store. Given that Content Store acts as a part of the operation flow of NDN router, the line rate operations, which include cache replacements and name lookups, should be satisfied.

For the size of Content Store, a recent study reports that the benefits of ubiquitous caching in information-centric networking can be largely realized by caching content in edge networks. As a result, the size of Content Store in core routers can be small, and the routers could even operate without a Content Store. But generally, a small Content Store is still preferred because popular content distribution, such as live streaming, can be supported more efficiently [41]. However, as investigated in [42], a typical end host has small FIB and PIT, but a large Content Store on the NDN testbed. If 1 GB memory is allocated to Content Store, assuming an average

Data size of 4 KB, Content Store could still contain 262144 entries.

Moreover, although Content Store stores arriving Data as long as possible, it is a temporary cache rather than a persistent Data storage, and its memory usage or entry count should be sure to stay within the capacity limit. Hence, the cache replacement policy must be implemented to determine which content to replace first at a time when the memory usage reaches the maximum capacity, which is usually classified into three main categories as order-based policy where the first cached content is replaced first, like First In First Out (FIFO), popularity-based policy where less popular content is replaced first, like Least Recently Used (LRU) and Least Frequently Used (LFU), and priority-based policy where low priority content is replaced first [14]. Based on the fact that the cache replacement policy is various and the data structure used in Content Store must support fast replacement operation under the utilized cache replacement policy, the impact of caching policy on packet forwarding has been concerned gradually, such as [43]–[45].

D. Requirements of PIT

PIT stores all the Interests that have been forwarded but not been satisfied yet. Each PIT entry records the data name carried in the Internet, together with its incoming and outgoing interface(s) [4].

For the name lookup, when the Interest with name */ndn/UA/videos* is received, PIT uses the name matching algorithm **ENM** to create or update an PIT entry with name */ndn/UA/videos*, which is the tuple *<name, list interfaces, list nonces, expiration>* [7]. For an incoming Data with name */ndn/UA*, the name matching algorithm **ANPM** is firstly performed in PIT to find all pending Interests whose name is same as the prefix of Data name. This would match PIT entries with the name /, */ndn*, and */ndn/UA*.

Given that a lookup has to be performed in PIT no matter whether the incoming packet is Interest or Data, PIT is highly dynamic. As reported in [31], in a more realistic scenario where Data is correctly transmitted as response to the Interest, flow balance or a load of 50%, the frequency of PIT's operations is about 6 Million per second.

Meanwhile, PIT design has been recognized as a flow table management problem [32]. As highlighted in [31], [35], [36], PIT has to contain 1 million entries for 10 Gbps gateway trace and 1.5 million entries for 20 Gbps. It is also suggested in [39], [46] that the core and edge routers should have different number of entries. The size of PIT at core routers does not exceed 2 million entries even in the worst case when all traffic is bottlenecked outside the ISP network [39].

At last, PIT entries are deleted after a timeout to avoid the size of PIT exploding over time. A timeout also enables protection against simple attacks that could overflood a router's PIT [47]. It follows that each PIT entry has to be associated to a timer, and mechanisms to detect timer expiration and to purge expired entries are needed [31].

E. Requirements of FIB

FIB is utilized to forward Interest toward potential source(s) of matching Data. It is almost identical to FIB in IP except it allows for a list of forwarding interfaces rather than a single one. For an Interest, the list of forwarding interfaces can be found in corresponding FIB entry, which are an important reference for forwarding.

For the name lookup, only Interest that waits to be forwarded needs to be queried in FIB. Given that the names in NDN are hierarchically structured, the Interest should be forwarded based on the results of the name matching algorithm **LNPM** with the requested names as the lookup keys [48]. Therefore, FIB should be keyed by name prefixes.

Meanwhile, the number of entries in FIB of NDN is expected to be much larger than that of IP. Although the number of entries in FIB is determined by the namespace design and the effectiveness of name prefix aggregation, more than 10 million domain names are expected to handle a network on the scale of the current Internet [48].

For the access frequency, FIB requires frequent read operations and fewer write operations, as FIB is mainly responsible for reading the forwarding information of next hop for an Interest. Furthermore, in order to achieve efficient forwarding, FIB has to support forwarding strategy, which can retrieve the entry of the longest matching name prefix from FIB, and decide when and where to forward the Interest.

IV. Data Structures Implemented in NDN Forwarding Plane

For the design of NDN forwarding plane, the main point is to design the index of Content Store, PIT and FIB, which influences the lookup speed, memory consumption, as well as the possibility of supporting the forwarding strategy. Currently, trie, hash table, Bloom filter and skip list are the data structures that are widely used as the index in all schemes proposed. Meanwhile, given that the four data structures have different features, these structures sometimes are combined with the additional data structures to make up for the lack of functionality no matter in Content Store, PIT or FIB. For example, Bloom filter has to be combined with hash table or trie to map the data stored in the memory. Additionally, the deployment of these data structures is different among the three tables. For FIB, trie is more often utilized as the index. In PIT, Bloom filter is inclined to be chosen. As for Content Store, skip list is more appropriate to satisfy its requirement for the cache replacement policy. Furthermore, for the data structure design of index, there is usually a trade-off between lookup speed and memory consumption.

In the section, the four data structures are described and analyzed. For each data structure, its definition and interpretation are given first, and then the characteristics in the design and deployment are indicated and analyzed respectively.

A. Trie

Trie [8] is a data structure used to store and retrieve information, which comprises of function-argument or item-term pairs-information conventionally stored in unordered

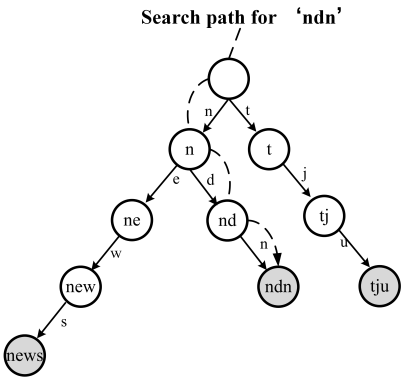


Fig. 5. Basic structure of trie.

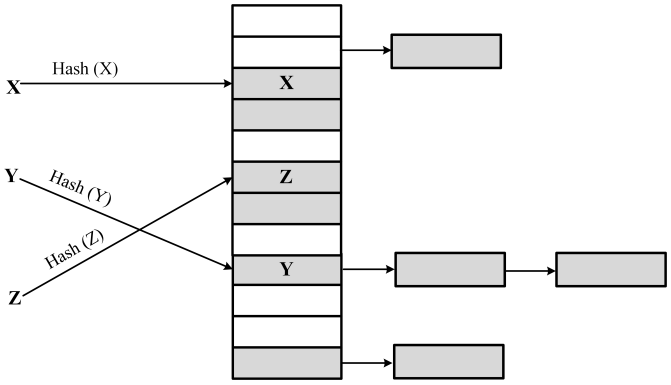


Fig. 6. Basic structure of hash table.

lists, ordered lists, or pigeonholes. Generally, trie supports three basic operations, namely lookup, insertion and update. Moreover, trie is convenient in handling elements of diverse lengths, and can utilize redundancies in the information stored.

Figure 5 illustrates a basic data structure of trie, where three elements, namely *ndn*, *news* and *tju*, are organized according to the characters. When *ndn* is searched, the root is firstly queried. And then the search path is based on *n*, *d*, and *n*, as shown in the figure. Therefore, the time complexity of lookup and insertion in the trie is related with the length of elements, namely O(n). Meanwhile, the trie utilizes the public prefixes to reduce the lookup time, which means trie trades space for time, thus its space complexity is huge.

Trie is an ordered tree-like data structure used to deal with string lookup, where the keys are usually strings. Based on the logical characteristics of trie, trie can reduce the memory consumption of the hierarchical names and naturally support the name prefix lookup, consequently trie as the index is applied in NDN forwarding plane. When trie is utilized as the index, how to design its granularity is the first consideration, with the reason that the granularity of trie influences not only the memory consumption, but also the types of lookup unit during the name retrieval, where there are mainly four types of granularity, namely **component**, **encoded component**, **character** and **binary**. Firstly, using the component of name as the granularity of trie is a natural basic proposal, e.g., the name */ndn/UA* is divided into *ndn* and *UA* to store. Secondly, encoded component can also be implemented as the granularity, e.g., the name */ndn/UA* is encoded as /1/2 and divided into 1 and 2 to store. Therefore, when using encoded component as the granularity, the size of trie is obviously compressed by encoding longer components as shorter codes, but the encoding process may consume some CPU cycles and make the lookup speed lower. Thirdly, character as the granularity in trie is also common, e.g., the name */ndn/UA* is treated as *n*, *d*, *n*, */*, *U* and *A* to store. It achieved further memory compression compared with component as the granularity because different names are better integrated together, e.g., if another name is */ndn/UCLA*, *ndn/U* is the same thus only *CLA* needs to be stored when using character as the granularity, while the whole second component *UCLA* needs to be stored when using component as the granularity. Finally, Each name is treated as a binary string when using binary as the granularity, e.g., the name */ndn/UA* is treated as 01101110..., where the finer granularity enables greater integration of different names compared with the character as granularity, thus can vastly compress memory. In addition to the four types above, some special designs use **name string** and **name prefix** as the granularity, which are also viable and valuable in some cases.

Given that the logical characteristics of trie can reduce the memory consumption of the hierarchical names stored in NDN router and naturally support finding the matching name prefix, there are a total of 22 schemes proposed for NDN forwarding plane with this data structure, where the most are proposed for FIB. The reason is that FIB has more than 10 million entries and needs to support the name matching algorithm LNPM. If trie is used to store lots of names, the lookup performance based on LNPM is undoubtedly improved owing to the logicality of trie. Meanwhile, the memory cost consumed by name storage is also effectively reduced by aggregating redundant components or bits in the names. However, that the lookup speed is relatively slow is the obvious drawback of trie. To overcome this, exploring how to handle the depth of trie to guarantee the lookup efficiency may be of some help, such as aggregating some nodes to reduce the depth of trie.

In the following sections of schemes, the trie-based schemes are further analyzed according to the four types divided by the granularity, which are component, encoded component, character and binary.

B. Hash Table

Hash table [9] is a compact data structure that stores more key-value pairs. Generally, hash table is composed of a dictionary or associative array and a mapping function named hash function, where the hash function can compute the keys to values. When a key needs to be inserted, hash table utilizes this hash function to map the key into a slot corresponding to the value and stores the key.

In the Figure 6 [9], the basic structure of hash table storing multiple elements is shown. When searching element X, X as the key is input into the hash function, and the hash value calculated serves as the address of slot. Obviously, the time complexity of hash table is constant, i.e. O(1), whether it implements lookup, insertion or deletion operations. However,

due to the hash collision that the hash values of some elements are same, the chained linked list is used in hash table to deal with the collision, but also increases the lookup time. To reduce the hash collisions, hash table generally reserves enough memory space, therefore it has high the space complexity.

Given the high frequency for processing operations in NDN forwarding plane, hash table is usually a basic data structure utilized in the forwarding plane because of its advantage of fast lookup. Just like the traditional usage of hash table, the first consideration is which hash function should be choose to improve lookup performance and which type of hash table should be choose to deal with the false positive and reduce the memory consumption, but what is more important in NDN forwarding plane is how to support the algorithms of name matching described in Section III. Mainly, there are two methods. The first one namely **optimal linear search** starts name lookup from the longest length of name prefix and then decreases progressively by component granularity, which is a simple original method and similar with the longest prefix matching of IP. The second method namely **random search** begins with the name prefix of a certain length which is usually determined by the statistical result of names, and then the matching result determines the length of name prefix of next search. This method is aimed at reducing the number of hash lookups by shortening the search path, thus increases the lookup speed, which is more suitable for achieving the name matching algorithm LNPM. That is to say, if the match is found, the longer prefixes are required to be searched to find the longest matched name prefix; otherwise, the shorter prefixes are required to be searched to find a name prefix that can be matched.

Considering the advantages of fast lookup, there are a total of 14 schemes proposed for NDN forwarding plane with hash table by now. Nevertheless, hash table as the index has to store the whole name string in the entry to ensure the accurate forwarding, which consumes more memory. Aiming at this drawback, replacing the name string by using a fingerprint or a hash value may be helpful to reduce the memory consumption.

In the following sections of schemes, the hash table-based schemes are further analyzed according to the two types classified by the method that implements name lookup, which are optimal linear search and random search.

C. Bloom Filter

Bloom filter [10] is a space-efficient probabilistic data structure that supports set membership queries. The data structure was conceived by Burton H. Bloom in 1970. The structure offers a compact probabilistic way to represent a set that can result in false positives, but never in false negatives. This makes Bloom filter useful for many different kinds of tasks that involve lists and sets. The accuracy of Bloom filter depends on the size of the filter, the number of hash functions used in the filter, and the number of elements added to the filter. The more elements are added to Bloom filter, the higher probability that the query operation reports false positives [49].

The overview of a Bloom filter is presented in Figure 7 [49], in which the original state of each bit is 0. Bloom filter in

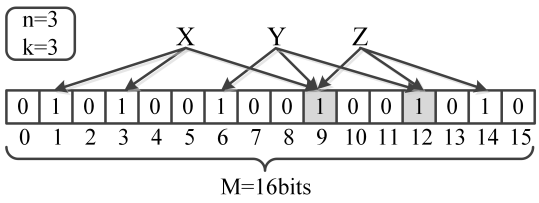


Fig. 7. Basic structure of Bloom filter.

Figure 7 is composed of 16 bits, and three elements have been inserted, namely X, Y and Z. Each of the elements have been hashed using three hash functions to bit positions in Bloom filter, which the corresponding bits have been set to 1. When an element needs to be checked, the element will be hashed by the same three hash functions. If the three bit positions are 1, the element is in the Bloom filter, otherwise, is not. As similar with hash table, whether Bloom filter performs lookup or insertion operations, the time complexity is O(1). Although Bloom filter is essentially a bit string, the space complexity for a set with n elements is also O(n).

Bloom filter plays an important role in NDN forwarding plane, for it can not only achieve the fast retrieval, but also reduce the memory consumption. When Bloom filter is utilized as the index in NDN forwarding plane, the problem that Bloom filter can only determine whether an element is in the set but not locate its memory address needs to be resolved, where there are mainly two methods. The first one namely **interface deployment** assigns a Bloom filter to each interface. Each of the Bloom filters operates independently and only records the elements of its own interface. If a match is found, the interface is located according to which interface the Bloom filter is disposed on, thus eliminating the need for additional locating element. The second one namely **composite structure** combines Bloom filter with an additional structure like trie or hash table, where the Bloom filter is used to determine the existence of element and the additional structure is arranged to locate element.

Given its good performance in fast retrieval and memory compression, 20 schemes with Bloom filter are proposed for NDN forwarding plane statistically. In addition, as for the deployment of data structures in the three tables, Bloom filter is more often utilized in PIT, as a higher memory access frequency is required by PIT. However, it is still a major drawback that Bloom filter cannot locate the address alone, which means better solutions need to be found.

In the following sections of schemes, the Bloom filter-based schemes are further analyzed according to the two types classified by the method that locates memory address of the element, which are interface deployment and composite structure.

D. Skip List

Skip list [11] is a linked list structure with parallel. That is, skip list is constructed in form of layers, where an ordered linked list acts as the bottom layer and every higher layer assists the ordered lists below to perform fast lookup. Moreover,

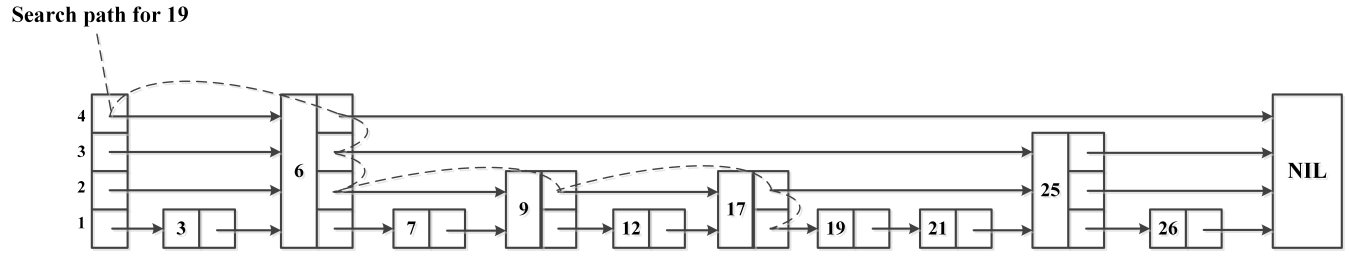


Fig. 8. Basic structure of skip list.

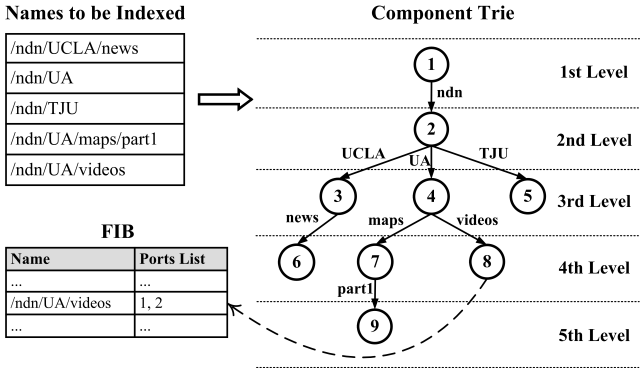


Fig. 10. Basic structure of component trie.

an element in layer *i* appears in layer *i+1* with some fixed probability. Compared with the linked list, skip list can skip over intermediate list nodes during the name lookup. Hence, the lookup speed of skip list is more faster than the linked list. Furthermore, owing to the order connection of nodes, skip list can implement fast and simple insertion and deletion operations in any node.

Figure 8 shows the structure of a original skip list and steps involved in skip list search. As illustrated in the figure, a search for the target element 19 starts from the head node in the top list, and performs horizontally until the current node is greater than or equal to 19. When the current node is greater than 19, the process is repeated after returning to the previous node and vertically dropping down to the next lower list [50]. When the current node is equal to 19, the target is found. According to the searching procedure, the time complexity of lookup, insertion or deletion operation in skip list is O(logn). However, skip list obviously exchanges space for time, thus its space complexity is related with the number of elements inserted, namely O(n).

Given that the order of data storage is preserved by using the linked list, skip list itself can support the replacement policy, which is more appropriate as the index of Content Store to satisfy fast update operation and the cache replacement policy simultaneously. So far, 2 schemes based on this data structure are proposed, where the one [32] is a standard implementation of skip list and the other [50] changes the lookup order of skip list, which improves the lookup speed. However, due to the use of linked list structure, the lookup speed of skip list is still slower compared with hash table and Bloom filter, which needs to be further improved.

V. Data Structure Schemes of the FIB

From this section, 58 data structure schemes aiming at the NDN forwarding plane are introduced, where 7, 8 and 39 schemes are respectively proposed for Content Store, PIT and FIB. Besides, 4 schemes implement the unified index which organizes some tables by using a single index. To introduce these schemes more logically, they are successively analyzed on the basis of the data structure used, namely trie, hash table, Bloom filter and skip list, as mentioned in Section IV. Given the largest number of data structure schemes of FIB, it is introduced first of all. And the data structure schemes of PIT, Content Store and the unified index are presented in the following sections.

In the section, the data structure schemes proposed for FIB are introduced, which are classified into three types, namely trie-based, hash table-based and Bloom filter-based, as illustrated in Figure 9. Further, the schemes with trie are divided into component, encoded component, character and binary, while the schemes with hash table are divided into optimal linear search and random search to analyze. Besides, the schemes with Bloom filter are classified into interface deployment and composite structure. At last, a summary about all the data structure schemes proposed for FIB is presented, which gives the respective features of trie-based, hash table-based and Bloom filter-based schemes and the better data structure schemes so far.

A. Trie-based Schemes

Trie is widely applied in FIB. The reason is that trie can store the logic relation of names and better support the name matching algorithm LNPM. As shown in Table III, the amount of schemes with this data structure has reached 18. Given that NPT [51], NCE [55], MATA [59], Binary Patricia Trie [46] and TB^{2}F [65] are the classical ones for component, encoded component, character and binary, these schemes are emphatically introduced, while the others are briefly discussed.

