PBC: Effective Prefix Caching for Fast Name Lookups

Chuwen Zhang*, Yong Feng*, Haoyu Song†, Beichuan Zhang‡ Yi Wang§¶, Ying Wan*, Wenquan Xu*, Bin Liu*¶
*Department of Computer Science and Technology, Tsinghua University, China. †Futurewei Technologies, USA.
‡University of Arizona, USA. §Southern University of Science and Technology, China. ¶Peng Cheng Lab, Shenzhen, China.

***Abstract*—Name lookup based on the Longest Prefix Match (LPM) is a basic function in many network applications. Caching is usually used to speed up the lookups. However, caching prefixes for LPM has a unique challenge: one needs to guarantee a cached prefix is indeed the longest for correctness. To achieve this, existing solutions have to cache either the entire prefix trie-branch or only the leaf nodes, which undermines cache utilization and thus reduce the hit ratio. In this paper, we propose Plus-Bitmap Caching (PBC), which associates a bitmap to each cached prefix to denote the existence or absence of any longer prefix in the main table. This bitmap not only guarantees the correctness of LPM lookup, but also minimizes the extra information stored in cache. Meanwhile, cache consistency for prefix updates can be efficiently maintained. Experimental results show that, compared with previous work, PBC increases cache hit ratio by 16% over a wide range of cache size, and exhibits a more steady performance when more non-leaf prefixes are hit or a prefix is hit by more different names. PBC is a general approach that can be applied to other LPM-based applications**

I. Introduction

Name lookup is a basic function for many network applications such as domain name resolution and content search. There are two types of name lookup: the Exact Match (EM) and the Longest Prefix Match (LPM). The former is widely used in technical fields such as search engine, data center, storage system, and web application. The latter is becoming popular because of the rise of Named Data Networking (NDN) [1]. NDN packets carry content names instead of IP addresses. Routers forward NDN packets by name lookups in a name prefix table. While address lookup in IP routers is also based on LPM, name lookup in NDN is more challenging because 1) IP addresses are short and fixed-length, but names are hierarchical strings with unbounded and variable length; and 2) the size of an IP prefix table is moderate (e.g., $< 10^6$), but the number of prefixes in a name table can be very large (e.g., $> 10^8$) [2]. The ever increasing traffic makes the line speed forwarding in NDN routers harder and harder.

Therefore, LPM-based name lookup is an active research topic. Hardware-based schemes with TCAM [3], [4] or GPU [5], [6] support fast lookup with massive parallelism but entail high cost and low update efficiency. Software-based schemes, on the other hand, are more flexible, scalable, and cost effective, but need careful algorithm designs, which draw more attention on research. As the basic data structure for name lookups, a name trie [7], [8] can support incremental updates easily, but cannot achieve high lookup performance, due to the memory accesses required for traversing the trie. Some algorithms resort to hash table or Bloom Filter [9]–[13] to improve the lookup performance, at the cost of scalability.

ISBN 978-3-903176-28-7

One major hurdle for fast software-based name lookup is that the limited CPU cache or fast on-chip memory cannot accommodate the entire lookup table data structure. Hence, some name lookups may require multiple main memory (e.g., DRAM) accesses which limit the throughput. A basic remedy is to only cache popular prefixes on chip. Since the content requests in NDN generally follow a long tail distribution [14], [15], the cache hit ratio is supposed to be high.

However, a simple cache will not work because a cached prefix may not be the longest matching prefix for a name, so a cache hit does not cancel the need for searching in the main table. Several designs try to address this issue [16], usually requiring a large amount of extra information to be stored in cache, which lowers the efficiency of cache utilization, and in turn affects the overall lookup performance.

In this paper, we propose Plus-Bitmap Caching (PBC) to solve this problem. PBC associates a small bitmap with each cached prefix to indicate the existence or absence of any more specific prefix in the main table. With the help of the bitmaps, unnecessary slow main memory accesses are avoided. The use of bitmap is more space efficient than the other caching schemes, so space can be saved to cache more prefixes. Compared with the other caching schemes, PBC increases the hit ratio by up to 16% and maintains a more stable performance with different name tables and lookup request patterns. In a single-thread software implementation, PBC can increase the lookup throughput by 4.78 times and reduce the delay by 80% on the basis of a trie-based main table. It can support uninterrupted lookup with fast updates by parallel.