1) NPT: Name Prefix Trie (NPT) is a basic scheme with trie to manage vast names in NDN forwarding plane [51]. NPT is the most basic component trie, whose basic structure is illustrated in Figure 10. Each name component is represented as an edge and a lookup state is represented as a node of trie. In addition, there is the concept of level, i.e., one node and the edges organized by the node belong to one level. As shown in Figure 10, the 1st level is composed of the root and the component *ndn*. During the lookup for a name, the first component of the name is firstly queried from the root. If

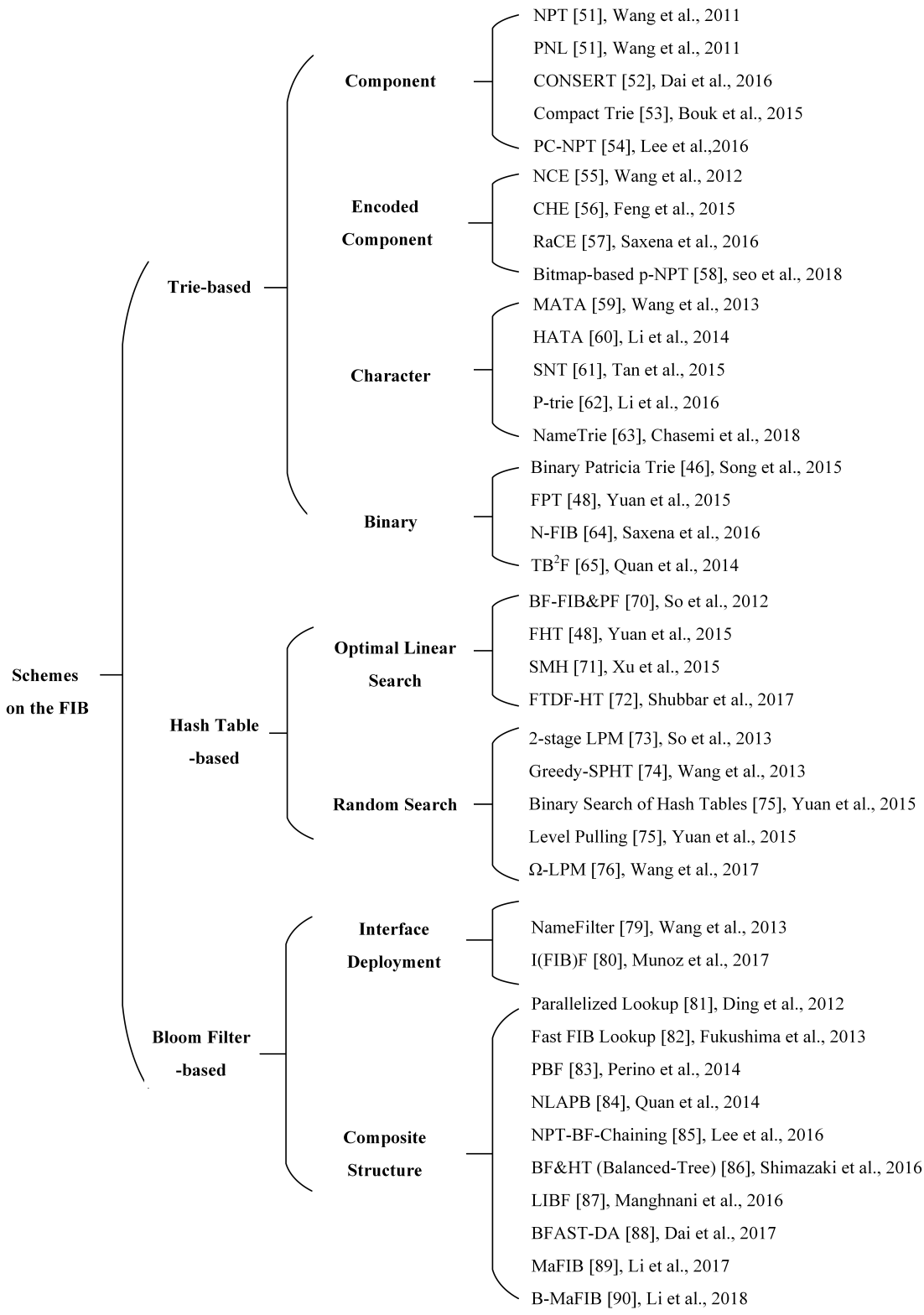


Fig. 9. Classification of data structure schemes on the FIB.

there is a match, the lookup state is transferred from the root to the node of next level and the next components continue to be queried iteratively. When the failing match or leaf node is encountered, the lookup finishes and the forwarding information of the longest matching name prefix is returned. For instance, when the name */ndn/UA/videos* is given, the first component *ndn* is firstly checked in the 1st level and the match is found. Then the lookup state is transferred from the state 1 to the state 2. The second component *UA* is matched in the 2nd level then the lookup state is transferred to the state 4.

TABLE III
TRIE-BASED SCHEMES ON THE FIB.

UNL	Scheme Name	Key Feature	Pros	Cons
Component	NPT [51] Wang et al.	Basic name prefix trie	Storing names logically	Depth of trie influences the lookup speed
	PNL [51] Wang et al.	Hardware parallelism	Improving the lookup speed of NPT	Duplicated states increase the memory consumption
	CONSERT [52] Dai et al.	Removing the redundancy	Minimizing the number of name prefixes	Different candidaes induce the different output routing tables
	Compact Trie [53] Bouk et al.	Novel naming method, merging nodes	Reducing the impact of redundant information at memory	Impacting the accuracy for longest name prefix matching
	PC-NPT [54] Lee et al.	Path compression	Reducing the average number of node accesses	Depth of trie influences the lookup speed
Encoded Component	NCE [55] Wang et al.	Encoding components	Shrinking FIB size than NPT	The mapping processing affects the lookup speed
	CHE [56] Feng et al.	Component hash encoding	Performing better than NPT at memory consumption	Inflexible encoding method
	RaCE [57] Saxena et al.	New encoding method, path compression	Lower memory consumed than NCE	The mapping processing affects the lookup speed
	Bitmap-based p-NPT [58] Seo et al.	New encoding method, Bitmap	Performing better than NPT	There may be hash collisions for hashBrief
Character	MATA [59] Wang et al.	Multiple ATA	Outperforming ATA in lookup speed	Memory utilization is not proper
	HATA [60] Li et al.	Parallelism	Improving memory utilization than MATA	Some stages of a big HATA must be moved to external RAMs
	SNT [61] Tan et al.	Splitting trie, Using CithHash in transition	Lower memory consumed than MATA	The different datasets have different split length
	P-trie [62] Li et al.	Dividing trie nodes into three types	The throughput is three times higher than MATA	Auxiliary forwarding information consumes more memory
	NameTrie [63] Ghasemi et al.	minASCII encoding	Performing better than NPT	The mapping between name and codes induces the complexity
Binary	Binary Patricia Trie [46], Song et al.	Binary tokens of names	Minimizing the impact of redundant information at memory	Forwarding loop may occur for speculative forwarding
	FPT [48] Yuan et al.	Utilizing fingerprints to construct Patricia trie	Higher lookup speed than binary Patricia trie	Fingerprints may occur collision
	N-FIB [64] Saxena et al.	Aggregate names	Lower memory consumed than binary Patrcia trie	The depth of trie can be optimized by load balancing hash
	TB^{2}F [65] Quan et al.	Dividing name into two segments	Minimizing the drawbacks of CBF and tree-bitmap	The different datasets have different split-level parameters

UNL: Unit of Name Lookup

The lookup continues until the leaf-node, namely state 8, is found.

Obviously, NPT can reduce memory consumption. However, as the most basic component trie, the depth of NPT is relatively high, which negatively influences the lookup speed.

2) PNL: To boost the lookup speed of NPT, Wang et al., 2011 proposed Parallel Name Lookup (PNL) [51] by using the router hardware parallelism. In this scheme, the lookup states of different levels in NPT are grouped according to the access probability of each state and stored into multiple physical modules, each of which manages a group. If several states simultaneously try to transfer to states of the same module, the conflict occurs. To reduce the conflicts, some states whose access probability exceeds a specified threshold are duplicated into other physical modules. If the above situation arises, these states can transfer to different modules rather than one same module, and then the lookup continues.

Based on the physical parallelism, PNL obviously improves

the lookup speed of NPT. The overall memory consumption still needs to be reduced due to the duplicated states.

3) CONSERT: Additionally, in order to accelerate the lookup, Dai et al., 2016 proposed CONSERT to minimize the number of name prefixes in component trie [52]. In CONSERT, the forwarding information of each non-leaf node is moved to a newly added child. Moreover, for each node with children, the most popular forwarding information among all the children is treated as this node's forwarding information. Meanwhile, the leaf node containing same forwarding information with its parent and its edge are removed from trie. To avoid losing the information of nodes removed, the information is stored into a set, in which the component is rechecked if the matching fails at one node. During a lookup, if the last matched node of the name contains the newly added child, the forwarding information of this child is returned as the final matching result.

To evaluate the performance of CONSERT in terms of the memory consumption and lookup speed, the experiments are implemented on a platform equipped with two 6-core Intel Xeon E5645 of 2.4 GHz and RAM with DDR3 ECC 48 GB. In the memory consumption, CONSERT consumes 75.651 MB memory for a dataset with 1 million prefixes. In the lookup speed, CONSERT can reach 0.90 million packet per second (MPPS) with a single thread [52]. From the experimental results, CONSERT significantly improves lookup speed compared with NPT by reducing the number of name prefixes. Given that the final forwarding information of a name prefix is chosen from a set of candidate forwarding information, CONSERT may produce different output routing tables for a given input one [52].

4) Compact Trie: In addition, given that reducing the depth of trie can improve lookup speed of NPT, Bouk et al. 2015 proposed Compact Trie [53], which implements a novel naming method to reduce the depth of trie. In the naming method of Compact Trie, the name is split into three parts, namely the scheme part, the prefix part and the Hash part. Here, the scheme part represents the name is a content or protocol, the prefix part confirms the content originating node, as well as the Hash part is used to record the hash value of the name. Based on this naming method, any node with one child in Compact Trie is merged into a single node with its child, where the single node stores several components and connects with the next node by using a pointer. When querying the longest matching name prefix for a name, multiple components of the name are matched with the components stored at a node. If there is a match, the next node is queried. Otherwise the lookup process terminates.

For the performance evaluation of this scheme, the experiments are constructed on the virtual machine with one processor of Intel Core i5-4670 CPU at 3.4 GHz and memory of 3.9 GiB. The results show that for the dataset with 0.175 million prefixes, the memory is consumed by 10.88 MiB and the insertion speed reaches 0.286 MPPS [53]. By aggregating the components, Compact Trie can reduce the depth of trie thus achieving high lookup speed. In Compact Trie, the longest matching name prefix may be shorter than the component trie without compression, which may influence the forwarding accuracy.

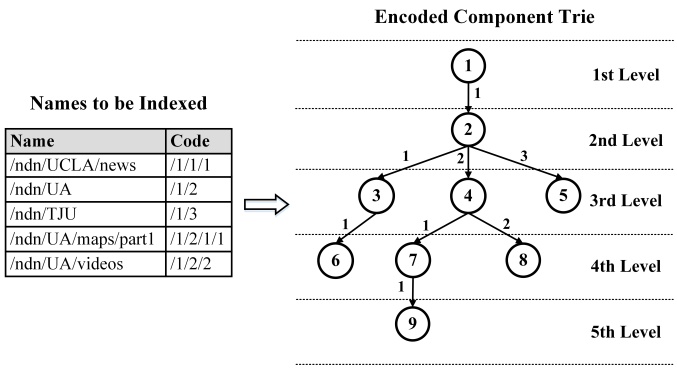


Fig. 11. Basic structure of encoded component trie.

5) PC-NPT: Similarly, Lee et al., 2016 proposed a path-compressed Name Prefix Trie (PC-NPT) [54] to optimize the lookup speed of NPT. In PC-NPT, each empty node, which has only one child and no forwarding information, is merged with its child and the components of merged edges are also combined into one component string stored on one edge. At the same time, the corresponding edge is assigned a skip value to indicate the number of merged children.

In order to evaluate the compression performance of this scheme, the experiment is implemented. And the result shows that the number of empty nodes can be reduced by more than 97% compared with NPT [54]. Obviously, PC-NPT reduces the average number of node accesses owing to merging empty nodes, and improves the lookup speed.

6) NCE: In order to reduce the memory consumption of NPT, Wang et al. 2012 proposed the Name Component Encoding (NCE) scheme to build an encoded component trie named Encoded Name Prefix Trie (ENPT) [55], [66], [67]. The basic structure of encoded component trie is shown in Figure 11. NCE consists of a Code Allocation Mechanism (CAM) and the State Transition Arrays (STA) storing the transition states. When a name arrives, the CAM is utilized to allocate a code to each component of the name. After all components are encoded as corresponding codes, these codes are mapped to the corresponding edges in NPT. Finally, a new trie named ENPT is constructed. For a given name, the lookup is implemented in ENPT by applying the STA and the index of FIB entry is finally returned.

In the CAM, according to the level information of each component in NPT, the component code allocation is implemented in every level and each component organized by one node is allocated a unique code. If several nodes in a level contain the same component, this component is reassigned a code, which is one larger than the maximal code allocated by these nodes. For example, the two nodes in the 2nd level contain one same component *UA*, which is respectively encoded as 2 and 3 at the both nodes. On the basis of CAM, the component *UA* needs to be reassigned a code 4 to replace code 2 and code 3.

The STA is composed of three parts, namely Base Array, Transition Array and Manage Array, where Transition Array contains three arrays. In Base Array, each entry has 4 bytes. The first two bits mean in which Transition Array the

information of corresponding state is recorded, where '00' represents the 1st array, '01' means the 2nd array and '10' means the 3rd array. Meanwhile, the left bits stand for the position of the target entry in Transition Array. Transition Array stores the number of transition states and index of FIB entry. Moreover, the component codes and the corresponding next lookup states are also stored into Transition Array. As for Manage Array, it indicates the free entries in the Transition Array. During the name lookup, the position of the lookup state in Transition Array is found by querying the Base Array. Then the component code is matched with the codes stored in Transition Array. The lookup is implemented sequentially until the longest matching name prefix is found. Finally, an entry index pointing FIB entry will be returned.

In the following, an example is given to show the lookup process in ENPT. Suppose the given name is */ndn/TJU* and corresponding encoded name is /1/3. Firstly, the first entry of Base Array is queried, where the first two bits are '00' and the left bits are '01' meaning the information of state 1 is stored in the first entry of the 1st array. Then, the first component code 1 is matched with the codes stored in this entry, and a match corresponding to the next lookup state 2 is found. The lookup turns to the second entry in the Base Array, in which the first two bits are '01' and the left bits are '01'. Then the second component code 3 continues to be queried in the first entry of the 2nd array. Finally, the matching name prefix /1/3 is found and the entry index pointing to corresponding FIB entry is also returned.

For the memory consumption and the lookup speed, the evaluation experiments are constructed on a PC equipped with Intel Core 2 Duo CPU of 2.8 GHz. For a dataset with 1 million names, the memory consumption of NCE is 79.64 MB and the average lookup speed is about 1.55 MPPS [55]. According to the results, NCE obviously shrinks FIB size by employing encoded components and improves the lookup speed compared with NPT, while the mapping between components and codes affects the lookup speed.

7) CHE: In addition to NCE, Feng et al., 2015 proposed a component hash encoding (CHE) [56] to encode components. CHE makes use of a recursive incremental hash function to generate continuous hash codes for all the children belonging to one same parent. That is, for a parent with three children, the hash code of the second child is related to the hash code of the first child, and the hash code of the third child is also related to the second. After the hash function assigns a hash code for each component in the component trie, the name lookup is performed by the State Transition Array (STA). Differing from the STA in NCE, this STA only consists of Base Array and Transition Array. Additionally, the STA assigns a transition array for each level of trie. For a given name, the lookup process is similar with NCE.

Aiming at the performance in terms of the memory consumption, the evaluation experiments are run on a platform with an Intel Core 4 Duo CPU of 8 GHz. The result indicates that for the 3124531 components from DMOZ, the original size of NPT is about 22.71 MB but CHE only consumes memory with 10.38 MB [56]. By encoding components with hash function, CHE reduces the memory consumption compared with NPT. However, the hash codes of the contiguous children are associated, which makes this encoding method inflexible.

8) RaCE: Given that the component encoding and the path compression can improve the performance of trie, Saxena et al., 2016 proposed RaCE [57] based on the both methods. When processing a name, the name is firstly divided into components and each component is assigned one code by the component encoding module. Differing from the CAM of NCE, each component in RaCE is assigned with a particular code and the same components share the same code regardless of the levels. Then these encoded components are mapped to an Encoded Name Trie. To achieve the path compression, the compact Radix trie is employed to construct this Encoded Name Trie in RaCE, where a single child having no index entry is merged with its parent. During the name lookup, the encoded name is queried from the root of the Encoded Name Trie. In addition, RaCE directly stores the forwarding information into nodes of trie.

Based on a server with Intel(R) Xenon(R) CPU E5-2695 v2 with 2.40 GHz and RAM with 128 GB DDR3, the performance evaluation experiments are implemented. For the dataset with 10 million names, RaCE only consumes memory with 21.93 MB and the lookup speed can reach 0.708 MPPS [57]. Owing to merging the child and parent, RaCE reduces the depth of trie thus improving the lookup speed. And RaCE further reduces the overall memory consumption compared with NCE by directly storing the forwarding information into the nodes. In common with NCE, the lookup performance may be affected by the mapping process from the components to codes.

9) Bitmap-based p-NPT: Additionally, to improve the memory usage and lookup performance, Seo et al., 2018 proposed the use of a priority trie and an encoded bitmap structure, called Bitmap-based p-NPT [58]. The scheme contains three parts, i.e. a Bitmap table, an Encoding table and a Node table, where the Bitmap table stores each node of the p-NPT, the Encoding table stores the child components of nodes, and the Node table stores the name prefixes and the corresponding forwarding information. Meanwhile, each entry of Bitmap table stores the hashBrief of the name prefix in the node, namely the fingerprint calculated by hash function, avoiding the unnecessary off-chip memory accesses. During the lookup procedure, the entry in the Bitmap table is first checked. If the node is a black node, the given input name always matches the name prefix stored at this node. Thus, the Node table needs to be checked to obtain the forwarding information. If the node is a priority node, the fingerprint of the input name is compared with the stored hashBrief. If there is a match, the name is checked in the Node table to obtain the forwarding information. Otherwise, the lookup procedure moves to a lower level based on the Encoding table.

For evaluating the performance of the scheme, a series of experiments are implemented. In the memory consumption, Bitmap-based p-NPT only consumes the memory with 18603.5 KB for the 600,000 name prefixes, which requires 46% less memory than NPT [58]. As for the lookup, the scheme reaches less than 2.5 off-chip memory accesses in average for each name lookup against FIB tables with 10,000 to 600,000

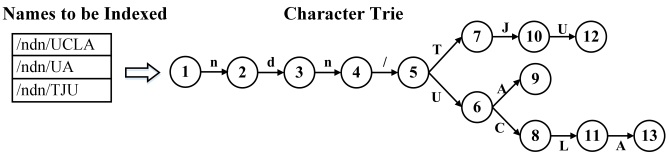


Fig. 12. Basic structure of character trie.

name prefixes [58]. The experimental results indicate that this scheme proposed can promote the memory usage and lookup speed compared to NPT by using Bitmap and the encoding. However, there may be hash collisions for hashBrief in the Bitmap table.

10) MATA: Except for the encoded component, the character used as granularity can also reduce the memory consumption of trie. In 2013, Wang et al. proposed multiple aligned transition arrays (MATA) [59] to perform lookup in the character trie. Figure 12 illustrates the basic structure of character trie with one-stride. In the character trie, each node represents a lookup state and an edge of trie stands for one character. Moreover, each valid transition, which guides the transition between the two lookup states during a lookup, is recorded into the basic Aligned transition array (ATA). During the name lookup, the lookup traverses the character trie from the root to the bottom by querying the valid transitions in ATA. In order to reduce the number of memory accesses, the MATA is utilized to query a name in the character trie.

In the basic ATA, to avoid the mapping conflicts from different transitions to one same ATA entry, a particular state ID is assigned to every state. Moreover, the state ID and input character act as the index for each transition, meanwhile this input character and state ID of next lookup are stored into the ATA entry. Suppose the current state ID is 800 and the ASCII code of an input character is 96, the transition can be found in the 896th array entry.

In the MATA, one-stride is replaced by multi-stride. That is, several characters rather than a character are processed on every transition. Moreover, when reading upon each transition, the component delimiter '/' is treated as the last character. Given that the available memory can limit multi-stride of a huge ATA, the MATA is divided into multiple small ATA. When two valid transitions are stored into a same ATA, if there is no collision between each other but collision with other valid transitions stored in this ATA, then the conflicting transition is stored to another ATA with the same length. If there is a collision between them, one transition tries to be stored to another ATA with different length. While each transition originating from a state is stored into the same ATA.

For example, the name with */ndn/TJU* is given and the MATA is three-stride. The *ndn* is firstly transformed to an integer, and then the address of first transition can be obtained according to the sum of root state ID and this integer. By reading upon the first transition, the next state ID is got. Then the integer of character '/' is added to this state ID, which means the position of the second transition. The lookup process continues iteratively until the matching name prefix is found.

This scheme is run on a GPU with NVIDIA GTX590 to evaluate the lookup performance. The result shows that MATA with 4-stride reaches 29.75 MPPS under average workload with lookup latency below 100 μs [59]. Clearly, MATA effectively improves lookup speed compared with the character trie with one-stride by increasing the number of characters processed per transition. For all transitions with the same state ID, they have to be stored into the same ATA, which restricts the memory utilization of MATA [60].

11) HATA: In order to improve the memory utilization of MATA, Li et al., 2014 proposed Hierarchical Aligned Transition Arrays (HATA) [60], [68] to further compress memory. In HATA, every level is assigned several ATAs, which store all valid transitions of this level. Moreover, the transitions belonging to the same state in the same level can be stored in different ATAs, which is different from MATA. For the name lookup, the target position of a transition can be obtained from the offset value provided by the corresponding state as well as the 32-bit value of input character. When the target position is found in the target level, there may be several candidate transitions stored at the same position of all ATAs. To confirm the final valid transition fastly, these ATAs are stored in different RAMs, which implement parallel processes. When the state having FIB pointer is encountered during a lookup, FIB pointer is recorded and the last pointer is returned as the matching result.

To evaluate the performance of HATA, the experiments are run on the FPGA chip. HATA consumes only 8.2 MB memory for FIB with 1 million entries. And in the lookup throughput, HATA with 15 pipeline stages can achieve average 125 MPPS with only 0.12 μs latency [60]. Owing to applying multi-candidates and parallel process, HATA significantly improves the memory utilization and lookup performance compared with MATA.

12) SNT: For reducing the memory consumption of trie, Tan et al. 2015 proposed the Split Name character Trie (SNT) [61]. Firstly, SNT makes a change to MATA, where each transition is represented by the 32-bit hash value of each component. In addition, to eliminate redundancy, the character trie is split into two smaller tries according to split length selected, namely a certain number of components. Every name whose length is shorter than or equal to the split length is constructed as one character trie. And other names are decomposed into two parts, where all second parts are constructed as other character trie. Meanwhile, each second part is assigned a hash table and the hash values of all first parts connecting with one second part are stored into this hash table. The lookups in both tries are respectively implemented in two changed MATAs. For an incoming name, SNT firstly confirms which trie the lookup is implemented in based on the name length. Then the lookup is performed in the same way with MATA.

Focusing on the memory consumption of SNT, the experiment is constructed on a DELL T620 server. The result indicates that SNT consumes 110 MB memory for the dataset with 1 million names [61]. Apparently, owing to splitting the character trie and compressing all first parts, SNT reduces the memory consumption compared with MATA. However, for

different datasets, choosing a suitable split length is difficult, and different lengths also influence the performance of SNT.

13) P-trie: In addition, to boost the lookup speed of MATA, Li et al., 2016 proposed a P-trie [62] with auxiliary forwarding information. In P-trie, all nodes are divided into three types. The first type of node stores the last character of a name. The second type of node has different forwarding information with the parent but the same forwarding information with all prefixes of its subtree. Except for the both, the rest of nodes belong to the third type. When querying a name, if there is a match at the first type or third type of node, the lookup continues. If the match is found at the second type of node, the forwarding information is immediately returned. If mismatch occurs at any node, the forwarding information stored in the last matched node is returned. In order to implement this trie, MATA has a small change. Each array entry of MATA is added an element to store forwarding information. Meanwhile, each entry is also set the most significant bit representing the second type, where 1 means the second type of node and 0 means not. The lookup process of this scheme is similar with MATA.

For the throughput, the experimental result shows that the throughput of P-trie achieves three times higher than the MATA [62]. P-trie observably improves the lookup speed by adding forwarding information.

14) NameTrie: To store and index forwarding information efficiently, Ghasemi et al., 2018 proposed a novel data structure, named NameTrie [63], which is an optimized character-trie. The NameTrie scheme contains two parts, namely minASCII name encoding and the NameTrie data structure. For minASCII encoding, the codes are in the range 32-126 for US-ASCII printable characters and the codes outside of 32-126 can be utilized to construct the NameTrie data structure. For example, The range 0-3 is reserved for End Of Piece (EOP) and delimiter (/). For the NameTrie data structure, there are two main components, including NameNodes which stores the nodes of trie and EdgeHT which conducts the edges of trie. When a name is queried in NameTrie, the name is first encoded by minASCII. Then the lookup starts from the root, and the next node continues to be queried from EdgeHT when encountering a mismatching byte or EOP. When the longest matching name prefix is found, the pointer to the actual FIB entry can be obtained from EdgeHT.

The experiments are implemented on a PC that is equipped with an Intel Core i7 2.1-2.8 GHz CPU, 6 MB cache, and 8 GB DRAM, to evaluate the performance of the NameTrie. For the 12 million names, the memory consumption of NameTrie is 467.33 MB, and the memory compression is 89.43% compared with NPT [63]. In the speed, the NameTrie in software can achieve 3.56, 3.72, and 3.25 million name insertions, lookups, and removals per second, respectively [63]. From the results, the NameTrie can obviously reduce the memory consumed and speed up the information lookup. However, the mapping between name and codes induces the complexity for indexing the forwarding information.

15) Binary Patricia Trie: Given that binary can further reduce the memory consumption of trie, Song et al., 2015 proposed a binary Patricia trie [46]. Generally, the basic structure of binary trie is based on bits, as shown in Figure 13. In

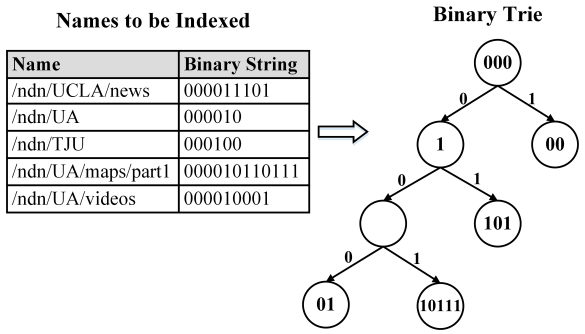


Fig. 13. Basic structure of binary trie.

this scheme, all names are treated as the binary strings coded by ASCII code to build the binary Patricia trie. Meanwhile, the parent only connects with two children by one pointer. And the branches of this trie, namely the bit 0 and 1, are also parts of the binary token of name. In addition, the binary Patricia trie stores the forwarding information into the nodes of trie thus reducing the memory consumption. When a name is inserted, the binary tokens of the name is queried from the root, while for the deletion, the corresponding leaf node is firstly removed and other nodes with single-child are combined with corresponding child along the path back.

The memory consumption of the binary Patricia trie is caused by the binary tokens and the differentiating bits, where the binary tokens take up the more memory. If the binary tokens are removed to reduce the memory consumption, the lookup scheme is changed from longest name prefix matching to longest name prefix classification, where the longest name prefix classification finds some candidate matching prefixes for a name. To support this change, FIB is divided into two subsets, where the first one is composed of all entries with flat name prefixes and the second one comprises of the entries whose names are covered by the name of some entries in the first subset.

Moreover, the two subsets are respectively constructed by speculative binary Patricia trie and binary Patricia trie. Differing from the binary Patricia trie, every internal node in speculative binary Patricia trie only stores a position of differentiating bit in form of the value, where the lookup is branched according to the value. For example, for the three flat prefixes, the first position of differentiating bit among them is the 12th bit, then the value 12 is stored in the root. According to this design, a novel forwarding namely speculative forwarding is implemented for an incoming name as follows. The longest name prefix matching is firstly performed in binary Patricia trie. If there is a match, the corresponding forwarding information is returned. If not, the longest name prefix classification is performed in speculative binary Patricia trie. Finally, the longest name prefix classification always returns some forwarding information.

In order to measure the performance of the binary Patricia trie, the experiments are implemented with SRAM and DRAM. For the lookup speed, the binary Patricia trie can achieve 142 MPPS and 20 MPPS based on SRAM and DRAM, respectively. And for the memory consumption, the binary

Patricia trie requires memory with 13.56 MiB for the dataset with 1 million names and the two Patricia tries used for speculative forwarding only consume 5.58 MiB memory [46]. Owing to storing the forwarding information into the nodes, binary Patricia trie distinctly reduces the memory consumption and boosts the lookup speed. For speculative forwarding, the imprecise forwarding of the longest name prefix classification may cause forwarding loops [46].

16) FPT: Given that the fingerprints, namely the hash values of names have the more balanced information, Yuan et al., 2015 proposed the fingerprint-based Patricia-trie (FPT) [48] [69] to construct FIB. In the scheme, each name is treated as a fingerprint by employing a hash function and the binary strings of all the fingerprints are used to construct the binary Patricia trie. Moreover, to reduce the depth of trie, FIB is divided into smaller subsets, each of which is respectively constructed by a small FPT.

For the lookup performance of this scheme, the evaluation experiments are run on a server equipped with 12 Intel Xeon E5-2630 cores with 2.3 GHz. The result shows the average leaf-node depth of this trie is reduced from 52 to 24 by using 20-bit fingerprints. And the lookup latency of FPT is better than the binary Patricia trie [48]. Using fingerprints can further reduce the depth of trie, thus FPT has a higher lookup speed compared with original binary Patricia trie. As the number of names is proportional to the number of nodes in trie, using fingerprints instead of names cannot observably reduce the memory consumption of Patricia trie [48].

17) N-FIB: To further reduce the memory consumption of the binary Patricia trie, Saxena et al., 2016 proposed N-FIB [64] with an efficient Patricia trie. N-FIB is almost same with the binary Patricia trie. However, in order to shrink FIB size, N-FIB can directly aggregate multiple names into one. Meanwhile, the forwarding information of name is stored into the corresponding nodes. When inserting a name, if the given name is the possible matching name prefix of other names stored in nodes, these names are removed and their forwarding information is added together to the node of this given name. If the possible matching name prefix of the given name already exists in trie, only the forwarding information of this given name is stored in the corresponding node. In other cases, the insertion process of one name is similar with binary Patricia trie.

For measuring N-FIB, the evaluation experiments are run on a platform installed with Intel(R) Xenon(R) CPU E5-2695 v2 with 2.40 GHz and RAM with 128 GB DDR3. For a dataset with 29 million names, N-FIB consumes memory with 264.79 MB and the lookup time for 29 million names is 403.841 micro-second [64]. Distinctly, N-FIB further reduces memory consumption compared with binary Patricia trie by aggregating the names. And owing to reducing the depth of trie, the lookup speed is improved.

18) TB^{2}F: Additionally, in order to reduce the depth of trie, Quan et al., 2014 proposed TB^{2}F [65] with a compressed multi-bit trie named Tree-Bitmap and counting Bloom filters. In this scheme, each name is divided into two segments, where the first segment with fixed-length is stored into the Tree-Bitmap and the second segment with variable length is stored into a counting Bloom filter. During the name lookup, the two segments of the name are respectively queried in the Tree-Bitmap and counting Bloom filter, then the forwarding information is returned after the hash check.

The first segment of the name is determined by a split-level parameter, i.e., the number of split components. Suppose the split-level parameter is 3 and the name */ndn/UA/maps/part1* is given. Then the name is split to two segments, namely the first segment with */ndn/UA/maps* and the second segment with */part1*. The Tree-Bitmap is applied to store all first segments and each node of this tree maintains two types of bitmaps, where the first one is for the internally stored prefixes and the second one is for the external pointers. During the lookup for a name, the match of first segment is firstly implemented in the Tree-Bitmap. Additionally, in order to further reduce the lookup time, parallel lookup is employed. During the first segment matching, the hash values for all possible prefixes of the second segment are pre-calculated.

A series of small counting Bloom filters are utilized to store all the second segments, each of which is connected to the corresponding node of Tree-Bitmap via a pointer. During the lookup for the second segment of a name, the hash values of all possible prefixes are directly used for the lookup in counting Bloom filters. Finally, the longest matching name prefix obtained from the two matching name prefixes is checked in a hash table, and then the forwarding information is returned.

In this scheme, for the name whose length is shorter than or equal to the split-level parameter, the lookup is only implemented in Tree-Bitmap until the longest matching name prefix is found or the matching fails. While the names with longer length are firstly divided into two segments. Then the first segment is queried in Tree-Bitmap, meanwhile the pre-computation for the second segment is implemented. If there is a match in Tree-Bitmap, the second segment is queried in corresponding counting Bloom filter with the results of pre-computation. Then, the two matching name prefixes are linked to one as the longest matching name prefix. Finally, a hash check in hash table is performed to find the forwarding information.

In order to test the efficiency of this scheme, the experiments are implemented on a router testbed installed with two 4-core Intel Xeon(R) CPU of 2.27 GHz and 16G RAM. In the experiments, the lookup speed is about 2 MPPS [65]. As shown in the experimental results, the TB^{2}F boosts the lookup speed by reducing the depth of trie. Meanwhile, minimizing the number of the prefixes stored in counting Bloom filter also can reduce the false positives. Nevertheless, the different datasets have different split-level parameters, which influence the overall performance of TB^{2}F.

B. Hash Table-based Schemes

Hash table is also a basic data structure utilized on FIB for the advantages of fast lookup. As shown in Table IV, a total of 9 schemes with this data structure are proposed. In the following, the 2-stage LPM [73] as the typical one with random search is emphatically introduced, while the other schemes are given a brief exposition.

TABLE IV
HASH TABLE-BASED SCHEMES ON THE FIB.

SM	Scheme Name	THT	THF	NHF	Key Feature	Pros	Cons
Optimal Linear Search	BF-FIB&PF [70] So et al.	CHT	CityHash64	-	Combining with BF	Performing better than hash table in search	Hash table induces high memory consumption
	FHT [48] Yuan et al.	DHT	-	5	Fingerprint collision table	Eliminating the impact of fingerprint collision	Hash table induces high memory consumption
	SMH [71] Xu et al.	CHT	-	= CN	Component as the hashing key	Performing better than hash table in search	Hash table induces high memory consumption
	FTDF-HT [72] Shubbar et al.	CHT	SipHash	1	two-dimensional filter	Reducing needless access	Hash table induces high memory consumption
Random Search	2-stage LPM [73] So et al.	CHT	SipHash	2	Improved hash table	Reducing the number of hash lookups	The different datasets have different the searching length
	Greedy-SPHT [74] Wang et al.	ISONPHT	-	1	Perfect hash table	Performing better than SPHT in lookup speed	The different datasets have different the searching length
	Binary Search of Hash Tables [75] Yuan et al.	CHT	CityHash SipHash	2	Binary search	Performing better than hash table in search	The different datasets have different length in the root
	Level Pulling [75] Yuan et al.	CHT	-	-	Combining with trie	Performing better than binary search	There is a trade-off between the lookup reduction percentage and the size of the cache-like structure
	Ω-LPM [76] Wang et al.	CHT	-	-	Statistical Optimal search path	Performing better than binary search	Recalculating optimal search path affects the update performance

SM:Search Method; THT: Type of Hash Table; THF: Type of Hash Function; NHF: Number of Hash Function
CHT: Chained Hash Table; DHT: d-left Hash Table; ISONPHT: Improved String-Oriented Near-Perfect Hash Table; CN: Component Number
- : Not mentioned

1) BF-FIB&PF: In 2012, So et al. proposed a lookup scheme for FIB with hash table and Bloom filter [70]. In this scheme, every name is treated as a hash value stored into each entry of hash table. When a name is given, the lookup always starts from the entire name and continues with the shorter prefixes till the longest matching name prefix is found. However, the names with the same hash value are stored into the linked list and the chain walks limit the lookup speed. To discover lookup miss before checking the linked list, every hash table bucket is assigned a Bloom filter. Meanwhile, given that the sequential lookups is performed on Content Store, PIT and FIB, the next lookup in hash table can be pre-fetched during the previous lookup.

Based on a processor with 2 GHz, the experiments are constructed to evaluate the performance of this scheme. The result indicates that the average forwarding rate of this scheme reaches 1.53 MPPS with a single thread [70]. Owing to the detection for mismatch earlier, the scheme improves the lookup speed.

2) FHT: In addition to the BF-FIB&PF, Yuan et al., 2015 proposed a collision free fingerprint-based hash table (FHT) [48] [69] with d-left hash table for speculative forwarding. In FHT, every name is treated as a fingerprint in form of hash value stored into the entry of hash table. In addition, FHT utilizes a collision table to deal with fingerprint collisions. During the insertion for a name, if the fingerprint collision occurs, the name is stored into the collision table. During a lookup, the collision table is firstly checked. If the matching fails, the lookup is performed in hash table.

Aiming at the memory consumption of FHT, the experiment is constructed. And the result shows FHT with 4-left hash table only requires the memory with 3.14 GB for 1 billion names [48]. In FHT, storing fingerprints requires less the memory than storing names, and the collision table eliminates the impact of fingerprint collision.

3) SMH: To further improve the lookup performance, Xu et al., 2015 proposed a Scalable Multi-Hash (SMH) [71]. Differing from operating the entire name with hash function, each component is treated as a hash value in SMH, which is stored into corresponding hash tables. That is, the first component of the name is recorded in the first hash table and the second component corresponds to the second hash table. Moreover, the chains are in charge of storing the other components having the same hash value. To reduce the false positives, the entry of each component is added a Name ID which the component belongs to. During a lookup, every component of the name is firstly transformed into a hash value, and then the lookup is sequentially implemented in corresponding hash tables.

In order to measure the performance of SMH, the evaluation experiments are implemented on a server, which is equipped with Intel Core i5-2300 CPU with 2.8 GHz. For the dataset

with 2 million names, the lookup speed reaches 21.45 Mbps and the memory consumption is 61.63 MB [71]. Owing to using the hash value of each component, SMH improves the lookup speed. And the false positives are also reduced by using Name ID. However, the memory consumption caused by the pointers in these hash tables needs to be further reduced.

4) FTDF-HT: Beyond that, Shubbar et al., 2017 proposed a new approach named fast two-dimensional filter with hash table (FTDF-HT) [72], aiming at reducing the unnecessary access to the hash table and minimizing the search time. The architecture of FTDF-HT is composed of two parts, i.e. the FTDF and hash table, where FTDF acts as the pre-searching and hash table is responsible for storing the interface information. FTDF, which is a matrix of m * n bits, maps every element into a single position inside the matrix by calculating the fingerprint of the element and applying a quotienting technique to get the (q) most significant bits as the row and the (r) least significant bits as the column. If the element is inserted into FTDF, the corresponding position of matrix is set to one. In FTDF-HT, the name and its prefixes need to be inserted FTDF and hash table, but the interface information of the name only is added into the hash entry corresponding to the name. In the search phase, all the name prefixes from the shortest one to the longest one are first checked in FTDF. If one prefix is matched, hash table is queried to find the matching entry. If FTDF returns the negative result, the search is terminated.

In the performance evaluation, the experiments are performed in a PC, which is equipped with Linux Ubuntu 14.04 LTS and processor Intel Core i7-4500U CPU with 1.80 GHz. The experimental results indicate that FTDF has the low false positive and memory access compared to Bloom filter and Quotient filter. Moreover, for the dataset with 10 million names, FTDF-HT can achieve 3.940 MSPS in the lookup speed. From the evaluation results, FTDF-HT can boost name lookup as only one hash function needs to be calculated in one query, where this hash value can work in both FTDF and hash table. While the hash collision and the memory utilization in hash table need to be still improved.

5) 2-stage LPM: In order to reduce the number of hash lookups, So et al., 2013 proposed a 2-stage LPM scheme [73], [77] with chained hash table. In the scheme, each name is treated as a hash value stored into the entry of hash table by using a 64-bit SipHash function. For an incoming name, a 2-stage LPM is implemented to find the longest matching name prefix. The name lookup starts from a certain short prefix and continues to shorter prefix or restarts from a longer prefix.

To achieve fewer cache misses, hash table utilizes a compact array instead of the linked list, where 7 pre-allocated compact entries comprises of a hash bucket. Meanwhile, prefetching hash table bucket for the next lookup is also applied to reduce the impact of data cache misses, which is triggered after finishing all hash computations but before the lookup.

In the 2-stage LPM, virtual FIB entries are utilized. During the name insertion, not only this name is inserted into FIB, but also the corresponding prefix with the certain short length, namely virtual entry, is also inserted into FIB. At the first stage of this scheme, the lookup for a name starts from a certain short prefix and continues to shorter prefixes until the match is found. If there is no match, the lookup terminates. Otherwise, there may be the matching name prefix with a longer length L, and then the second stage starts to perform from the prefix with length L in the same way with the first stage.

Suppose the certain short prefix has two components and the routing table contains two entries, namely */ndn* and */ndn/TJU*. When */ndn/UA/maps* is inserted, a virtual entry */ndn/UA* is also stored into the table, where the longer length L is the length of three components. When */ndn/UA/maps/part1* arrives, the lookup starts from */ndn/UA*. Then a matching name prefix */ndn/UA* is found and the lookup restarts from the prefix */ndn/UA/maps*. Finally, the longest matching name prefix is obtained and the forwarding information is returned. When */ndn/UCLA/news* arrives, the lookup also starts from */ndn/UCLA* but the matching fails. Then the lookup continues to the shorter prefix */ndn*, and finally the corresponding forwarding information is returned.

By applying Cisco ASR 9000 router, the experiments are performed to test the performance of this scheme. In the memory consumption, this scheme consumes about 153 MB memory for FIB with 1 million entries. And with one single thread, the average forwarding throughput can reach 1.90 MPPS [73]. Owing to changing the starting length, this scheme reduces the number of hash lookups thus achieving great lookup performance. While the length of certain short prefix is difficult to be chosen for different datasets, which needs to be considered.

6) Greedy-SPHT: In addition to the 2-stage LPM, Wang et al., 2013 proposed Greedy-SPHT [74], [78] to reduce the number of hash lookups. Greedy-SPHT presents the improved string-oriented perfect hash table, which rebuilds the hash function to treat each name as a hash value without collision. Moreover, the scheme makes use of two arrays in the name lookup, where the first array records the component number of the longer prefix in the next lookups and the second array records the component number of shorter prefix. During the name lookup, the lookup firstly starts from a certain short prefix. If a match with the current short prefix is found, the lookup is implemented with the component number stored in the first array. Otherwise, the component number stored in the second array acts as the prefix length of next lookup.

For evaluating the performance, the scheme is implemented on a server with 6-core CPU. For the dataset with 3 million names, the lookup speed of Greedy-SPHT reaches 3.961 MPPS with a single thread and 57.138 MPPS with 24 parallel threads. And the memory is consumed by 72.95 MB for this dataset [78]. By using perfect hash table, Greedy-SPHT greatly reduces the memory consumption. Moreover, Greedy-SPHT improves the lookup speed owing to querying from a chosen short prefix. In common with 2-stage LPM, the different datasets have different length of certain short prefix, which influences the performance of this scheme.