The rest of this paper is organized as follows. Section II surveys the related work. Section III introduces the PBC scheme. Section IV details the PBC-enabled system. Section V presents the evaluation. Section VI concludes the work.

II. Background and Relate Work

A. Existing LPM Name Lookup schemes

Depending on the application scenario, a trie [7], [8], [17] can be bitwise, character-based, or substring-based. The time complexity of lookups and updates in a trie is determined by its depth. Hash-based schemes [9], [10] need to allocate a hash

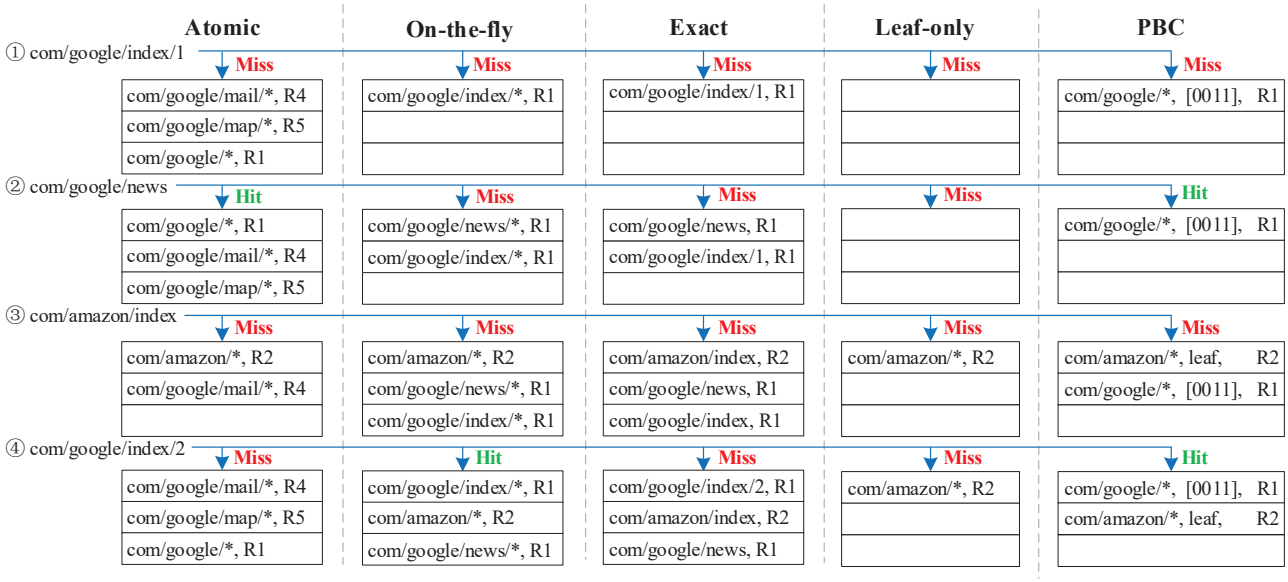


Fig. 1: Behavior of the name caching schemes. The cache size is 3 and the replacement policy is Least Recently Used (LRU)

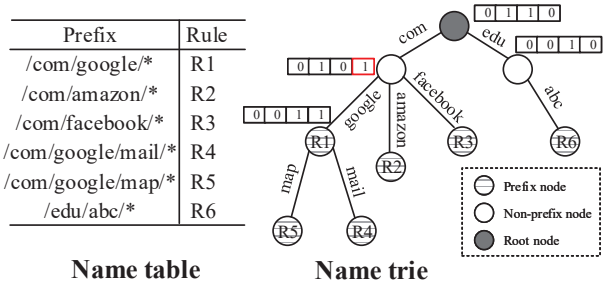


Fig. 2: A name table and the corresponding trie

table for each prefix length and search them sequentially. The time complexity of lookups is linear with the name lengths, denoted by l. A binary search on the hash tables [11], [12] reduces the lookup complexity to $O(\lg(l))$ by trading off the update complexity. The latest work, BFAST [13], applies Counting Bloom Filter to accelerate name lookup in some cases. However, the scalability of BFAST is a concern.