7) Binary Search of Hash Tables: Besides the aforementioned schemes, Yuan et al., 2015 proposed a lookup scheme with the binary search of hash tables [75]. In this scheme, all prefixes with the same length shares one same hash table. And these hash tables are constructed as a balanced binary

search tree, where each node represents a hash table. In this tree, the root stores the prefixes with certain short length, the shorter prefixes are stored in the left subtree, as well as the longer prefixes are stored in the right subtree. During the name lookup, the lookup starts from the root. If the match is found in the corresponding hash table, the lookup continues in the right subtree, otherwise in the left subtree.

Based on a server with two 6-core Intel Xeon E5-2630 processors, the lookup latency of this scheme is evaluated. The result displays that the lookup latency is respectively reduced by 21% and 45% by employing CityHash and SipHash [75]. Because of the binary search, this scheme obviously boosts lookup speed. Similarly, choosing different lengths of prefixes stored in the root has different impacts for the performance.

8) Level Pulling: In order to further optimize longest name prefix matching with the binary search of hash tables, Yuan et al., 2015 also proposed level pulling [75]. In this scheme, all names are inserted into a name component trie. If the trie nodes of the name exceed a certain threshold, the name and the number of children are cached into a cache-like structure. When a name is queried, the cache-like structure is firstly checked. If a match is found, the lookup continues in the binary search tree by using the next-level suffix of matching name prefix. If not, the lookup with this name is implemented in the binary search tree until the longest matching name prefix is found or the matching fails.

By applying a Python program, the number of hash lookups is tested. For the dataset from Alexa, the maximal hash lookup reduction percentage can reach 12.08% [75]. Owing to utilizing the cache-like structure, the scheme further reduces the average number of hash lookups. However, the choice of threshold is a trade-off between the hash lookup reduction percentage and the size of the cache-like structure [75].

9) Ω-LPM: Similarly, given that the number of hash table probes, namely the search path of a hash-based LPM algorithm, directly determines the lookup performance, Wang et al., 2017 proposed Ω-LPM [76] to optimize the search path of the hash-based LPM, thus improving the lookup performance. To support the random search, which means only the longer prefixes need to be queried when one prefix is matched or the shorter prefixes need to be queried when one prefix is dismatched, Ω-LPM reconstructs FIB by inserting the name prefixes of one name into the original FIB, aiming at increasing the lookup speed. Meanwhile, for further decreasing the search time complexity, the scheme presents a dynamic programming algorithm to find the optimal search path. According to FIB, the dynamic programming algorithm can calculate the number of hash table probes in the optimal search path from a prefix set to another prefix set, where the prefixes in a set have the same number of components. Moreover, the optimal search path, i.e. the optimal prefix length to start, is also calculated by applying this algorithm. When implementing the name lookup, the optimal search path elected by the dynamic programming algorithm acts as the starting prefix. If the optimal search path is matched, the longer prefix is checked to find the exact longest matching prefix. Otherwise, the shorter prefix is checked.

For evaluating the lookup performance improved by optimizing the search path of hash-based LPM, the experiments are implemented in a commodity PC with two 6-core CPUs, which runs Linux Operating System in the version 2.6.41.9-1.fc15.x86_64. For 10 million names, it only needs to check 2.03 prefixes on average to find the longest matching prefix, and the search path length of Ω-LPM decreases gradually as the prefix table size grows. Meanwhile, in the lookup speed with the single thread, NameFilter with Ω-LPM can achieve 2.03 MPPS and 1.88 MPPS under average workload and heavy workload, respectively, which reaches 1.43 times faster than NameFilter. Beyond that, the extra prefixes caused by reconstructing the FIB decreases gradually as the number of prefixes increases, therefore the extra memory consumption can be negligible on FIB with tens million names. Significantly, Ω-LPM can reduce the number of hash table probes owing to utilizing the dynamic programming algorithm, and possess both low memory consumption and excellent scalability. However, considering that the optimal search path must be recalculated when FIB is updated, the update performance of Ω-LPM needs to be further improved.

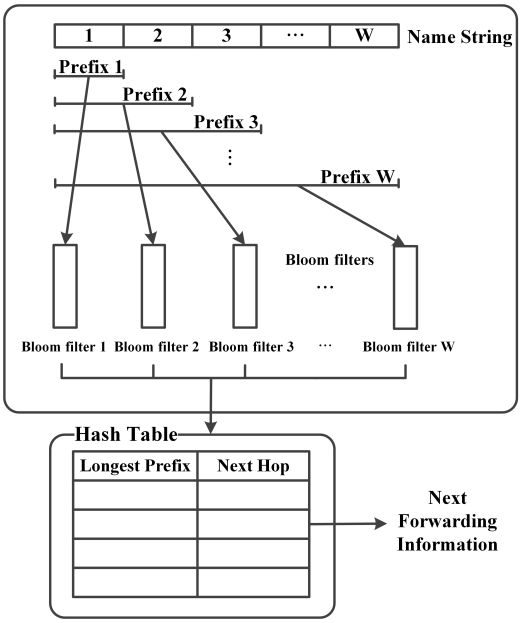


Fig. 14. Basic structure of Bloom filter on the FIB.

C. Bloom Filter-based Schemes

So far, the number of schemes with Bloom filter has reached 12, as shown in Table V. Generally, to support the LNPM, FIB employs multiple Bloom filters and a hash table, where each Bloom filter determines whether a name with some fixed length is in FIB and hash table is responsible for storing the forwarding information of names, as illustrated in Figure 14. Given that NameFilter [79] and BFAST-DA [88] are representatives of the two types which are interface deployment and composite structure, they are emphatically introduced, while the others are briefly expounded.

1) NameFilter: In 2013, Wang et al. proposed NameFilter [79] with a two-stage Bloom filter. In this scheme, the first stage utilizes multiple Bloom filters, each of which records the

TABLE V
BLOOM FILTER-BASED SCHEMES ON THE FIB.

Usage	Scheme Name	TBF	ADS	THF	NHF	Key Feature	Pros	Cons
Interface Deployment	NameFilter [79] Wang et al.	SBF CBF	CHT	DJB	-	Merging BF	Performing better than BF	Performance is related with the number of router interfaces
	I(FIB)F [80] Munoz et al.	IBF	N/A	-	-	Iterated hash	Performing better than BF at false positives	Iterated hashes of name is related with each other
Composite Structure	Parallelized Lookup [81] Ding et al.	CBF	CHT	-	-	Parallelism	Reaching high throughput by parallelism	False positives is not well solved
	Fast FIB Lookup [82] Fukushima et al.	SBF	CHT	-	-	Using the information of previous hop	Smaller latency than BF	False positives impact the searching accuracy
	PBF [83] Perino et al.	SBF CBF	CHT	-	-	Each BF as a block	Performing better than HT in search	A suitable threshold of expansion is difficult to choose
	NLAPB [84] Quan et al.	CBF	Trie CHT	-	-	Combining BF with trie	Minimizing the drawbacks of BF and trie	The different datasets have different split level parameters
	NPT-BF-Chaining [85], Lee et al.	SBF	CHT	64-bit CRC	-	Transforming prefix trie to HT	Achieving a single memory access on average during a lookup	Hash table induces high memory consumed
	BF&HT (Balanced-Tree) [86] Shimazaki et al.	SBF	CHT	-	11	Organizing conflicting entries into binary tree	Accesses in HT cannot increase	Hash table induces high memory consumed
	LIBF [87] Manghnani et al.	SBF	CHT	-	10	Binary search	Reducing number of lookup than BF	The different datasets have different length in the root
	BFAST-DA [88] Dai et al.	CBF	CHT	DJB CRC-32	9	First-Rank-Indexed, Hardware instructions	Outperforming NCE and NameFilter	Shifting related entry is needed when deleting a entry
	MaFIB [89] Li et al.	MBF CBF	CHT	MD5 SHA1	2	Localizable BF	Minimizing memory consumed of on-chip	There are collisions in MA values
	B-MaFIB [90] Li et al.	MBF CBF	Bitmap CHT	CityHash 256	1	Allocating memory Dynamically	Reducing memory consumed of off-chip	There are collisions in MA values

TBF: Type of Bloom filter; ADS: Additional Data Structure; THF: Type of Hash Function; NHF: Number of Hash Function
SBF: Standard Bloom filter; CBF: Counting Bloom filter; IBF: Iterated Bloom filter; MBF: Mapping Bloom filter; CHT: Chained Hash Table
N/A: Not applied; - : Not mentioned

prefixes with the same length, and the second stage assigns a Bloom filter to each interface. During the name lookup, the length of longest matching name prefix for a name is firstly determined by the first stage. Then the second stage is responsible for ascertaining the corresponding forwarding information.

In the first stage, to accelerate the lookup speed, several hash values are stored into one single word. During a lookup, the first hash value of the name is firstly calculated with DJB hash function. Then according to the previous hash value of the name, the location of the next hash value is obtained. After several loops, all the hash values are stored into one word. Moreover, the first bit of the word is in charge of calculating address in Bloom filters and the rest bits are calculated by using a 'AND' operation. If the result of 'AND' operation is 1, the longest matching name prefix exists. Otherwise, the matching fails.

Given that the number of memory accesses in the second stage is large, all Bloom filters connecting to interfaces are combined to a Merged Bloom filter. All bits at the same location of Bloom filters are combined into a bit string stored in a word. Suppose 8 Bloom filters are given and the first bit of each Bloom filter is 1, then all bits are merged into a bit string 11111111 stored into the first entry of Merged Bloom filter. When querying a name, the corresponding bit strings of the name are processed via a 'AND' operation and the location showing '1' means the forwarding interface. For example, the bit strings of a name are respectively 11111111, 11001111 and 01000000. Then the strings are calculated by using 'AND' and the result is 01000000, which means that 2 is the forwarding interface. In addition, given that Bloom filter cannot support dynamic deletion, each Bloom filter is equipped with a counting Bloom filter. The names are firstly recorded in counting Bloom filter then the result of counting

Bloom filter is mapped to Bloom filter. Finally, all Bloom filters are merged into the Merged Bloom filter in the second stage.

For the performance evaluation in terms of the lookup speed and the memory consumption, the experiments are constructed on a platform equipped with two 6-core CPUs (Intel Xeon E5645*2) with 1.6 GHz. For the dataset with 3 million names, the lookup speed of NameFilter achieves 2.033 MPPS with a single thread. And the NameFilter consumes memory with 64.73 MB to store these names [79]. As shown in the results, NameFilter can greatly improve the lookup speed with the two-stage Bloom filter. And the memory consumption is obviously reduced owing to replacing hash table with Merged Bloom filter. Nevertheless, the false positives caused by Bloom filters are not discussed in this scheme.

2) I(FIB)F: Additionally, Munoz et al., 2017 proposed to construct a FIB utilizing Iterated Bloom Filters (IBFs) defined as I(FIB)F [80], aiming at reducing the probability of false positives. Here, IBFs apply the iterated hash function, which hashes an input iteratively by feeding the output of each iteration into the input of the next. Based on the properties, IBFs are to split the m bits of a Standard Bloom filer to d IBFs containing m/d bits. That is, the first level of IBFs treats the first component of the input as the hash key, and then the second level considers the concatenation of the first hash value and the second component as the hash key. In the scheme, a separate I(FIB)F composed of IBFs is deployed in each forwarding interface. When an Interest arrives, the iterated hashes of name are directly queried against each possible interface. And the interface with the largest match is selected for forwarding.

Through evaluating the performance of I(FIB)F, the results indicate that the probability of false positives is observably reduced compared with Standard Bloom filter, as some hashing results in first several IBFs are coincide. Besides, owing to less iterated hash functions (sometimes only one) used in IBFs, the complexity required is lower than Standard Bloom filter.

3) Parallelized Lookup: To improve the lookup speed, So Ding et al., 2012 proposed parallelized lookup scheme [81] with Bloom filter and hash table. In this scheme, all names are inserted into multiple Bloom filters, each of which records the names with the same length. For a name lookup, all possible prefixes of the name are simultaneously queried in Bloom filters. Only a positive result is returned, the lookup is implemented in hash table. Finally, the forwarding information of the name is returned from hash table.

Aiming at the throughput of this scheme, the experiment is implemented based on TILEPro64 multicore platform. The result indicates that the throughput exceeds 6 M/s with 43 tiles when the load factor is 10.0 [81]. Clearly, this scheme can achieve high throughput with parallelization.

4) Fast FIB Lookup: In addition, Fukushima et al., 2013 proposed fast FIB lookup [82] for non-aggregable names based on the information of previous hop. In this scheme, the name prefixes are divided into leaf prefix and non-leaf prefix, where the leaf prefix has no child in assumptive prefix trie and the non-leaf prefix is contrary. Meanwhile, the number of leaf prefixes for different length is stored in a small table. During a lookup for a name, the lookup starts from the longest matching name prefix length of previous hop. Firstly, the small table is checked to judge whether the prefix with this length is a leaf prefix. If there is a match, which means there is not the matching name prefix with longer length, and then the lookup with this prefix length continues in Bloom filter stored in on-chip memory. When the result is positive, the check is implemented in hash table stored in off-chip memory and the forwarding information is returned. If the prefix with this length is a non-leaf prefix or the result in Bloom filter is negative, the lookup with this name length is performed in Bloom filter as the conventional lookup.

Through the experiment, the lookup latency of this scheme is evaluated. For FIB with 620 million prefixes, the latency is about less than half of the conventional lookup [82]. Owing to using prefix length of previous hop, this scheme improves the lookup speed.

5) PBF: In order to improve the lookup performance, Perino et al., 2014 proposed a prefix Bloom filter (PBF) [83], which is responsible for identifying the length of longest matching name prefix. PBF comprises of several blocks, each of which is represented by a Bloom filter. In this scheme, every name is stored into a block according to the hash value of its first component. Moreover, given that many names with the same first component lead to a high probability of false positives in one block, the block expansion is proposed. If the number of the names with length L in a block exceeds the threshold, the names with length L or longer length are stored into another block decided by the hash value of the prefixes with length L. In addition, a bitmap is added to each block to record the prefix length of the expansion. During a lookup, the corresponding block is firstly found according to the first component of the name. Then the bitmap is checked to identify if the expansion occurs at this name length or shorter length. If not, the lookup is performed in this block. Otherwise, the destination block is found and the lookup is implemented. After the length of longest matching name prefix is determined by PBF, the lookup is implemented in hash table for the forwarding information.

To measure the performance of this scheme, the experiment is implemented. And the result shows that the forwarding rate reaches 6.7 MPPS when PBF size is larger than 20 MB [83]. Distinctly, PBF achieves efficient forwarding operation by using several blocks and reduces the false positives with block expansion.

6) NLAPB: Given that the less names are added to Bloom filter, the probability of false positive is lower, Quan et al., 2014 proposed a lookup scheme called Adaptive Prefix Bloom filter (NLAPB) [84], [91] as an index of FIB, which is similar with TB^2F. In the scheme, every name is divided into two parts. The first part with fixed length is recorded in counting Bloom filter to determine whether the forwarding information of this name is stored in FIB. And the second part with variable length is stored into a component trie which is indexed by a hash table. During the name lookup, the lookup is implemented according to the name length as the lookup process in TB^2F.

For the lookup speed of this scheme, the experiments are

constructed on a server installed with 6-core CPU (Intel Xeon (R) 2.27 GHz). The result indicates that the scheme can achieve about average lookup speed with 0.8 MPPS [84]. By splitting the names, the number of the prefixes stored in Bloom filter is less, thus the probability of false positive is reduced in NLAPB. Moreover, the lookup speed is also improved owing to reducing the depth of trie. However, the management for names is computationally expensive due to the calculation of hashes and updating names in trie [53]. In common with TB^2F, the different datasets also affect the performance of this scheme.

7) NPT-BF-Chaining: For reducing the number of memory accesses during a lookup, Lee et al., 2016 proposed NPT-BF-Chaining [85], [92] with Bloom filter and hashing-based name prefix trie. In the scheme, the name prefix trie is transformed to a hash table by using the hash function, where each prefix stored in trie is treated as a fixed-length string stored into the hash entry. Moreover, the NPT-BF and NPT-BF-Chaining schemes are utilized in the name lookup of this hash table. In NPT-BF, the lookup is firstly implemented in a Bloom filter by using the prefix with a certain short length. If the result is positive, hash table is checked and the lookup with longer prefix continues in Bloom filter when a match is found in hash table. In order to further reduce the number of memory accesses in hash table, the lookup is sequentially performed in Bloom filter until a negative result is returned or all possible prefixes of the incoming name are queried. After the length of longest matching name prefix is determined by Bloom filter, a check is implemented in hash table. In addition, putting the forwarding information of the best matching name prefix into every internal node that has no forwarding information can further improve lookup performance, where the best matching name prefix represents the direct ancestor of the internal node.

Owing to the consecutive lookup in Bloom filter and pre-placing the forwarding information, NPT-BF-Chaining can achieve a single memory access on average during a lookup.

8) BF&HT (Balanced-Tree): In addition, Shimazaki et al., 2016 proposed a lookup scheme with hash table and Bloom filter [86] to improve the lookup performance. In the scheme, all names are recorded in a Bloom filter, which is responsible for identifying whether a name is stored in hash table. Meanwhile, each name and corresponding forwarding information are stored in a hash table. Additionally, in order to address hash collisions in a hash entry, all entries with the same hash value are organized as a balanced binary search tree, which is built in the lexicographical order of each name. When querying a name, the lookup algorithm in Bloom filter and hash table is the same with the common.

Through the performance evaluation, the number of lookups is respectively reduced by 47.8% and 14.1% compared with hash table with linear list, as well as Bloom filter and hash table with linear list [86]. As shown in the results, owing to applying the binary tree in hash entry, the number of lookups in hash table cannot increase even if there are hash collisions.

9) LIBF: Given that reducing the number of lookups in Bloom filter can improve the lookup speed, Manghnani et al., 2016 proposed Length Indexed Bloom Filter (LIBF) [87], which is similar with the binary search of hash tables. In this scheme, all names with same prefix length are recorded in one Bloom filter and these Bloom filters are constructed to a balanced binary tree, where each node represents a Bloom filter. When a name arrives, the lookup is firstly implemented from the root. If the matching result is positive, the lookup continues in the right subtree for a longer matching name prefix. Otherwise the lookup is performed in the left subtree for the shorter. After the longest matching name prefix is determined, the check in hash table is implemented to find the forwarding information.

To measure the performance in terms of the memory consumption and the probability of false positive, the simulation is run on the NDNSim. In the memory consumption, this scheme consumes about 1.2 MB memory for 0.091 million names. And the probability of false positive is less than 0.2% with five hash functions [87]. From the results, the overall probability of false positive in LIBF is reduced with multiple Bloom filters. Moreover, by using the binary search in Bloom filters, the number of lookups is reduced obviously thus boosting the lookup speed. Nevertheless, there is a similar concern with binary search of hash tables [75] to be considered.

10) BFAST-DA: Moreover, in order to further reduce the memory consumption, Dai et al., 2017 proposed an index called Bloom Filter Aided Hash Table (BFAST) [88]. BFAST is composed of a main counting Bloom filter (mCBF) and a multi-function hash table along with two methods supporting lookup and insertion. In BFAST, the mCBF is applied to balance the load among hash table slots, where each name is treated as a signature stored in the least-loaded slot corresponding to the smallest counter. To better support lookup and insertion, the two methods are employed when processing the name. Moreover, given that the pointers in hash table consume a lot of memory, the First-Rank-Indexed scheme is utilized in BFAST, which reduces the overall memory consumption.

In the both methods, the first method uses a series of auxiliary counting Bloom filters, whose number is equal to the number of hash functions, to record corresponding relation between the hash function and the smallest counter. When inserting a name, the name is firstly inserted into the mCBF then according to the smallest counter in mCBF, the name is stored into corresponding slot of hash table. If the smallest counter is calculated by the first hash function, the name is also inserted into the first auxiliary counting Bloom filter. However the first method has to consume the extra memory, so BFAST utilizes the second method called dynamic adaptation (DA) to resolve this problem. When a name is inserted, the other names affected are reinserted to the least-loaded hash table slots, but the corresponding counters do not need to be added. And the deletion is implemented along with lookup when deleting a name. Suppose there is a name */ndn/UA* and the corresponding counters contain the 1st, 3rd and 4th counters and the smallest counter is the 1st counter. When inserting the name */ndn/TJU*, the corresponding counters including the 1st, 2nd and 3rd counters are increased. Then */ndn/TJU* is inserted into the least-loaded hash table slot. While for the name */ndn/UA*, the increase of three counters makes the 4th counter the smallest counter among its three counters. Thus, the name */ndn/UA* is moved into the 4th slot from the 1st slot.

In the First-Rank-Indexed scheme, all hash table slots are divided into buckets, where the 1st names in all slots belonging to the same bucket are stored into an address-contiguous array called FIA. In addition, hash table degenerates to a bitmap. If the hash table slot is not empty, the corresponding bit is set to 1, otherwise 0. And the bitmap is also split into multiple parts, each of which corresponds to a bucket equipped with a FIA. Moreover, each FIA has an initial address which is stored in the pointer array. By using hardware instructions, the memory address of every entry in FIA can be calculated. Suppose the name */ndn/UA* stored in the 4th slot is queried. The 4th bit of first part in bitmap is firstly checked and the result is 1, which means the 4th slot is non-empty. Then the name is confirmed to exist in the third entry of the 1st FIA via hardware instruction. Finally, the address of the third entry is obtained from the pointer array. After a check in this entry, the forwarding information is returned.