B. Name Caching Schemes

We now review the LPM schemes using a cache as an accelerator. Liu proposed several cache designs for binary IP trie [18], which employ the prefix leaf-pushing technique for correctness. Wan *et al* proposed a cache design that avoids the indiscriminate leaf pushing by generating virtual leaf prefixes on demand [19]. However, the work does not pay enough attention on efficient cache-consistency maintenance, which can influence the lookup performance significantly. Besides, Rottenstreich *et al* use lossy compression to create a LPM cache with much less memory than the theoretical size limits.

Inspired by the caching schemes for IP prefixes, some caching schemes for name lookup are discussed in [16]. We use the name table and the corresponding trie shown in Fig. 2 to illustrate the behaviors of different name caching schemes under the lookup request shown in Fig. 1. Each branch in the trie is a name segment. A name segment is a sub-string between two adjacent slashes in a name. We assume the cache has only three entries. We adopt the terms used in [18]: If a longer matching prefix exists, then relatively, the shorter one is a "sub-prefix" and the longer one is a "super-prefix".

1) Atomic Caching: In this scheme, a hit prefix must be accompanied by all its super-prefixes in the cache; To evict a prefix, all its sub-prefixes in the cache must be evicted as well. Obviously, the cache utilization rate is low and cache replacement is complicated. In Fig. 1, the first lookup request */com/google/index/1* hits the prefix */com/google/**, so it and its two super-prefixes are inserted to the cache. When admitting the prefix */com/amazon/** that the third lookup request */com/amazon/index* hits, the cache must evict the tail entry of [*/com/google/map/**, R5] as well as the entry of its super-prefix */com/google/**. Cache-consistency is easy to maintain since the scheme only caches original prefixes.

2) On-the-fly Caching: This scheme [16] is essentially an extension of the scheme in [19]. If a lookup request matches an intermediate prefix node, a longer virtual prefix is generated and cached. For example, in Fig. 1, the intermediate prefix node */com/google/** is matched for the first lookup request */com/google/index/1*, so the on-the-fly cache generates and caches a virtual prefix in the entry [*/com/google/index/**, R1]. Virtual prefixes may introduce redundancy, e.g., the cached virtual entries [*/com/google/news/**, R1] and [*/com/google/index/**, R1] after the fourth lookup request in Fig. 1. Moreover, cache-consistency is more difficult to maintain. For example, if an update modifies the rule associated with a prefix, it needs to find all the related virtual prefixes in the cache and update them.

3) Exact Caching: This scheme only caches the full name and the rule derived from the longest matching prefix. For example, the first lookup request in Fig. 1 generates a cache entry [*/com/google/index/1*, R1]. Although simple, the cache utilization is not efficient. The cache-consistency maintenance is as complex as that for the on-the-fly cache.

4) Leaf-only Caching: This scheme, as the name suggests, only caches leaf prefixes. This can lead to a low cache hit ratio if most lookup requests only match intermediate prefixes. As shown in the example in Fig. 1, leaf-only cache fails to make full use of cache space and has a low hit ratio. The cache-consistency needs to take extra care when an update makes a cached leaf prefix intermediate.

Tab. I summarizes the properties of the above caching schemes and PBC.

III. PBC Construction and Analysis

A. PBC Overview

The above caching schemes either use the cache space inefficiently or fail to cache the intermediate prefixes. Ideally, a caching scheme should have the following two properties: 1) it can cache each prefix individually regardless of the prefix type, and 2) if a cached prefix is indeed the longest matching prefix of a name, it should be able to tell that and avoid the unnecessary search in the main table. PBC strives to achieve these goals. In PBC, each non-leaf trie node is associated with a bitmap. All the immediate child nodes of a trie node are summarized in its bitmap, A prefix node, if matched in the main table according to LPM, will be cached along with its bitmap. When a name lookup hits a prefix in the cache, the attached bitmap is tested. If the corresponding bit is '1', a further search in the main table is needed. Otherwise, the current prefix is final.

We use the name table and the corresponding trie in Fig. 2 to illustrate the PBC caching and lookup process. The trie contains two types of nodes: prefix nodes and non-prefix nodes. If the prefix node [*/com/google/**, R1] is to be cached, its two immediate child nodes, [*/com/google/map/**, R5] and [*/com/google/mail/**, R4] (which happen to be prefix nodes as well), need to be summarized in its bitmap. Assume the bitmap is 4-bit long and the name segment *map* and *mail* are encoded to bit 2 and 3, respectively (i.e., the bitmap is [0011]). If the cache only contains an entry [*/com/google/**, R1] and its bitmap, the lookup request for the name */com/google/map* will hit it. After testing the bitmap and getting the positive return value, a further search in the main table is followed, as a longer matching prefix may exist. For another example, if the lookup request for name */com/google/earth* arrives, the same bitmap is tested and it is likely to get a negative return value, so R1 will be the final matching rule.

While PBC can provide information about the potential matching super-prefixes, a false cache miss happens when the matching prefix in the cache is not thought to be the longest matching prefix but it actually is, resulting in an extra search in the main table. False cache misses are caused by two cases. First is from the bitmap construction. In the above example, if the name segment *earth* is also encoded to bit 2 or 3 in the bitmap, false cache miss occurs. Therefore, unless all the possible immediate name segments (not only the immediate child nodes) for a prefix node are encoded to unique indexes in the bitmap, this kind of false cache miss cannot be avoided. However, it is impractical as the name set in NDN cannot be known entirely in advance. Even if it is known, the collision-free encoding costs large memory or long time on bitmap testing, such as the Perfect Hash. For PBC, we use general hash functions to encode each immediate child node to construct a bitmap, which is essentially a single-hash Bloom filter [20], so the false positive of the bitmap causes false cache misses.

Another more subtle case can also cause false cache misses. The bitmap only summarizes the immediate child nodes and these nodes are not necessarily prefix nodes. A name may match one of these nodes but fail to match any longer prefix node. In this case, the bitmap test still returns a positive answer, leading to a false cache miss. For example, in Fig. 2, assume the root prefix stores the rule R0 and is cached. The lookup request for the name */com/youtube* will lead the search to the left sub-trie. However, the search cannot go any further and a false cache miss occurs.

Although false cache misses do not cause errors for name lookup, it reduces the cache hit ratio. To keep a low False Cache Miss Ratio (FCMR), The key measure we take is to lower the False Positive Ratio (FPR) of the bitmap, which can be directly translated into a lower FCMR. Constructing fixed-size bitmaps to reduce the FPR is not reasonable, as the trie is usually highly unbalanced in reality, i.e., most parts are quite sparse, but some nodes, e.g., the root, have thousands of immediate child nodes. The analysis on the real name table from DMOZ [21] shows that about 95% prefixes have no more than 8 immediate child nodes and 99.9% prefixes have no more than 512. Therefore, we use adaptive bitmaps to keep a low FPR.

Assume the bitmap size is m and the number of immediate child nodes is n. The proportion of bit '1' in the bitmap is at most n/m, so the FPR for a non-member name segment is at most n/m. This implies that, to maintain the same FPR θ, we should make the bitmap size proportional to the number of immediate child nodes. On the other hand, for implementation convenience and lookup performance, the size of the bitmap is better to be the power of 2.

Based on these considerations, we take the maximum FPR to $\theta = 12.5\%$ as an example. We allocate 64-bit bitmaps to nodes with up to 8 immediate child nodes. We double the bitmap size for nodes with up to two times more immediate child nodes. The bitmap size continues to increase for nodes with more immediate child nodes, but is capped at 4096 bits. According to the statistic result that 99.9% prefixes have no more than 512 immediate child nodes, setting the maximum bitmap size to 4096 is reasonable.