For the performance evaluation including the memory consumption and lookup speed, the experiments are implemented on a platform, which is equipped with two Xeon E5645 (6 cores, 2 threads, 2.4 GHz) and DDR3 ECC with 48 GB. The results show that BFAST with the second method (BFAST-DA) can achieve 2.14 MPPS with a single thread in the lookup speed and 24.53 MPPS with 24 threads in forwarding throughput. In the memory consumption, BFAST-DA requires 341.66 MB of memory for 10 million names [88]. Obviously, BFAST-DA significantly reduces the memory consumption owing to the First-Rank Indexed scheme. But for the deletion, the corresponding entry of FIA needs to be removed, which also affects other entries stored in the FIA.

11) MaFIB: Based on the fact that Bloom filter cannot locate the element, Li et al., 2017 proposed the MaFIB [89] with Mapping Bloom filter (MBF). MaFIB is composed of two parts, namely MBFs stored in on-chip memory and a Packet Store stored in off-chip memory, where MBF is responsible for indexing and the Packet Store stores the entries of all names. MBF comprises of Bloom filter and Mapping Array (MA), in which Bloom filter is in charge of determining whether a name exists in MBF and the MA value is treated as the offset address to locate position in Packet Store. And all names with the same length are recorded into one Bloom filter. To locate the name, Bloom filter is divided into several parts, each of which corresponds to a bit of MA. Before inserting a name, all bits of MA are initialized to 0. When inserting the name, if the parts in Bloom filter are mapped by the hash functions, the corresponding bits of MA are set to 1. Then according to the value of MA, the entry of the name is found in the Packet Store. In order to support update operation, each Bloom filter is assigned a counting Bloom filter stored in off-chip memory, which maps the update result to Bloom filter.

Based on a PC installed with an Intel Xeon E5-1650 v2 CPU with 3.50 GHz, the performance of this scheme is measured. For dataset with 1 million or 2 million names, the on-chip memory consumption is only 4.194 MB. In the building time, MaFIB takes 6427 ms for the dataset with 1 million names [89]. As shown in the results, MaFIB can significantly reduce the memory consumption in the on-chip by employing Bloom filter. Moreover, the MA resolves the problem that Bloom filter cannot locate the name. However, the collisions of MA values are not discussed in this scheme [93].

12) B-MaFIB: Though MBF has good performance in the memory consumption of the on-chip memory, it has to reserve enough memory space in the off-chip memory, which causes a lot of wasted memory. To deal with this, Li et al., 2018 improved MBF to Bitmap-Mapping Bloom filter(B-MBF), and proposed an enhanced FIB named B-MaFIB on the base of B-MBF [90]. Except for Bloom filter and MA, B-MBF adds a Bitmap, where each bit is changed into two bytes. Differing from MBF, the MA value in B-MBF is no longer the offset address of the off-chip memory, but the location of the name in Bitmap, and the offset addresses are stored in Bitmap. When inserting a name, the position of the name in Bitmap can be calculated based on the MA value. In accordance with the order in which the name is inserted in the Bitmap, the corresponding sequence number is assigned to this name and recorded in this slot, serving as the offset address. Finally, the memory location in the Packet Store is dynamically assigned to this name by the offset address and the corresponding base address, meanwhile the forwarding information of the name is stored in the Packet Store.

For measuring the performance of B-MaFIB, the experiments are performed in a PC, which is equipped with an Intel Core i3-3220 CPU of 3.30 GHz and DDR3 SDRAM of 4 GB. Compared to MaFIB, B-MaFIB reduces the memory consumption of the off-chip memory without increasing the consumed memory of the on-chip memory, in which the memory consumed in the off-chip memory only 107.885 MB for 2 million names [90]. For the throughput, B-MaFIB achieves 1.099 MSPS, which about obtains six times speedup over MaFIB. Obviously, B-MaFIB can reduce the memory consumed of the off-chip memory owing to Bitmap, and improve the throughput. While the collisions of MA values are also not discussed in this scheme.

D. Summary

Hitherto, there are 18, 9 and 12 data structure schemes proposed for FIB with trie, hash table and Bloom filter, respectively. Among them, trie is more suitable to FIB because trie's logical structure can reduce the memory consumption of the hierarchical names and better support the name matching algorithm LNPM.

The number of schemes with trie [46], [48], [51]–[65] is the largest for the logic of trie contributes to FIB name lookup, but the shortage cannot be ignored that the lookup speed is relatively slow. Among all the schemes with trie proposed, the hardware parallelism and reducing the depth of trie are utilized to improve the performance of trie, including merging the single child with its parent (such as PC-NPT [54]) and splitting a trie to small tries (such as SNT [61]).

Compared with trie, the schemes based on hash table [48], [70]–[76] can perform lookup in fast speed. And in these schemes, the method including optimal linear search and random search to achieve the name matching algorithm LNPM needs consideration, where the prefix length from which the lookup starts in hash table is vital for reducing the number

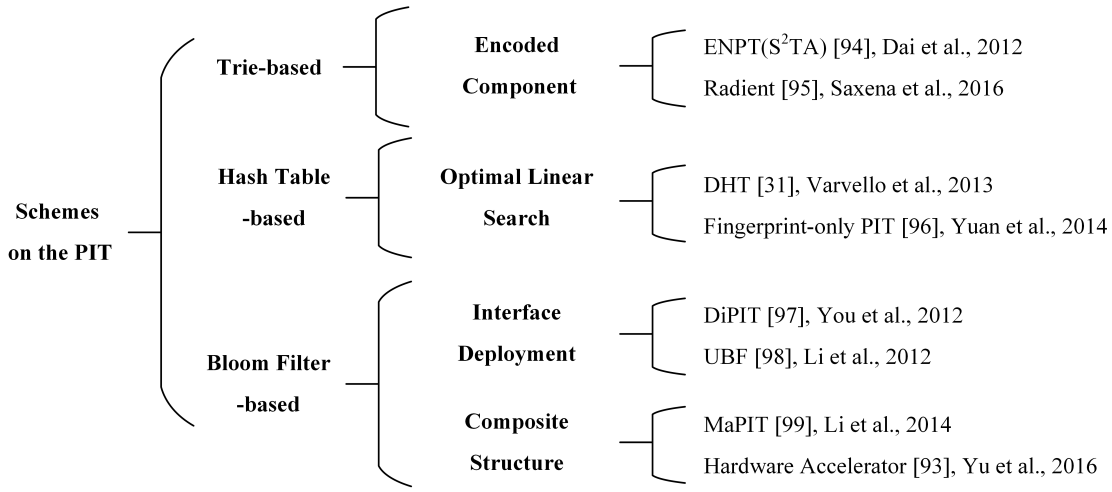


Fig. 15. Classification of data structure schemes on the PIT.

of hash lookups. Moreover, given that hash table has to store name strings to ensure the forwarding correctness, some schemes treat the name strings as fingerprints stored in the hash entry to reduce the memory consumption, for instance FHT [48].

Except for these schemes, the schemes with Bloom filter [79]–[90] have advantages not only in lookup speed, but also in memory consumption. However, the fact that Bloom filter only determines whether an element exists in the set decides that the deployment of Bloom filter must be considered. Therefore, some schemes assign a Bloom filter to each interface, such as NameFilter [79], while the most record all names into Bloom filters regardless of interfaces and assign a hash table to store the forwarding information, for example PBF [83] and MaFIB [89].

At present, binary Patricia trie [46] and BFAST-DA [88] are the better data structure schemes proposed. Binary Patricia trie can boost lookup speed and reduce memory consumed significantly by storing forwarding information into nodes and operating with binary, whereas the imprecise forwarding of the longest name prefix classification for speculative forwarding may cause forwarding loops [46]. As for BFAST-DA, it vastly reduces the memory consumption by using the FIA and improves the lookup speed, but the corresponding entry of FIA needs to be removed when performing deletion, which also affects other entries stored in the FIA.

VI. Data Structure Schemes of the PIT

In the section, all the data structure schemes aiming at PIT are presented based on trie, hash table and Bloom filter as shown in Figure 15. Further, the schemes with trie both belong to one type, i.e., encoded component, while the schemes based on hash table also fall into one category, i.e., optimal linear search. Moreover, similar with FIB, Bloom filter-based schemes are classified into two types, including interface deployment and composite structure. In the end, a summary which discusses the features of the data structure schemes and the more appropriate schemes for PIT is provided.

A. Trie-based Schemes

Given that PIT requires the more fast retrieval, trie is less used on PIT due to the slow lookup speed. At present, only 2 schemes with trie are proposed for PIT, as shown in Table VI. And the both schemes adopt encoded component as the granularity of trie and improve the encoding mechanism compared with NCE. In the following, the both schemes are given an introduction.

1) ENPT (S^2TA): In 2012, Dai et al. improved NCE and proposed an improved ENPT for PIT [94]. The improved scheme is similar with NCE, but makes the improvement in terms of the CAM and the STA. When an Interest arrives, the improved CAM assigns an available code to each component of the Interest name with a code allocation function, where the available code is new or freed by other components. Then all encoded components are constructed as an ENPT. Finally, the lookup is performed in ENPT by using the Simplified STA (S^2TA), which takes code as index to locate the next lookup state.

In NCE, every component is allocated a unique and consistent code and the code cannot be reused after deleting the name, which produces many codes wasted. Aiming at this problem, the CAM is improved. When a name is deleted, its codes are freed and available for the following names. And each component is dynamically allocated an available code, which is a new code or a freed code. Moreover, a code allocation function is utilized to allocate available codes to components with a hash function, and a pair including the component and corresponding code are stored in a hash table. This function contains three parameters, namely component, level and preceding code. The component parameter represents the component that needs to be encoded. The level parameter stands for the corresponding level which the encoded component belongs to. The preceding code parameter represents the code of the preceding component.

For Interest names, the code allocation function separately assigns available codes to components within each set, which represents all the edges from the same node. Therefore, one same component in different sets may be assigned different

TABLE VI
TRIE-BASED SCHEMES ON THE PIT.

UNL	Scheme Name	Key Feature	Pros	Cons
Encoded Component	ENPT(S^2TA) [94] Dai et al.	Improving encoding method	Outperforming NCE	Inflexible encoding method
	Radient [95] Saxena et al.	New encoding method, Path compression	Lower memory consumed than NCE	The mapping processing affects the lookup speed

UNL: Unit of Name Lookup

codes. For Data names, the function needs to find the correct code for each component within multiple codes. Suppose that the name */ndn/TJU* is taken as an Interest name. The component *ndn* is encoded as 1 by invoking the code allocation function, and then the function continues to encode the component *TJU* using *TJU*, 2 and 1 as the three parameters. Then an available code 2 is assigned to component *TJU*. Assuming that another Interest name */edu/TJU* arrives, the component *TJU* is assigned another available code 3. Consequently, as a component belonging to the 2nd level, *TJU* has two different codes, namely 2 and 3.

In addition, the original STA in NCE only handles linearly querying a code in a Transition Array. Therefore, the original STA is improved as a S^2TA to address this problem. In the S^2TA, the codes, which are not stored in the STA, are directly treated as the indexes to locate the next lookup state. For this S^2TA, there are two advantages. On the one hand, this method does not need to move data during the insertion and deletion. On the other hand, the complex memory management is not involved.

For the performance of this scheme, the experiments are constructed on the platform installed with Intel Xeon E5520 with 2.27 GHz and 15.9 GB RAM. The experimental results indicate that for the dataset with 10 million names, the memory consumption is 48.91 MB and the lookup speed achieves 3.27 MPPS [94]. Distinctly, compared with NCE, this scheme reduces the memory consumption and improves lookup speed. However, the code of current component is related with the code of the previous component, which influences the overall performance.

2) Radient: Additionally, in order to further reduce the memory consumption, Saxena et al., 2016 proposed the Radient [95], which is almost same with RaCE. Radient comprises of a Component Radix Trie and an encoded Name Radix Trie, where the both tries are constructed by Radix trie. For an incoming name, the Component Radix Trie is utilized to allocate a unique code to each component in the same way with RaCE. Then all encoded components are built to an encoded Name Radix Trie, in which the name lookup is implemented. In common with RaCE, the forwarding information is also stored in leaf nodes of encoded Name Radix Trie.

Based on a server equipped with Intel(R) Xenon(R) CPU E5-2695 v2 with 2.40 GHz, the experiments measuring the performance of this scheme are implemented. In the memory consumption, the total amount of memory used is 34.61 MB and 140.10 MB for the datasets with 10 million names and 29 million names, respectively. In the frequency of insertion, deletion and update, the scheme can achieve 0.077 MPPS, 0.062 MPPS and 0.086 MPPS for the dataset with 10 million names, respectively [95]. As expressed in the results, Radient significantly reduces the memory consumption compared with NCE. Moreover, this scheme achieves incremental updates on real-time basis. Nevertheless, in common with RaCE, the mapping between the components and codes influences the performance, which still needs to be improved.

B. Hash Table-based Schemes

DHT [31] and fingerprint-only PIT [96] with hash table are proposed for PIT, both of which use optimal linear search and transform each name into a hash value stored in hash entry. In Table VII, the information of both schemes are shown in detail. Here, the both are given an analysis.

1) DHT: In 2013, Varvello et al. proposed open-addressed d-left hash-table (DHT) [31] to reduce the memory consumption. DHT is a combination of the open-addressing method and d-left hash table. In hash table with the open-addressing, the size of every bucket is determined by the number of names which can be found with a single memory access, and a bucket stores entries with a fixed number. In the d-left hash table, every name is hashed d times with d hash functions and stored in the least-loaded bucket. Meanwhile, to minimize the memory consumption, the hash value of each name rather than the name is stored in hash entry. During the name lookup, the name is firstly treated as a hash value, and then the lookup is implemented in hash table.

For evaluating the performance of this scheme, the experiments are run on a Cavium Octeon network processor installed with 2 MB of shared memory and 4 GB of off-chip DRAM memory. In the experiments, the memory consumption is 483 MB for dataset with 8 million names. And for a dataset with 1 million names, the lookup speed can achieve 3.4 MPPS with 8 active cores [31]. Compared with hash table that stores names, DHT can reduce the memory consumption by storing hash values of names.

2) fingerprint-only PIT: In addition, based on the idea that most of the duplicate Interests can be aggregated in the edge routers while the core routers can relax the Interest aggregation, Yuan et al., 2014 proposed the fingerprint-only PIT for the core routers with d-left hash table [96]. In this PIT, every name is treated as a fixed-length fingerprint by employing the hash function and stored in the least-loaded bucket, in which each hash entry contains five parameters for supporting lookup. Moreover, when the number of entries exceeds the size of hash table, the overflowing entries are stored in a overflow table.

TABLE VII
HASH TABLE-BASED SCHEMES ON THE PIT.

SM	Scheme Name	THT	THF	NHF	Key Feature	Pros	Cons
Optimal Linear Search	DHT [31] Varvello et al.	ODHT	32-bit CRC	3	Using Open-addressing in d-left hash table	Improving lookup speed than d-left hash table	Hash table induces high memory consumption
	Fingerprint-only PIT [96], Yuan et al.	DHT	CityHash64	5	Fingerprint, Splitting PIT	Reducing memory consumed by using fingerprint	All segregated PITs have to be queried for Data

SM: Search Method; THT: Type of Hash Table; THF: Type of Hash Function; NHF: Number of Hash Function
ODHT: Open-addressed d-left Hash Table; DHT: d-left Hash Table

For each hash entry, the five parameters are the Occupy bit, the Collision bit, the fingerprint, the expiration time and the interface list, respectively. The Occupy bit shows if the hash entry is empty. When an entry is deleted, this bit is set to 0. The Collision bit represents whether several Interests with this fingerprint stored have been received. The fingerprint is the hash value calculated with a name. The expiration time with a fixed length is determined by the configuration and the unit of timer. If the existence time of an entry exceeds this time, the entry is deleted automatically. The interface list records all incoming interfaces of Interests that have the same fingerprint.

Additionally, the size of fingerprint-only PIT may be larger than one single memory chip. Hence, the single PIT is split into multiple segregated PITs, each of which is constructed by d-left hash table and stores the pending Interests from a few interfaces. For an incoming Interest, its information is added into one segregated PIT according to the incoming interface. Conversely, for an incoming Data, all segregated PITs are queried to obtain the interfaces, which corresponding Interests come from.

Focusing on the performance including the memory consumption and lookup latency, the simulation for fingerprint-only PIT is run on a hardware platform, which is equipped with 8 Intel Xeon E5540 cores, DDR3 with 12 GiB and L3 cache with 8 MiB. When PIT size is 16 million and the dataset contains 12.53 million names, the memory consumption of overflow table is less than 0.73 MiB. And for PIT size with 65536 entries, the lookup latency is about 1.2 μs [96]. As shown in the results, the fingerprint-only PIT can reduce the memory consumption by replacing names with fingerprints.

C. Bloom Filter-based Schemes

Compared with trie and hash table, Bloom filter utilized on PIT is more popular, as PIT needs to support more frequent read operations and write operations. As shown in Table VIII, DiPIT [97] and UBF [98] implement interface deployment, and MaPIT [99] and Hardware Accelerator [93] execute composite structure. Among these schemes, DiPIT and MaPIT are emphatically discussed, while others are briefly introduced.

1) DiPIT: In 2012, You et al., 2012 proposed a distributed scheme named DiPIT [97], [100] with Bloom filter to reduce the memory consumption. Given that Bloom filter cannot store forwarding information, a small PIT is constructed on each interface, where each small PIT is performed by a counting Bloom filter. In Figure 16, the basic structure of Bloom filter

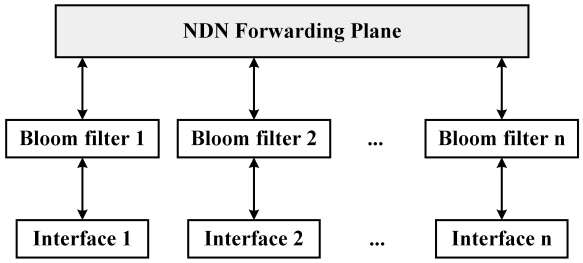


Fig. 16. Basic structure of Bloom filter implemented on the interface.

implemented on the interfaces is illustrated. Moreover, in order to address the false positives caused by counting Bloom filters, a Bloom filter is shared by all small PITs. When querying an Interest name, the lookup is sequentially performed in counting Bloom filter and the shared Bloom filter.

Each small PIT operates independently and records the Interest names into the corresponding counting Bloom filters according to the incoming interfaces of Interests. For an incoming Data, the Data name is queried in parallel on each small PIT. If a match is found, the Data is forwarded to the interfaces which corresponding Interests come from. Moreover, during the insertion for a name, the corresponding counter is increased by one. On the contrary, when a name is deleted, the corresponding counter is decreased by one.

Meanwhile, the shared Bloom filter is shared by all interfaces. Given that the Data returned does not involve the operation in shared Bloom filter, the shared Bloom filter is refreshed periodically to reduce its own false positives. When an Interest arrives, the Interest name is firstly checked in Content Store, and then queried in small PIT associated to incoming interface if no matching occurs in Content Store. If there is also no match in small PIT, the Interest is transferred to FIB and its name is added in the counting Bloom filter. Otherwise, this name continues to be queried in the shared Bloom filter. If a negative result is returned, the Interest is transferred to FIB and the name is added to the shared Bloom filter. Otherwise, the Interest is a duplicate which has been forwarded but unsatisfied, thus the Interest is discarded. When a Data arrives, the Data name only needs to be verified in each small PIT without querying the shared Bloom filter, and then the Data is forwarded to the matching interfaces.

In order to measure the performance of DiPIT, the experiments are performed based on the assumption which the networking line card contains 16 interfaces. When the Interest arrival rate is 100 Mpck and the probability of false positive

TABLE VIII
BLOOM FILTER-BASED SCHEMES ON THE PIT.

Usage	Scheme Name	TBF	ADS	THF	NHF	Key Feature	Pros	Cons
Interface Deployment	DiPIT [97] You et al.	CBF SBF	N/A	-	-	Shared BF	Reducing memory consumed by using BF	All CBFs have to be queried for Data
	UBF [98] Li et al.	SBF	N/A	-	12	The work of two BFs exchange over time	Reducing memory consumed than CBF	How to choose the length of each period needs to be considered
Composite Structure	MaPIT [99] Li et al.	MBF CBF	CHT	MD5 SHA1	2	Localizable BF	Minimizing memory consumed of on-chip	There are collisions in MA values
	Hardware Accelerator [93] Yu et al.	SBF	CHT	H_3	2	Utilizing FPGA	Performing better than BF in search	False positives of BF need to be resolved

TBF: Type of Bloom filter; ADS: Additional Data Structure; THF: Type of Hash Function; NHF: Number of Hash Function
SBF: Standard Bloom filter; CBF: Counting Bloom filter; MBF: Mapping Bloom filter; CHT: Chained Hash Table
N/A: Not applied; - : Not mentioned

is limited as 0.1%, the memory consumption is only 690 MB with 7 hash functions [97]. As shown in the results, DiPIT significantly reduces the memory consumption owing to assigning space-efficient Bloom filter to each interface. But for an incoming Data, all counting Bloom filters have to be queried, which negatively affects the lookup speed.