We can analyze the bitmap as a single-hash Bloom filter against an optimal k-hash Standard Bloom Filter (SBF). The false positive for SBF is,

$$\left[1-\left(1-\frac{1}{m'}\right)^{nk}\right]^k \approx \left(1-e^{-\frac{nk}{m'}}\right)^k, \qquad (1)$$

where m' denotes the size of the SBF and n denotes the size of the member set. If the optimal value $k = \frac{m' \ln 2}{n}$ is taken,

TABLE I: Comparison of the name caching schemes

Scheme	Objects cached when cache miss	Extra operation for cache replacement	Cache-consistency complexity	Variant of Eq. 3	Key factors for cache hit ratio
Atomic	The hit prefix and all its super-prefixes	Admit super-prefixes Evict sub-prefixes	Up to 1 LPM cache lookup	$C = \sum_{i \in \mathcal{P}} \lvert\mathcal{D}_i\rvert \cdot (1 - e^{-\lambda_i T}) e^{\sum_{j \in \mathcal{U}_i} -\lambda_j T}$	Leaf prefix hit ratio
Exact	The exact lookup name	None	Traverse the cache or search the recorded virtual prefixes	$C = \lvert\mathcal{N}\rvert - \sum_{n \in \mathcal{N}} e^{-\lambda_n T}$	Replicated request name ratio
On-the-fly	The hit leaf prefix or virtual prefix	None	Traverse the cache or search the recorded virtual prefixes	$C \approx \lvert\mathcal{L}\rvert + \lvert\mathcal{N}_{\overline{\mathcal{L}}}\rvert - \sum_{i \in \mathcal{L}} e^{-\lambda_i T} - \sum_{n \in \mathcal{N}_{\overline{\mathcal{L}}}} e^{-\lambda_n T}$	Leaf and virtual prefix hit ratio
Leaf-only	The hit leaf prefix	None	Up to 1 LPM cache lookup	$C = \lvert\mathcal{L}\rvert - \sum_{i \in \mathcal{L}} e^{-\lambda_i T}$	Leaf prefix hit ratio
PBC	The hit prefix and its bitmap	None	Up to 2 EM cache lookups	$C \approx (\lvert\mathcal{P}\rvert - \sum_{i \in \mathcal{P}} e^{-\lambda_i T})$	FCMR

TABLE II: List of main notations

$\mathcal{P}$	set of prefixes
$\mathcal{P}_{\mathcal{L}}$	set of leaf prefixes
$\mathcal{N}$	set of exact names
$\mathcal{N}_{\overline{\mathcal{L}}}$	set of exact names that match non-leaf prefixes
$\mathcal{U}_i$	set of sub-prefixes of prefix i
$\mathcal{D}_i$	set of super-prefixes of prefix i
τ_i	time interval of two consecutive requests of object i
T	maximum sojourn time for object i
i_p	the parent object of object i
$A_i(t)$	age of the last request for object i at time t
λ_i	average request arrival rate for object i
C	total cache size
S_i	cache size occupied by object i
$I_i(t)$	indicator of whether object i is in the cache at time t
$\beta_i(t)$	occupancy probability for object i in the cache at time t
h_i	hit probability for an object i

the value of $m^{'}$ is $-\frac{n \ln \theta}{(\ln 2)^2}$. Since $\theta \approx n/m$, the ratio of $m^{'}$ and m is,

$$\frac{m^{'}}{m} \approx -\frac{\theta \ln \theta}{(\ln 2)^2}, \tag{2}$$

When setting $\theta = 12.5\%$, the ratio is 0.54, which means the size of the bitmap is about twice larger than the optimal SBF. We consider this as a trade-off for using just a single hash function, which is important for the cache performance.

B. PBC Behavior

We describe the behavior of PBC using the name lookup requests shown in Fig. 1. For the first lookup request */com/google/index/1*, since the cache is still empty, the matching prefix (along with its bitmap) is admitted to the cache after the search in the main table. The second lookup request */com/google/news* results in a cache hit, assuming no false positive happens when testing the bitmap. The third lookup request */com/amazon/index* misses the cache but matches a leaf prefix in the main table, so PBC admits it with a leaf flag. The fourth lookup request */com/google/index/1* also hits the cached entry [*/come/google/**, R1], assuming no false positive happens when testing the bitmap.