2) UBF: In addition, given the deletion error caused by false positive of counting Bloom filter, Li et al., 2012 proposed United Bloom Filter (UBF) [98] instead of counting Bloom filter to shrink PIT size, which comprises of two Bloom filters. In the scheme, one UBF is assigned to each interface. For UBF, the time is divided into two periods, where both Bloom filters are the current in turn. When each period starts, the non-current one initializes all bits to 0. During the period, the insertion is implemented in both Bloom filters, while the name lookup is performed only in the current one. At the end of the period, the identities of both Bloom filters are swopped. Bloom filter cannot support deletion, thus expiration acts as implicit deletion in UBF.

For the compression performance, the simulation experiment is performed with one CCN router and four hosts. The result shows that the memory consumption is reduced by 40% with 0.027% error rate compared with the original PIT [98]. Obviously, this scheme can reduce the memory consumption of PIT by employing UBF to record all names. Additionally, the error rate is significantly reduced compared with counting Bloom filter. While how to choose the length of each period needs to be considered.

3) MaPIT: Given that Bloom filter cannot locate the element, Li et al., 2014 proposed MaPIT [99] based on Mapping Bloom filter (MBF), which is similar with MaFIB. MaPIT is composed of two parts, namely the Index Table and Packet Store, which are stored on the on-chip memory and off-chip memory, respectively. The Index Table is constructed by a MBF including a Bloom filter and a Mapping Array (MA), where Bloom filter is applied to determine whether a name exists in the MBF and the MA value is treated as the offset address of storage position in Packet Store. And the Packet Store is a static storage caching all entries of the names. Additionally, to support the deletion, Bloom filter is equipped with a counting Bloom filter stored on the off-chip memory. During the insertion and deletion, the result of counting Bloom filter is mapped to Bloom filter synchronously. When querying a name, the name is firstly queried in MBF and then checked in the Packet Store if the matching result is positive in MBF.

In MBF, in order to locate the position of the name in the Packet Store, Bloom filter is separated into several parts, each of which corresponds to a single bit of MA whose size equals to the part number of Bloom filter. Before each name is inserted, the MA initializes all bits to 0. During an insertion, some bits in MA are set to 1 corresponding to the parts to which hash functions map in Bloom filter. Then, the storage position of this name in Packet Store can be found via the MA value. Suppose that Bloom filter is divided to 6 parts and the number of hash functions is three, and the names */ndn/TJU* needs to be inserted. Before the name */ndn/TJU* arrives, all bits of MA are set to 0, namely 000000. During the insertion, the 1st, 3rd and 4th parts of Bloom filter are mapped by hash functions, and then the corresponding bits in MA are set to 1. Finally, the MA value 101100 acts as the offset address in the Packet Store. After the name */ndn/TJU* is stored into the corresponding entry, the MA is re-initialized to 000000 for the next name.

Aiming at the memory consumption and building time, the evaluation experiments for MaPIT are implemented on a PC, which is installed with an Intel Core i3-3220 CPU at 3.30 GHz and DDR3 SDRAM with 4 GB. In the on-chip memory, MaPIT only requires 2.097 MB no matter whether the dataset is with 1 million names or 2 million names. In the building time, this scheme can reach 7488 ms for 1 million names [99]. From the results, MaPIT can minimize the on-chip memory consumption and directly locate the storage position without walking the Packet Store. But for MaPIT, the collisions of MA values may influence the forwarding correctness.

4) Hardware Accelerator: To speed up the packet processing, Yu et al., 2016 proposed a hardware accelerator [93] to reduce the workload of the software. In this scheme, the software manages the tables and protocols, and the hardware accelerator that is mainly composed of an on-chip Bloom filter and off-chip hash table performs a mirrored and simplified

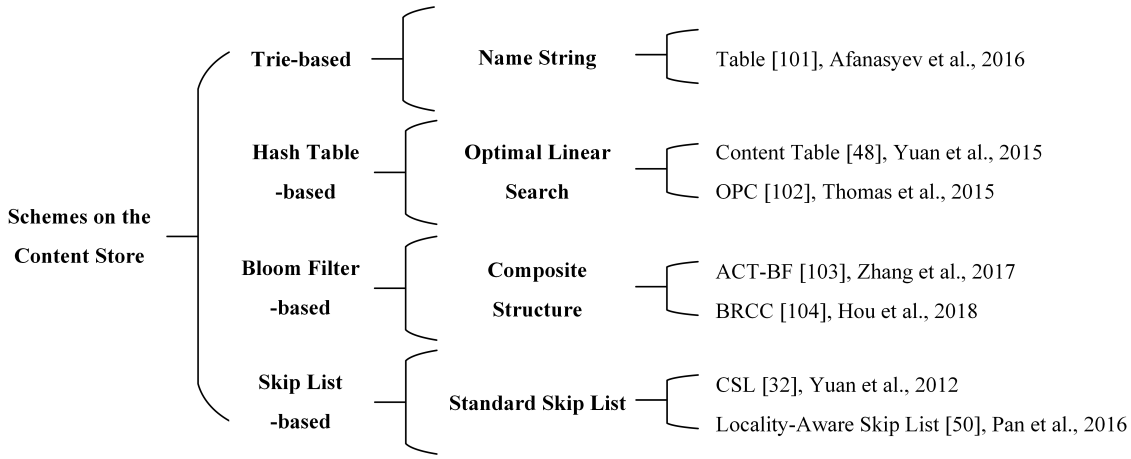


Fig. 17. Classification of data structure schemes on the Content Store.

copy of PIT. For boosting more efficient implementation of PIT, each name is treated as a fixed-length name ID with a hash function. Meanwhile, both the software and hardware accelerator utilize a name ID table to record each unique name ID, according to which the router can directly determine how to forward the Interest without FIB lookup. Moreover, the segment number of the packet is extracted from the name to assist retrieval. When an Interest arrives, the name ID and segment number of Interest are extracted, as well as the software sends the lookup Interest command to the hardware accelerator. Then the hardware accelerator firstly queries the key (name ID, segment number) in Bloom filter. If there is a match, the check is performed in hash table, and the matching result and the storage address of PIT are returned to software. If not, both the name ID and segment number are stored into hash table and PIT stored in the software. Meanwhile, the name ID is queried in the name ID table, and then the Interest is forwarded based on the forwarding interfaces stored in the name ID table if the matching result is true and valid.

In order to test the packet processing rate, the simulation is implemented on a Xilinx Virtex-7 FPGA. The experimental result indicates the overall processing rate can reach 56 MPPS to 60 MPPS [93]. Clearly, this scheme can greatly improve the processing rate by moving the packet processing operations to the hardware.

D. Summary

For the data structure schemes of PIT, the amounts based on trie, hash table and Bloom filter are 2, 2 and 4 respectively. Among them, Bloom filter which can achieve the fast retrieval and low memory consumption is more suitable to PIT, as PIT requires frequent read and write operations as well as efficient storage for vast names.

The number of schemes with trie [94], [95] is less due to its slow lookup speed, where both of the two schemes apply encoded component and improve the encoding mechanism compared with NCE, such as ENPT (S^2TA) [94]. Furthermore, the hash table-based schemes [31], [96] execute fast name lookup and treat each name into a hash value to reduce the memory consumed, but how to further improve the memory usage needs to be considered in these schemes. With regard to the schemes based on Bloom filter [93], [97]–[99], they show up the efficient performance in name lookup and memory consumption, while the schemes assigning Bloom filter to interfaces need to be improved for the performance of Data lookup, as all the Bloom filters have to be queried to find the correct interfaces, for example DiPIT [97].

According to the introduction of the above data structure schemes, MaPIT [99] is regarded as the better scheme for PIT. MaPIT resolves the problem that Bloom filter can not locate the elements in an innovative way, thus directly locating the storage position without walking the off-chip memory. Besides, MaPIT improves the lookup speed, and minimizes the memory consumption in the on-chip. But the collisions existing in MA cause negative influence for forwarding performance.

VII. Data Structure Schemes of the Content Store

In the section, as illustrated in Figure 17, the data structure schemes of Content Store are also presented on the basis of trie, hash table, Bloom filter and skip list, respectively. The scheme with trie, namely Table [101], is treated as the name string according to the granularity. The schemes with hash table both belong to one type, i.e., optimal linear search, and the Bloom filter-based schemes also fall into one type, i.e., composite structure. As for the schemes with skip list, the both use the standard skip list to ensure the order of packets. More details of the data structure schemes are shown in Table IX and analyzed in the first subsection, and the commons of these data structure schemes and better choice for Content Store are given in the final summary.

A. Schemes

1) Table: In addition, in order to improve the performance of lookup and cache replacement, Afanasyev et al., 2016 proposed an improved implementation of Content Store used in NFD [101], which includes the Table and the CachePolicy. The Table acts as the index, and the CachePolicy applies the queues for cache replacement, where the two-way relation is built with the Table iterator between the Table and the queues.

TABLE IX
SCHEMES ON THE CONTENT STORE

BDS	Scheme Name	ADS	DCRP	Key Feature	Pros	Cons
Trie	Table [101] Afanasyev et al.	Queue	Priority-FIFO LRU	Using set	Performing better in lookup and cache replacement	Depth of trie influences the lookup speed
Hash Table	Content Table [48] Yuan et al.	DLL	LRU LFU	Using fingerprint	Supporting replacement flexibly	Local searching in LFU
	OPC [102] Thomas et al.	DLL Array	LRU	Object-level indexing, packet-level storage	Avoiding the memory waste	The cache replacement is object-level
Bloom Filter	ACT-BF [103] Zhang et al.	Trie	-	Filtering mismatch request	Reducing lookup delay than trie	There are more memory consumption
	BRCC [104] Hou et al.	CHT	-	Optimizing the cache	Improving the cache efficiency	Hash table induces high memory consumption
Skip List	CSL [32] Yuan et al.	CHT	-	Content-order lookup	Performing better in ordered lookup	Starting from the head node induces lookup delay
	Locality-Aware Skip List [50] Pan et al.	CHT	-	Locality-aware	Performing better than CSL in lookup	Performance in random order and retransmission needs to be improved

BDS: Basic Data Structure; ADS: Additional Data Structure; DCRP: Deployable Cache Replacement Policy
DLL: Doubly Linked List; CHT: Chained Hash Table
- : Not mentioned

Additionally, the Table, which is sorted by Data name with implicit digest, utilizes a class called set in C++ language as an ordered container to store concrete entries. When an entry needs to be queried, the lookup is implemented in the Table.

Moreover, the CachePolicy including the Priority-FIFO and LRU records the data usage information to perform replacement. Priority-FIFO stores the information of unsolicited Data, stale Data and FIFO Data by utilizing three queues, respectively. Based on the usage information, the entry of Data can move between queues. As for LRU, only one queue is used to store the usage information. If an entry is used, refreshed or newly inserted, its Table iterator is moved to the tail of the queue. If an entry is removed, its Table iterator gets kicked out of the head of the queue. Combining the Table with the CachePolicy can achieve efficient lookup and support the cache replacement policy very fully. Furthermore, Priority-FIFO has better caching performance than the common FIFO owing to the classification of Data.

2) Content Table: To improve the lookup speed, Yuan et al., 2015 proposed a data structure of Content Store [48]. The structure comprises of packet buffers and a Content Table, where the packet buffers store every cached Data and the Content Table using d-left hash table acts as an index. In order to speed up the matching process, the fingerprint of a name instead of the name is stored into corresponding Content Table entry. During a lookup, the fingerprint of the name is queried in the Content Table. If there is a match, the memory address of Data with this name in the packet buffer is obtained from the Content Table entry. According to the address, the Data can be found in packet buffer. In addition, the Statistical info field of entry stores the information required by the cache replacement policy, such as the last reference time and the number of references. Moreover, an overflow table is in charge of dealing with overflow.

The scheme can support two cache replacement policies, namely LRU and LFU. For the achievement of LRU, the Content Table must combine with a doubly linked list, where the address of each list node is stored in corresponding Content Table entry. When there is a cache hit or a Data needs to be inserted, the corresponding list node is moved to the head of list. When the replacement is implemented, the current tail node is evicted from the Content Table. As for the LFU, each Content Table entry holds a counter, which records the usage number of a Data. When evicting a Data, local searching, which only searches the entries in d buckets, is utilized to find the entry with the smallest counter. Owing to using the fingerprints, the Content Table can achieve fast lookup and reduce the memory consumption. Content Table can totally support LRU by applying doubly linked list, but not completely support LFU due to the local searching.

3) OPC: Given that the limited storage of SRAM is the bottleneck for the packet-level caching, Thomas et al., 2015 proposed a novel caching scheme named Object-oriented Packet Caching (OPC) [102] to solve this issue, which combines the object-level indexing with packet-level storage. OPC consists of three data structures, namely a doubly linked list, the Layer 1 (L1) and Layer 2 (L2) indexes, where the L1 index stored in SRAM is performed as a hash table and the L2 index kept in DRAM is an array. Each entry in the L1 index stores the information of an object, including the order of the last packet of that object and a pointer to the final packet. The L2 index sequentially holds the cached packets of each object, where all packets under the same object form a linked list with pointers. During the lookup for a packet, the L1 index is firstly queried with the corresponding object name as the key. If the matching object exists, meanwhile the order of this

packet is smaller than the order of the last packet, the L2 index is checked to find the cached packet.

The doubly linked list, which is also kept in SRAM, is responsible for ranking the objects with LRU to support the cache replacement policy. If the replacement is implemented for lack of L1 space, the least important object in the doubly linked list is evicted from the list and the L1 index. Meanwhile, all the packets under this object are erased from the L2 index. If the L2 index needs the additional space, the last packet of the selected object gets kicked out of the L2 index. By using the object-level indexing and packet-level storage, OPC can overcome the bottleneck of SRAM and avoid the memory waste caused by storing the packets of large but unpopular objects. Compared with CSL, the cache replacement implemented in OPC is object-level, hence the replacement is not complete.

4) ACT-BF: Furthermore, to reduce average lookup delay of content Store, Zhang et al., 2017 proposed Adaptive Compression Trie based Bloom Filter (ACT-BF) [103] to filter out mismatch requests. The scheme mainly consists of Bloom filter, Counting Bloom filter (CBF) and a name Trie, where Bloom filter in on-chip SRAM is used to promptly check whether the content requested exists and CBF kept in DRAM deals with the update of Bloom filter, as well as the name Trie in DRAM stores all the names of content cached in Content Store. Additionally, SSD is utilized as content storage. Given the rare memory of SRAM, the memory assigned to Bloom filter is limited. Thus, to control the false positives of Bloom filter under the space constraint, the name prefixes in certain level of name Trie, namely Compression Trie representing the name set, are stored in Bloom filter. Moreover, for achieving higher request filter ratio, Adaptive Compression Trie (ACT) is proposed to dynamically add the lengthes of name prefixes in Compression Trie when most mismatches occur, or decrease the lengthes if there is no improvement or space for expansion. For the incoming Interest, its name prefixes are firstly checked in Bloom filter. If the mismatch occurs, the Interest is forwarded to PIT and FIB. If not, the name is queried in name Trie to locate the content. Otherwise, the content doesn't exist, then Interest is processed by PIT and FIB.

In order to evaluate the performance of this scheme, the evaluation experiments are performed. And the results show that over 85% of the mismatch requests are filtered under the false positive ratio of 1% [103]. Owing to filtering the mismatch requests, this scheme can reduce the lookup delay of Content Store thus improving the overall performance of forwarding plane.

5) BRCC: In addition, Hou et al. 2018 proposed a sum-up Bloom-filter-based request node collaboration caching (BRCC) approach [104], which promotes data caching efficiency, caching space utilization, and data content searching speed of Content Store. BRCC can accommodate the request frequency and distance to the subscriber, dynamically adjusting the content deployed to the router and enabling adjacent nodes to exchange the information of the cached content for updating, where the updating information is saved in CIT. To boost the lookup speed in CIT, a sum-up counting Bloom filter called SCBF is proposed, which is composed of a CBF with k level, a sum-up Table (ST) and a hash-based lookup Table (LT). Each slot of ST corresponds with one slot in LT, and the value of each slot is determined by the corresponding counter from each level of CBF. When a name needs to be inserted, the name is hashed into CBF with k hash functions, and the corresponding counters are added 1. Then the values of corresponding slots in SL are also added 1, and the slot with the smallest value is selected from ST, where its index is used to locate name in LT. Meanwhile, the packet carries a Caching Remain Age (CRA) field to support cache replacement, where the packet whose the value of CRA is zero needs to replaced.

For evaluating the performance of BRCC, a simulation is implemented on the Matlab simulation tool integrating the C++ language platform. The results indicate that the cache hit ratio and caching efficiency of BRCC are better than LCE. Moreover, the false positives of SCBF are reduced owing to using ST and LT.

6) CSL: In 2012, Yuan et al. discussed the structure of Content Store applied to CCNx [32], which is composed of the Content Skip List (CSL) and Content Hash Table (CHT). CSL acts as an index and CHT is responsible for determining whether a Data has been stored in Content Store. For an incoming Interest, CSL is firstly checked to find the possible content enabling to satisfy this Interest. Then the possible content is further checked to identify if the content can match with this Interest. For an incoming Data, the full name as a lookup key is queried in CHT. If no matching, the Data name is stored into the CHT and CSL. Then the Data is stored into a content array, which is employed to store every cached Data.

In this scheme, CSL is the implementation of the standard skip list. When querying a target element in skip list, the lookup always starts from the head node of the top list. Then the lookup continues to be performed horizontally until a current node, which is larger than or equal to the target, is encountered. If the element stored in current node is the same with the target, the target is found. If the element is larger than the target, the lookup returns to previous node and vertically drops to the next lower list. Then the lookup continues in the same way as before. Suppose there is a skip list with 3 layers and the element 9 is requested. First, the element 6 stored in the head node of the 3rd layer is compared with 9. The lookup is implemented horizontally as 6 is smaller. The next element of the 3rd layer is 17, which is larger than 9. Thus the lookup returns to the element 6 and vertically drops to the element 6 of the 2nd layer. Similarly, the next element of the 2nd layer is checked and this element is exactly 9. Consequently, the element 9 is found.

Every incoming Data is allocated a unique accession number following the order of arrival, which is stored in the corresponding skip list node. When the old cached Data needs to be replaced, the Data that has small accession number and is unpopular gets kicked out of the content array. Moreover, focusing on the lookup throughput, the evaluation experiments are run on a platform with i5-5200u processor of 2.2 GHz and 8 GB memory [50]. The result indicates that CSL achieves 0.548 MPPS with a single thread for Seq [50]. All Datas are stored sequentially, so the CSL can achieve ordered lookup and fully support the cache replacement policy. Given that

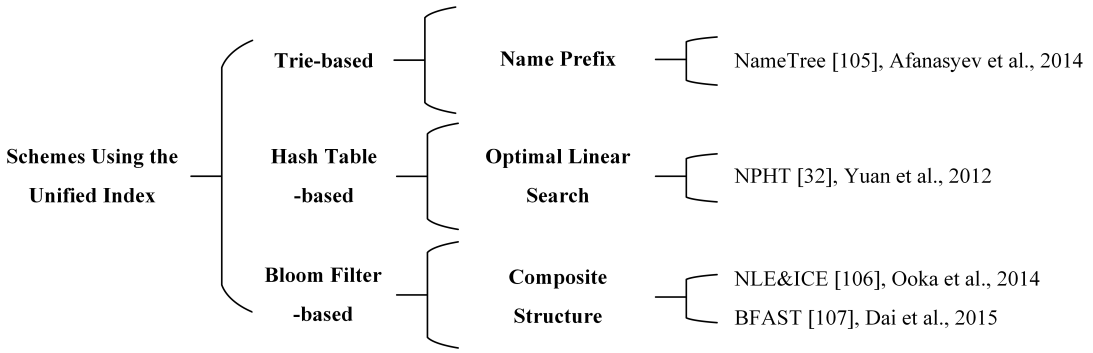


Fig. 18. Classification of data structure schemes using the unified index.

lookup in skip list always starts from the head node, locating an element needs more steps in average for a large dataset. Undoubtedly, the lookup cost of skip list needs to be further reduced to support line-rate packet processing [50].

7) Locality-Aware Skip List: Aiming at the lack of CSL, Pan et al., 2016 proposed a locality-aware skip list [50]. To reduce the average steps of a lookup, the locality-aware skip list needs to record the address of skip list node accessed when querying a node, which serves for the following lookups with same name prefixes. Additionally, a content index cache storing the access information of name prefixes is utilized, which takes the name prefixes as index keys. And each cache entry holds retraced nodes of each level, corresponding node addresses and the recent node queried, where the retraced nodes are defined as the nodes that are greater than or equal to the target. When querying the node with same name prefix, the optimal node address is firstly chosen from several node addresses. Then the lookup starts from the optimal node address instead of the head node address until the matching item is found.

As for the cache replacement policy, the nodes with the same name prefix are stored together based on the segment ID, but not all nodes are stored in some order. Consequently, the locality-aware skip list can not totally support the cache replacement policy due to the division storage. Furthermore, based on a platform installed with i5-5200u processor of 2.2 GHz and 8 GB memory, the evaluation experiments are implemented for this scheme. In the throughput, the scheme can reach 1.796 MPPS with single thread for Seq [50]. Owing to querying a node from the optimal node instead of the head node, the locality-aware skip list can achieve three times faster than CSL in lookup speed. While in the case of random order and retransmission, the lookup performance of this scheme needs to be further improved.

B. Summary

As mentioned above, only 7 data structure schemes are proposed for Content Store so far. In view of the fact that cache replacement policy should be the main concern in the design of Content Store, skip list is more appropriate to satisfy its requirement because skip list is ordered and supports the replacement policy.

Among these data structure schemes proposed [32], [48], [50], [101]–[104], not all schemes can support the cache replacement policy. For the schemes that can satisfy this requirement, except for the basic data structures, they must combine with the additional data structures to achieve this function. Moreover, some schemes can support various replacement policies by changing the additional data structure used or the data usage information stored, such as Content Table [48] and Table [101]. But some schemes can not fully support the cache replacement policy, such as OPC [102].

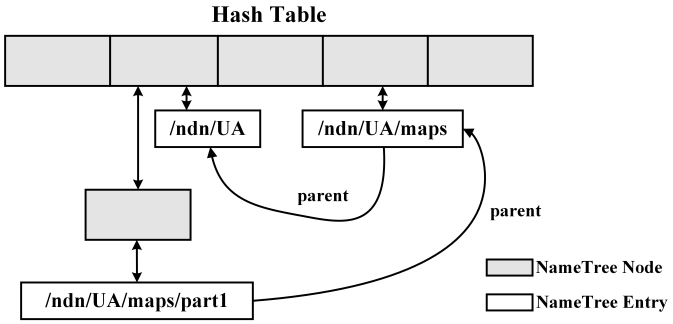


Fig. 19. Basic structure of NameTree.

Combining the requirements of Content Store for lookup speed and cache replacement policy, Content Table [48] and locality-aware skip list [50] are the better choices for Content Store. By using hash table, Content Table implements the fast lookup. Meanwhile it can well deploy LRU owing to the usage of doubly linked list, but can not totally support LFU due to the local searching. Locality-aware skip list improves the standard skip list, enhancing the lookup performance significantly. However, in the case of random order and retransmission, the lookup performance of this scheme needs to be further improved.

VIII. Data Structure Schemes Using the Unified Index

Given that the three tables perform the lookup with the names, they can be theoretically organized as a single unified structure in memory. However, the requirements of the three tables are very different, as mentioned in Table II. Therefore, only 4 data structure schemes achieve the unified index, which are presented based on trie, hash table and Bloom filter, as illustrated in Figure 18. Here, NameTree [105] with trie belongs to name prefix type, while NPHT [32] with hash table applies

TABLE X
SCHEMES USING THE UNIFIED INDEX

BDS	Scheme Name	ADS	THF	NHF	Key Feature	Pros	Cons
Trie	NameTree [105] Afanasyev et al.	CHT	CityHash	2	Indexing PIT and FIB	Improving the lookup speed than HT	There is a trade-off between lookup time and wasted memory
Hash Table	NPHT [32] Yuan et al.	N/A	-	-	Indexing PIT and FIB	Reducing the numbers of lookup	Hash table induces high memory consumption
Bloom Filter	NLE&ICE [106] Ooka et al.	CHT	-	-	Indexing Content Store, PIT and FIB	Performing better than HT in lookup	The top several Datas cannot be cached due to the threshold
	BFAST [107] Dai et al.	CHT	CRC-32	9	Indexing Content Store, PIT and FIB	Performing better than HT in lookup	The pointers in HT induce high memory consumed

BDS: Basic Data Structure; ADS: Additional Data Structure; THF: Type of Hash Function; NHF: Number of Hash Function
CHT: Chained Hash Table
N/A: Not applied; - : Not mentioned

optimal linear search. Moreover, the schemes with Bloom filter both belong to the same type, i.e., composite structure. More details of the data structure schemes are shown in Table X and analyzed in the first subsection, where NameTree as the classical one is emphatically analyzed. Then a summary that represents the features of data structure schemes and the better one with efficient performance is presented in detail.

A. Schemes

1) NameTree: In addition, to reduce the number of lookups, Afanasyev et al., 2014 proposed NameTree applied in NFD [105], which is a unified index for PIT and FIB. NameTree indexed by the names is composed of NameTree entries, each of which contains the entries of PIT and FIB, as well as the corresponding name. Logically, all NameTree entries are constructed to a tree with pointers. Structurally, NameTree is implemented by a chained hash table with linked list, as illustrated in Figure 19.

In hash table, every NameTree entry is mapped to a bucket. Instead of an existing library, this hash table is performed by scratch, which can control better the performance tuning. When a name is given, the name is treated as a hash value with CityHash function, and the hash value is stored into a bucket of hash table. When several names are mapped to the same bucket, the corresponding entries are chained by a linked list. Moreover, the size of hash table needs to keep pace with the number of NameTree entries. To minimize the overload of resizing process of hash table, a NameTree Node is added to every bucket and manages two pointers to the NameTree entry and next Node respectively. When a NameTree entry is empty and has no children, this entry is removed. During the lookup for an entry, the name of the entry is firstly queried in hash table. If there is a match, the information stored in the entry is returned. Otherwise the result is null.

During the packet forwarding, NameTree reduces the number of lookups thus boosting the lookup speed because of the related lookups on PIT and FIB. But the number of buckets in hash table is a trade-off between lookup time in the chaining and wasted memory caused by empty buckets [105].

2) NPHT: In 2012, Yuan et al. first proposed a Name Prefix Hash Table (NPHT) shared by PIT and FIB with hash table in CCNx [32]. In the scheme, the two types of entries are indexed during the name lookup, in which the forwarding information and pending Interest information are recorded, respectively. And each bucket in NPHT connects with the both entries via pointer. Moreover, each Interest stored in PIT contains a unique nonce field to avoid loop in the network, meanwhile all the nonce fields are also recorded in a Propagating Hash Table (PHT). When an Interest arrives, the PHT is firstly queried with the nonce filed of Interest. If a match is found, the Interest is discarded. If not, the lookup continues to be performed in NPHT and Content Store.

3) NLE&ICE: To accelerate the processing speed of Content Store, PIT and FIB, Ooka et al., 2014 proposed a hardware framework for the router, which is composed of the name lookup entity (NLE) and the interest count entity (ICE) [106], [108], [109]. NLE utilizes the CAM and a distributed-and-load-balancing Bloom filter to manage the variable-length names and ICE constructed by a hash table is responsible for selecting content worth caching. When a packet arrives, its name and content are operated by NLE and ICE, respectively. For the name, the connection between the name and any entries of three tables is handled by NLE, which enables to find the desired entry with only one lookup. During a lookup, the name is queried in Bloom filter. If the matching result is positive, the lookup is performed in CAM to obtain a pointer to RAM, where RAM stores the entries of three tables. As for the content, ICE records the request number for each content, which avoids caching rarely requested content. If the requested frequency of the content exceeds a given threshold, the content is cached.

Aiming at the performance in terms of lookup speed and insertion speed, the evaluation experiments are performed on the CCN-based router proposed. The results indicate that this scheme can reach 81.6 MPPS and 69.7 MPPS in the lookup speed and insertion speed, respectively [108]. This scheme significantly improves the processing speed. Moreover, only caching suitable Data can improve the caching performance. Due to the given threshold, the top several Datas cannot be

cached. Consequently, how to choose an optimal threshold needs to be considered.

4) BFAST: In addition, Dai et al., 2015 proposed BFAST [107] to implement indexing for three tables, which is similar with BFAST-DA. BFAST only consists of a multi-function hash table and a main counting Bloom filter along with the first method for insertion and deletion. In common with BFAST-DA, every name is stored as a signature in the least-loaded hash bucket determined by the smallest counter in the main counting Bloom filter. When inserting or deleting a name, the processing is performed by auxiliary counting Bloom filters in the same way with BFAST-DA. Additionally, each hash entry is linked with table entities by a pointer and a type parameter stored in hash entry indicates which table this hash entry belongs to. To accelerate the lookup speed, the interface information is directly stored in the hash entry.

By employing the platform equipped with two Intel Xeon E5645 (6 cores, 2 threads, 1.6 GHz) and DDR3 ECC of 48 GB, the experiments are implemented to measure the performance of this scheme. In the memory consumption, BFAST consumes 419.32 MB and 1517.60 MB memory for FIB with 3 million entries and 10 million entries, respectively. In the lookup throughput, BFAST can achieve about 1.8 MPPS for FIB with 10 million entries by using a single thread [107]. BFAST obviously improves the lookup throughput owing to the unified index. Nevertheless, there is still a consideration about how to reduce the massive memory consumption caused by the pointers stored in hash table [88].

B. Summary

Compared with separate index on FIB, PIT and Content Store, the amount of data structure schemes proposed for unified index is the least by only 4 related schemes. Among these, NameTree [105] and NPHT [32] focus on organizing PIT and FIB by a single index, and NLE&ICE [106] and BFAST [107] aim at the combination of the three tables. However, the effect of unified index is still open to question, as the three tables have really different requirements. For example, Content Store and PIT have high frequency for read operations and write operations, while FIB requires lots of read operations and few write operations. Moreover, for the special properties, Content Store must support the cache replacement policy, PIT has to perform timeout operation, and FIB has to support forwarding strategy. Furthermore, the algorithms of name matching in the three tables are also different. Therefore, it's not easy to propose a perfect unified index to satisfy all the requirements.

For the current data structure schemes proposed, BFAST has the more efficient performance. Through using an indicating pointer in each hash entry, BFAST achieves the union for three tables. Meanwhile, BFAST accelerates the lookup speed owing to storing the interface information into the hash entry. However, how to reduce the massive memory consumption caused by the pointers stored in hash table [88] is still a consideration.

IX. Issues, Challenges and Future Research Directions

In this section, through the analysis of the above schemes proposed, some issues, challenges and future research directions are found, discussed and concluded in five aspects which need to be resolved urgently. First and foremost, a novel data structure needs to be designed and implemented to meet all requirements of the forwarding plane. Secondly, the research on the better structure of Content Store is urgent in view of the importance of Content Store and the complication of its requirements. Thirdly, the necessity of a unified index of different tables needs to be discussed. Fourthly, combining with other contents in NDN research is expected. Finally, a unified benchmark is required to lay the foundation for the deployment of NDN. In the following, the five aspects are analyzed and discussed in detail.

A. Novel Data Structure

As analyzed in Section IV-VIII, trie, hash table, Bloom filter and skip list are the most widely used data structures implemented in NDN forwarding plane currently to store names logically and support different algorithms of name matching, but each of them has some disadvantages respectively. Just as previously described, trie can reduce the memory consumption of the hierarchical names stored in NDN router but has slow lookup speed relatively; hash table can implement the fast lookup but consumes more memory to ensure more accurate forwarding; Bloom filter can not only achieve the fast retrieval but also reduce the memory consumption, but cannot locate the memory address of the element; skip list can implement fast and simple insertion and deletion operations in any node but the lookup speed is still slower. Given that none of them can well satisfy all requirements of the forwarding plane in NDN, there has to be a trade-off between lookup speed and compressed storage. Therefore, a novel data structure needs to be designed and implemented to meet the requirements of both speed and storage. For example, Kraska et al., 2018 tentatively discussed the feasibility and schemes of replacing traditional index structures with learned models [110]. The key idea is that a model can learn the distribution of data being indexed by using machine learning technology to optimize storage space utilization and use the model to effectively predict the position or existence of data, which can provide benefits in lookup speed and compressed storage as shown in the experimental results. It may be of some help to improve the performance of content routers, but some problems must be solved due to the characteristics of variable-length names being indexed and dynamic write-heavy databases in NDN forwarding plane.

B. Better Structure of Content Store

Comparing the number of schemes of Content Store, PIT and FIB, an obvious discovery is that Content Store obtains the least attention. From 2010 to 2014, the research on PIT was always a hot topic and a series of designs including 8 schemes were presented, as PIT is a peculiar structure that differs from the routing table in IP. FIB was considered to

have more problems and obtained more attention after 2014, where there were 36 schemes proposed up to now. Conversely, the research on Content Store is little and only 7 schemes were proposed by far. This is the case that should not have happened, for the reason that Content Store as the core unit of NDN router has to process each incoming Interest or Data and its performance improvement is critical for maximizing the network efficiency. Furthermore, the 7 schemes cannot fully meet all the requirements of Content Store, fundamentally due to the complication of its requirements, which is that Content Store requires not only the fast name lookup but also the support for cache replacement policy as mentioned in Section III. Hence, in view of the importance of Content Store and the complication of its requirements, the research on better structure of Content Store, which can perfectly satisfy its requirements, is urgent in the future.

C. Unified Index

What needs to be discussed is the necessity of a unified index of different tables. As described in Section III, Content Store, PIT and FIB all implement retrieval according to the name of packet. Due to the similarity of the input of the index, the three tables can be organized as a single structure in memory with table entries recording their types in a unified structure theoretically. However, only 4 schemes achieve the unified index until now, which may also get the reason from Section III. Table II clearly shows that the requirements of the three tables are so different. For example, PIT requires a high access frequency, FIB has to support the name matching algorithm LNPM, as well as Content Store must support cache replacement policy. Different requirements mean that the most appropriate design of the data structure for each table is different, which makes the unified index hard to realize. In the future, it still needs to be further discussed whether a unified index that fully meets the requirements of different tables is needed.

D. Combining with Other Contents in NDN Research

Since NDN was proposed in 2010, design of the forwarding plane has received considerable attention and requirements of the forwarding plane such as routing, retrieval, and storage have been satisfied basically with the maturity of research. However, in addition to satisfying all the requirements, the design of NDN forwarding plane is best to further combine with other contents in NDN research. Firstly, it is expected to combine with other architecture research related to the forwarding plane in NDN. Taking Interest flooding attack (IFA) defense [22] in the domain of security as an example, the design of data structure on PIT is best to further consider the implementation of IFA defense, for PIT is target of the attack and can defense it by monitoring Interest occupancy or expiration rate, which is the key to IFA defense. Similarly in terms of congestion control [30] in the domain of network management, the design of data structure on Content Store is best to further take into account the implementation of congestion control, for Content Store is the key to congestion detection. Secondly, it is expected to combine with the research on deploying NDN in different network scenarios. When deploying NDN in different network scenarios, such as Mobile Ad Hoc Networks (MANET), Wireless Sensor Networks (WSN) and IoT, there are different design features and application performance of the forwarding plane. For example, when deploying NDN in MANET, it is necessary to first consider the lookup speed performance of the forwarding plane due to the mobility of nodes and high dynamics of the network. As for WSN and IoT, the devices are often small and affordable embedded devices with limited memory capacity, thus the memory consumption performance needs to be first considered. In view of the above, future works should not only focus on satisfying all the requirements of the forwarding plane but also further combine with other contents in NDN research, and then design the structure of the forwarding plane that is best suited for the actual deployment to contribute to the comprehensive implementation of NDN.

E. Unified Benchmark

As mentioned in the analysis of all schemes, the research of NDN forwarding plane lacks a unified benchmark, which makes all the schemes unable to be compared with each other. Among the experiments of these schemes proposed, there are significant differences in terms of the experimental setting. For example, some measure the performance based on CPU while some with GPU, even some implement the evaluation with FPGA. In addition, the single thread is often applied in the performance evaluation of the schemes, but some schemes perform the experiments with multi-thread. It is unfortunate that the experimental results of the lookup speed are quite different under different experimental settings, which makes it impossible to directly determine which of these schemes is better. Consequently, considering the systematic and long-term nature of the research, it is urgent to implement a unified benchmark to evaluate all kinds of schemes, which will lay the foundation for the deployment of NDN.

F. Summary

Through the analysis above, we conclude the issues, challenges and future research directions of NDN forwarding plane in five aspects. Firstly, given that none of trie, hash table, Bloom filter and skip list can well satisfy all requirements of the forwarding plane in NDN, a novel data structure needs to be designed and implemented, such as utilizing machine learning technology instead of traditional trie or hash table to index data. Secondly, we indicate that current research on data structures lacks attention to Content Store and better structure design of Content Store is urgent in view of the importance of Content Store and the complication of its requirements. Thirdly, considering the similarity of the input of the index but the different requirements of the three tables, the necessity of a unified index of different tables still needs to be discussed. Fourthly, future works should not only focus on satisfying all the requirements of the forwarding plane but also further combine with other contents in NDN research, such as other architecture research related to the forwarding plane in NDN and the research on deploying NDN in different network

scenarios. Finally, from the perspective of experiment and evaluation, we indicate that a unified benchmark to evaluate all kinds of schemes will contribute to the systematic and long-term nature of the research.

X. Conclusion

As the most promising proposal, NDN can directly pull contents based on the content names, irrespective of their hosting entity. However, due to complex names and unbounded namespace, the unique challenges are posed for the NDN forwarding plane. Since NDN was proposed, many data structures and algorithms have been proposed for Content Store, PIT and FIB to meet these challenges. To improve the efficiency of future research, this survey gives a train of more accurate requirements of NDN forwarding plane and across-the-board analyzes all the schemes proposed. Finally, some issues, challenges and directions in future research are discussed, and a conclusion is drawn that designing a novel data structure to meet all requirements of the forwarding plane and studying on a better structure of Content Store play important roles, while discussing the necessity of a unified index, combining with other contents in NDN research and implementing a unified benchmark are also required in this domain.

References

[1] G. Xylomenos, C. N. Ververidis, V. A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K. V. Katsaros, and G. C. Polyzos, "A survey of information-centric networking research," *IEEE Communications Surveys and Tutorials*, vol. 16, no. 2, pp. 1024–1049, 2014.

[2] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, and E. Yeh, "Named data networking (ndn) project," Xerox Palo Alto Research Center-PARC, Tech. Rep. NDN-0001, 2010, [Online]. Available: http://named-data.net/.

[3] V. Jacobson *et al.*, "Content-centric networking," 2009, [Online]. Available: http://www.ccnx.org/.

[4] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, P. Crowley, C. Papadopoulos, L. Wang, B. Zhang *et al.*, "Named data networking," *ACM SIGCOMM Computer Communication Review*, vol. 44, no. 3, pp. 66–73, 2014.

[5] Y. Cheng, A. Afanasyev, I. Moiseenko, B. Zhang, L. Wang, and L. Zhang, "Smart forwarding: A case for stateful data plane," Tech. Rep. NDN-0002, 2012, [Online]. Available: http://named-data.net/.

[6] S. Dharmapurikar, P. Krishnamurthy, and D. E. Taylor, "Longest prefix matching using bloom filters," *IEEE/ACM Transactions on Networking*, vol. 14, no. 2, pp. 397–409, 2006.

[7] C. Yi, A. Afanasyev, L. Wang, B. Zhang, and L. Zhang, "Adaptive forwarding in named data networking," *ACM SIGCOMM computer communication review*, vol. 42, no. 3, pp. 62–67, 2012.

[8] E. Fredkin, "Trie memory," *Communications of the ACM*, vol. 3, no. 9, pp. 490–499, 1960.

[9] T. Zink, "A survey of hash tables with summaries for ip lookup applications," 2009, [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:352-175851.

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

[11] W. Pugh, "Skip lists: A probabilistic alternative to balanced trees," *Commun. ACM*, vol. 33, no. 6, pp. 668–676, Jun. 1990. [Online]. Available: http://doi.acm.org/10.1145/78973.78977

[12] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, 2012.

[13] X. Jiang, J. Bi, G. Nan, and Z. Li, "A survey on information-centric networking: Rationales, designs and debates," *China Communications*, vol. 12, no. 7, pp. 1–12, 2015.

[14] D. Saxena, V. Raychoudhury, N. Suri, C. Becker, and J. Cao, "Named data networking: A survey," *Computer Science Review*, vol. 19, pp. 15–55, 2016.

[15] G. Zhang, Y. Li, and T. Lin, "Caching in information centric networking: A survey," *Computer Networks*, vol. 57, no. 16, pp. 3128–3141, 2013.

[16] I. Abdullahi, S. Arif, and S. Hassan, "Survey on caching approaches in information centric networking," *Journal of Network and Computer Applications*, vol. 56, no. 11, pp. 48–59, 2015.

[17] M. Zhang, H. Luo, and H. Zhang, "A survey of caching mechanisms in information-centric networking," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 3, pp. 1473–1499, 2015.

[18] B. Feng, H. Zhou, and Q. Xu, "Mobility support in named data networking: a survey," *Eurasip Journal on Wireless Communications and Networking*, vol. 2016, no. 1, p. 220, 2016.

[19] Y. Zhang, A. Afanasyev, J. Burke, and L. Zhang, "A survey of mobility support in named data networking," in *Computer Communications Workshops*, San Francisco, CA, USA, 2016, pp. 83–88.

[20] C. Fang, H. Yao, Z. Wang, W. Wu, X. Jin, and F. R. Yu, "A survey of mobile information-centric networking: Research issues and challenges," *IEEE Communications Surveys and Tutorials*, vol. PP, no. 99, pp. 1–1, 2018.

[21] E. G. Abdallah, H. S. Hassanein, and M. Zulkernine, "A survey of security attacks in information-centric networking," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 3, pp. 1441–1454, 2015.

[22] R. Tourani, S. Misra, T. Mick, and G. Panwar, "Security, privacy, and access control in information-centric networking: A survey," *IEEE Communications Surveys and Tutorials*, vol. PP, no. 99, p. 1, 2017.

[23] C. Fang, F. R. Yu, T. Huang, J. Liu, and Y. Liu, "A survey of green information-centric networking: Research issues and challenges," *IEEE Communications Surveys and Tutorials*, vol. 17, no. 3, pp. 1455–1472, 2015.

[24] C. Fang, F. R. Yu, T. Huang, and J. Liu, "A survey of energy-efficient caching in information-centric networking," *IEEE Communications Magazine*, vol. 52, no. 11, pp. 122–129, 2014.

[25] S. Rahel, A. Jamali, and S. E. Kafhali, "Energy-efficient on caching in named data networking: A survey," in *International Conference of Cloud Computing Technologies and Applications*, Rabat, Morocco, 2017, pp. 1–8.

[26] M. F. Bari, S. R. Chowdhury, R. Ahmed, R. Boutaba, and B. Mathieu, "A survey of naming and routing in information-centric networks," *IEEE Communications Magazine*, vol. 50, no. 12, pp. 44–53, 2012.

[27] X. Liu, Z. Li, P. Yang, and Y. Dong, "Information-centric mobile ad hoc networks and content routing: A survey," *Ad Hoc Networks*, vol. 58, pp. 255–268, 2016.

[28] W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, J. Thompson, J. Burke, B. Zhang, and L. Zhang, "Named data networking of things (invited paper)," in *IEEE First International Conference on Internet-Of-Things Design and Implementation*, Chengdu, Sichuan, China, 2016, pp. 117–128.

[29] Q. Chen, R. Xie, F. R. Yu, J. Liu, T. Huang, and Y. Liu, "Transport control strategies in named data networking: A survey," *IEEE Communications Surveys and Tutorials*, vol. 18, no. 3, pp. 2052–2083, 2016.

[30] Y. Ren, J. Li, S. Shi, L. Li, G. Wang, and B. Zhang, "Congestion control in named data networking-a survey," *Computer Communications*, vol. 86, pp. 1–11, 2016.

[31] M. Varvello, D. Perino, and L. Linguaglossa, "On the design and implementation of a wire-speed pending interest table," in *Computer Communications Workshops*, Turin, Italy, 2013, pp. 369–374.

[32] H. Yuan, T. Song, and P. Crowley, "Scalable ndn forwarding: Concepts, issues and principles," in *International Conference on Computer Communications and Networks*, Munich, Germany, 2012, pp. 1–9.

[33] S. Arianfar, P. Nikander, and J. Ott, "On content-centric router design and implications," in *Proceedings of the Re-Architecting the Internet Workshop*, Philadelphia, PA, USA, 2010, p. 5.

[34] H. Hwang, S. Ata, and M. Murata, "Realization of name lookup table in routers towards content-centric networks," in *Proceedings of the 7th International Conference on Network and Services Management*, Paris, France, 2011, pp. 353–357.

[35] H. Yuan and P. Crowley, "Performance measurement of name-centric content distribution methods," in *Symposium on Architectures for Networking and Communications Systems*, Brooklyn, New York, USA, 2011, pp. 223–224.

[36] D. Perino and M. Varvello, "A reality check for content centric networking," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, Toronto, Canada, 2011, pp. 44–49.

[37] M. Virgilio, G. Marchetto, and R. Sisto, "Pit overload analysis in content centric networks," in *Proceedings of the 3rd ACM SIGCOMM workshop on Information-centric networking*, Hong Kong, China, 2013, pp. 67–72.

[38] F. Li, F. Chen, J. Wu, and H. Xie, "Longest prefix lookup in named data networking: How fast can it be?" in *2014 9th IEEE International Conference on Networking, Architecture, and Storage*, Tianjin, China, 2014, pp. 186–190.

[39] G. Carofiglio, M. Gallo, L. Muscariello, and D. Perino, "Pending interest table sizing in named data networking," in *Proceedings of the 2nd International Conference on Information-Centric Networking*, San Francisco, CA, USA, 2015, pp. 49–58.

[40] C. A. Shue and M. Gupta, "Packet forwarding: Name-based vs. prefix-based," in *IEEE Global Internet Symposium*, Anchorage, AK, USA, 2007, pp. 73–78.

[41] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: Incrementally deployable icn," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 147–158, 2013.

[42] J. Shi, T. Liang, H. Wu, B. Liu, and B. Zhang, "Ndn-nic: Name-based filtering on network interface card," in *Proceedings of the 2016 conference on 3rd ACM Conference on Information-Centric Networking*, Kyoto, Japan, 2016, pp. 40–49.

[43] M. Li, "Recent advances in named data caching and routing," 2013, [Online]. Available: https://www.cse.wustl.edu/~jain/cse570-13/ftp/ndnrc/index.html.

[44] E. Yeh, T. Ho, Y. Cui, R. Liu, M. Burd, and D. Leong, "Forwarding, caching and congestion control in named data networks," 2013, [Online]. Available: https://arxiv.org/pdf/1310.5569.pdf.

[45] G. Rossini and D. Rossi, "Coupling caching and forwarding: benefits, analysis, and implementation," in *Proceedings of the 1st ACM Conference on Information-Centric Networking*. New York, NY, USA, 2014, pp. 127–136.

[46] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: From millions to billions," in *Proceedings of the 2nd International Conference on Information-Centric Networking*, San Francisco, CA, USA, 2015, pp. 19–28.

[47] M. Xie, I. Widjaja, and H. Wang, "Enhancing cache robustness for content-centric networking," in *IEEE INFOCOM*, Orlando, FL, USA, 2012, pp. 2426–2434.

[48] H. Yuan, "Data structures and algorithms for scalable ndn forwarding," *Dissertations and Theses - Gradworks*, 2015, [Online]. Available: https://openscholarship.wustl.edu/cgi/viewcontent.cgi?article=1143&context=eng_etds.

[49] S. Tarkoma, C. E. Rothenberg, E. Lagerspetz *et al.*, "Theory and practice of bloom filters for distributed systems," *IEEE Communications Surveys and Tutorials*, vol. 14, no. 1, pp. 131–155, 2012.

[50] T. Pan, T. Huang, J. Liu, J. Zhang, F. Yang, S. Li, and Y. Liu, "Fast content store lookup using locality-aware skip list in content-centric networks," in *Computer Communications Workshops*, San Francisco, CA, USA, 2016, pp. 187–192.

[51] Y. Wang, H. Dai, J. Jiang, K. He, W. Meng, and B. Liu, "Parallel name lookup for named data networking," in *Global Telecommunications Conference (GLOBECOM 2011)*, Houston, USA, 2011, pp. 1–5.

[52] H. Dai and B. Liu, "Consert: Constructing optimal name-based routing tables," *Computer Networks*, vol. 94, pp. 62–79, 2016.

[53] S. H. Bouk, S. H. Ahmed, and D. Kim, "Hierarchical and hash based naming with compact trie name management scheme for vehicular content centric networks," *Computer Communications*, vol. 71, pp. 73–83, 2015.

[54] J. Lee and H. Lim, "A new name prefix trie with path compression," in *IEEE International Conference on Consumer Electronics-Asia*, Seoul, Korea, 2016, pp. 1–4.

[55] 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 *2012 IEEE 32nd International Conference on Distributed Computing Systems (ICDCS)*, Macau, China, 2012, pp. 688–697.

[56] S. Feng, M. Zhang, R. Zheng, and Q. Wu, "A fast name lookup method in ndn based on hash coding," in *International Conference on Mechatronics and Industrial Informatics*, Cambridge, United Kingdom, 2015, pp. 575–580.

[57] D. Saxena, V. Raychoudhury, C. Becker, and N. Suri, "Reliable memory efficient name forwarding in named data networking," in *2016 IEEE Intl Conference on Computational Science and Engineering*, Paris, France, 2016, pp. 48–55.

[58] J. Seo and H. Lim, "Bitmap-based priority-npt for packet forwarding at named data network," *Computer Communications*, vol. 130, pp. 101–112, 2018.

[59] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang, "Wire speed name lookup: A gpu-based approach," in *Usenix Conference on Networked Systems Design and Implementation*, Lombard, IL, USA, 2013, pp. 199–212.

[60] Y. Li, D. Zhang, X. Yu, W. Liang, J. Long, and H. Qiao, "Accelerate ndn name lookup using fpga: Challenges and a scalable approach," in *International Conference on Field Programmable Logic and Applications*, Munich, Germany, 2014, pp. 1–4.

[61] Y. Tan and S. Zhu, "Efficient name lookup scheme based on hash and character trie in named data networking," in *2015 12th Web Information System and Application Conference (WISA)*, Jinan, Shandong, China, 2015, pp. 130–135.

[62] D. Li, J. Li, and Z. Du, "An improved trie-based name lookup scheme for named data networking," in *Computers and Communication (ISCC)*, Messina, Italy, 2016, pp. 1294–1296.

[63] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang, "A fast and memory-efficient trie structure for name-based packet forwarding," in *IEEE International Conference on Network Protocols*, Cambridge, UK, 2018.

[64] D. Saxena and V. Raychoudhury, "N-fib: Scalable, memory efficient name-based forwarding," *Journal of Network and Computer Applications*, vol. 76, pp. 101–109, 2016.

[65] W. Quan, C. Xu, A. V. Vasilakos, J. Guan, H. Zhang, and L. A. Grieco, "Tb2f: Tree-bitmap and bloom-filter for a scalable and efficient name lookup in content-centric networking," in *2014 IFIP Networking conference*, Trondheim, Norway, 2014, pp. 1–9.

[66] Y. Wang, H. Dai, T. Zhang, W. Meng, J. Fan, and B. Liu, "Gpu-accelerated name lookup with component encoding," *Computer Networks*, vol. 57, no. 16, pp. 3165–3177, 2013.

[67] T. Zhang, Y. Wang, T. Yang, J. Lu, and B. Liu, "Ndnbench: A benchmark for named data networking lookup," in *IEEE Global Communications Conference (GLOBECOM)*, Atlanta, GA, USA, 2013, pp. 2152–2157.

[68] Y. Li, D. Zhang, X. Yu, J. Long, and W. Liang, "From gpu to fpga: A pipelined hierarchical approach to fast and memory-efficient ndn name lookup," in *IEEE International Symposium on Field-Programmable Custom Computing Machines*, Boston, Massachusetts, USA, 2014, pp. 106–106.

[69] H. Yuan, P. Crowley, and T. Song, "Enhancing scalable name-based forwarding," in *Symposium on Architectures for Networking and Communications Systems*, Beijing, China, 2017, pp. 60–69.

[70] W. So, A. Narayanan, D. Oran, and Y. Wang, "Toward fast ndn software forwarding lookup engine based on hash tables," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, Austin, TX, USA, 2012, pp. 85–86.

[71] D. Xu, H. Zhang, Y. Sun, and Y. Liu, "Scalable multi-hash name lookup method for named data networking," *Journal of Harbin Institute of Technology (New Series)*, vol. 22, no. 6, pp. 62–68, 2015.

[72] R. Shubbar and M. Ahmadi, "Efficient name matching based on a fast two-dimensional filter in named data networking," *International Journal of Parallel, Emergent and Distributed Systems*, no. 4, pp. 1–19, 2017.

[73] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and dos-resistant forwarding with hash tables," in *Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems*, San Jose, CA, USA, 2013, pp. 215–226.

[74] Y. Wang, D. Tai, T. Zhang, J. Lu, B. Xu, H. Dai, and B. Liu, "Greedy name lookup for named data networking," *ACM SIGMETRICS Performance Evaluation Review*, vol. 41, no. 1, pp. 359–360, 2013.

[75] H. Yuan and P. Crowley, "Reliably scalable name prefix lookup," in *Symposium on Architectures for Networking and Communications Systems*, Oakland, CA, USA, 2015, pp. 111–121.

[76] Y. Wang, Z. Qi, H. Dai, H. Wu, K. Lei, and B. Liu, "Statistical optimal hash-based longest prefix match," in *Symposium on Architectures for Networking and Communications Systems*, Beijing, China, 2017, pp. 153–164.

[77] W. So, A. Narayanan, D. Oran, and M. Stapp, "Named data networking on a router: forwarding at 20gbps and beyond," *ACM SIGCOMM Computer Communication Review*, vol. 43, no. 4, pp. 495–496, 2013.

[78] 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 *IEEE 22nd International Symposium of Quality of Service*, Hong Kong, China, 2014, pp. 198–207.

[79] 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 *IEEE INFOCOM*, Turin, Italy, 2013, pp. 95–99.

[80] C. Munoz, L. Wang, E. Solana, and J. Crowcroft, "I(fib)f: Iterated bloom filters for routing in named data networks," in *International Conference on Networked Systems*, Marrakech, Morocco, 2017, pp. 1–8.

[81] S. Ding, Z. Chen, and Z. Liu, "Parallelizing fib lookup in content centric networking," in *2012 Third International Conference on Networking and Distributed Computing*, Hangzhou, China, 2012, pp. 6–10.

[82] M. Fukushima, A. Tagami, and T. Hasegawa, "Efficient lookup scheme for non-aggregatable name prefixes and its evaluation," *IEICE Transactions on Communications*, vol. 96, no. 12, pp. 2953–2963, 2013.

[83] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue, "Caesar: a content router for high-speed forwarding on content names," in *Proceedings of the tenth ACM/IEEE symposium on Architectures for networking and communications systems*, Los Angeles, CA, USA, 2014, pp. 137–148.

[84] W. Quan, C. Xu, J. Guan, H. Zhang, and L. A. Grieco, "Scalable name lookup with adaptive prefix bloom filter for named data networking," *IEEE Communications Letters*, vol. 18, no. 1, pp. 102–105, 2014.

[85] J. Lee, M. Shim, and H. Lim, "Name prefix matching using bloom filter pre-searching for content centric network," *Journal of Network and Computer Applications*, vol. 65, pp. 36–47, 2016.

[86] K. Shimazaki, T. Aoki, T. Hatano, T. Otsuka, A. Miyazaki, T. Tsuda, and N. Togawa, "Hash-table and balanced-tree based fib architecture for ccn routers," in *2016 International SoC Design Conference (ISOCC)*, Jeju, South Korea, 2016, pp. 67–68.

[87] V. Manghnani, "Length indexed bloom filter based forwarding in content centeric networking," 2016, [Online]. Available: https://pdfs.semanticscholar.org/8f67/753e6ea3746a64a6a99d3c5836c22bbc81c7.pdf.

[88] H. Dai, J. Lu, Y. Wang, T. Pan, and B. Liu, "Bfast: High-speed and memory-efficient approach for ndn forwarding engine," *IEEE/ACM Transactions on Networking*, vol. 25, no. 2, pp. 1235–1248, 2017.

[89] Z. Li, K. Liu, D. Liu, H. Shi, and Y. Chen, "Hybrid wireless networks with fib-based named data networking," *EURASIP Journal on Wireless Communications and Networking*, vol. 2017, no. 1, p. 54, 2017.

[90] Z. Li, Y. Xu, K. Liu, X. Wang, and D. Liu, "5g with b-mafib based named data networking," *IEEE Access*, vol. 6, pp. 30 501–30 507, 2018.

[91] K. Chandana, S. B. Patil, N. Taranath, and P. Patil, "Efficient lookup for nlapb in named data networking," in *International Conference on Applied and Theoretical Computing and Communication Technology*, Palo Alto, USA, 2015, pp. 27–32.

[92] H. Lim, M. Shim, and J. Lee, "Name prefix matching using bloom filter pre-searching," in *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, Washington, DC, USA, 2015, pp. 203–204.

[93] W. Yu and D. Pao, "Hardware accelerator to speed up packet processing in ndn router," *Computer Communications*, vol. 91, pp. 109–119, 2016.

[94] H. Dai, B. Liu, Y. Chen, and Y. Wang, "On pending interest table in named data networking," in *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*, Austin, TX, USA, 2012, pp. 211–222.

[95] D. Saxena and V. Raychoudhury, "Radient: Scalable, memory efficient name lookup algorithm for named data networking," *Journal of Network and Computer Applications*, vol. 63, pp. 1–13, 2016.

[96] H. Yuan and P. Crowley, "Scalable pending interest table design: From principles to practice," in *IEEE INFOCOM*, Toronto, Canada, 2014, pp. 2049–2057.

[97] W. You, B. Mathieu, P. Truong, J.-F. Peltier, and G. Simon, "Dipit: A distributed bloom-filter based pit table for ccn nodes," in *2012 21st International Conference on Computer Communications and Networks*, Munich, Germany, 2012, pp. 1–7.

[98] Z. Li, J. Bi, S. Wang, and X. Jiang, "Compression of pending interest table with bloom filter in content centric network." in *International Conference on Future Internet Technologies*, Seoul, South Korea, 2012, p. 46.

[99] Z. Li, K. Liu, Y. Zhao, and Y. Ma, "Mapit: an enhanced pending interest table for ndn with mapping bloom filter," *IEEE Communications Letters*, vol. 18, no. 11, pp. 1915–1918, 2014.

[100] W. You, B. Mathieu, P. Truong, J.-F. Peltier, and G. Simon, "Realistic storage of pending requests in content-centric network routers," in *IEEE International Conference on Communications in China*, Beijing, China, 2012, pp. 120–125.

[101] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "Nfd developer's guide," no. NDN-0021, 2016, [Online]. Available: http://named-data.net/.

[102] Y. Thomas, G. Xylomenos, C. Tsilopoulos, and G. C. Polyzos, "Object-oriented packet caching for icn," in *Proceedings of the 2nd International Conference on Information-Centric Networking*, San Francisco, CA, USA, 2015, pp. 89–98.

[103] R. Zhang, J. Liu, T. Huang, T. Pan, and L. Wu, "Adaptive compression trie based bloom filter: Request filter for ndn content store," *IEEE Access*, vol. PP, no. 99, p. 1, 2017.

[104] R. Hou, L. Zhang, T. Wu, T. Mao, and J. Luo, "Bloom-filter-based request node collaboration caching for named data networking," *Cluster Computing*, no. 4, pp. 1–12, 2018.

[105] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto *et al.*, "Nfd developer's guide," Tech. Rep. NDN-0021, 2014, [Online]. Available: http://named-data.net/.

[106] M. Murata and A. Ooka, "Hardware design and evaluation of cam-based high-speed ccn router," 2014, [Online]. Available: http://www.anarg.jp/achievements/web2013/paper/a-ooka14mastersthesis-CCNRouterHardware.pdf.

[107] H. Dai, J. Lu, Y. Wang, and B. Liu, "Bfast: Unified and scalable index for ndn forwarding architecture," in *IEEE Conference on Computer Communications*, Hong Kong, China, 2015, pp. 2290–2298.

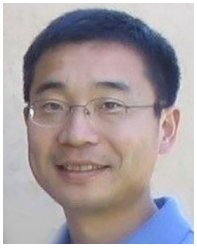
[108] A. Ooka, S. Atat, K. Inoue, and M. Murata, "Design of a high-speed content-centric-networking router using content addressable memory," in *2014 IEEE Conference on Computer communications workshops*, Shanghai, China, 2014, pp. 458–463.

[109] A. Ooka, S. Ata, K. Inoue, and M. Murata, "High-speed design of conflictless name lookup and efficient selective cache on ccn router," *IEICE Transactions on Communications*, vol. 98, no. 4, pp. 607–620, 2015.

[110] T. Kraska, A. Beutel, E. H. Chi, J. Dean, and N. Polyzotis, "The case for learned index structures," in *Proceedings of the 2018 International Conference on Management of Data*, Houston, TX, USA, 2018, pp. 489–504.

Zhuo Li received Ph.D. degree from Tianjin University, China in 2015 and M.S. degree from East China Normal University, China in 2010. Before 2017, he spent one year as a Post-Doctoral Research Associate in the Computer Science Department at the University of Arizona. He is currently working in the School of Microelectronics at Tianjin University. His main research interests include router architecture, fast packet processing, wireless communications and Future Internet Architecture, e.g., Named Data Networking and Information-Centric Networking. He now serves as the reviewer for IEEE Intelligent Transportation Systems Transactions and Magazine, IEEE Access, IEEE Transactions on Mobile Computing and IEEE Communications Letters.

Yaping Xu is a master candidate in the School of Microelectronics, Tianjin University. She got the B.S. degree in communication engineering from Harbin Engineering University, China, in 2016. Her research interests mainly include router architecture, fast packet processing and Future Internet Architecture, e.g., Named Data Networking and Information-Centric Networking.



Beichuan Zhang received Ph.D. degree from UCLA in 2003, and B.S. degree from Peking University, China in 1995. Before 2005, he spent two years as a postdoc at USC/ISI-East and UCLA. He is currently working in the Computer Science Department at the University of Arizona. His research area is computer networks in general. He has been working on Internet routing architectures and protocols, green networks, routing security, and Internet content distribution. Dr. Zhang is the receipient of the Applied Networking Research Prize by ISOC and IRTF in 2011, and the Best Paper Award at ICDCS in 2005.

Liu Yan is a master candidate in the School of Microelectronics, Tianjin University. She got the B.S. degree in electronic information engineering (English intensive) from Dalian University of Technology, China, in 2017. Her research interests mainly include router architecture, fast packet processing and Future Internet Architecture, e.g., Named Data Networking and Information-Centric Networking.

Kaihua Liu is currently a full professor with the School of Microelectronics, Tianjin University, China. He received the B.S. degree in Semiconductor physics and devices, the M.S. degree in circuits and systems and Ph.D. degree in signal and information processing from Tianjin University, in 1981, 1991 and 1999, respectively. His current research interests include radio frequency identification, digital signal processing and Future Internet Architecture. He is a member of IEEE.