C. Theoretical Cache Hit Ratio

Now we analyze the theoretical cache hit ratio of PBC. The main notations are listed in Table II. We use the approximation described in [22], [23] to analyze the cache hit ratios of the five caching schemes, which is validated by both theories [24] and experiments [22]. The approximation assumes the matching on a certain prefix follows a Poisson process, in which the inter-request time follows an independent and identical Exponential distribution, and the occurrence probability of a certain prefix follows a Zipf distribution, in which the probability of i-th most popular prefix is proportional to $1/i^{\alpha}$, where the system parameter α is recommended to be between 0.8 to 1. Such a configuration obeys the situation of Internet [14], [15].

The approximation works as follows. As in [25], we introduce the indicator random variable $I_i(t)$, which takes 1 if object i is in cache, and 0 otherwise, to establish the equality relationship between the cache size C and $\beta_i(t)$,

$$C = \sum \mathbb{E}[I_i(t) S_i] = \sum \beta_i(t) S_i. \tag{3}$$

$\beta_i(t)$ is the probability that the age of the last request $A_i(t)$ does not exceed maximum sojourn time T, assuming T is constant for any i. We have Eq. (4) for the Possion process,

$$\beta_i(t) = \mathbb{P}\{A_i(t) \leq T\} = F_{A_i}(T) = 1 - e^{-\lambda_i T}. \tag{4}$$

Combining Eq. (3) and (4), we can obtain T, and derive the cache hit ratio h in Eq. (5),

$$h = \sum p_i h_i = \sum p_i \mathbb{P}\{\tau_i < T\} = p_i (1 - e^{-\lambda_i T}). \tag{5}$$

Due to limited space, we omit the detailed derivations of the cache hit ratio for the five caching schemes but summarize the final results and the key affecting factors in Table I. For a given traffic model, fewer caching objects and smaller cache occupation size lead to longer sojourn time and higher cache hit ratio qualitatively. According to the analysis, PBC has the best performance when FCMR is low enough. By contrast, exact caching is overwhelmed by the large number of names; the performance of atomic, on-the-fly, and leaf-only caching schemes heavily depends on the traffic model. In the extreme case that all of the lookup requests only match the leaf prefixes, their variants of Eq. (3) degenerate to $C = (\lvert\mathcal{P}\rvert - \sum_{i \in \mathcal{P}} e^{-\lambda_i T})$, meaning their cache hit ratios are approximately equal to PBC's, which is confirmed by an experiment in Section V.

IV. System Design

We first describe the system framework of the PBC-enabled name lookup engine. Then we describe a software-based implementation and detail its lookup and update processes.

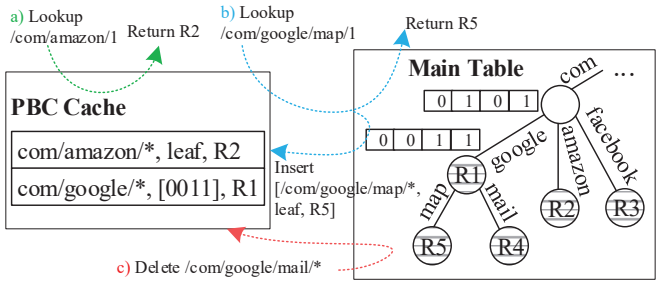


Fig. 3: System framework and example of the three workflows: a) The lookup for name */com/amazon/1* hits a leaf prefix in cache, and returns the corresponding rule R2; b) The lookup for name */com/google/map/1* misses the cache because it hits a '1' in the bitmap, continues to search in the main table, and inserts a new cache entry [*com/google/map/**, leaf, R5]; c) An update deletes a prefix */com/google/mail/** from the main table, which needs to search the cache and delete it if it exists.

A. Framework

Fig. 3 shows the system framework of the PBC-enabled name lookup engine, which contains a PBC cache and a main table. This general lookup framework applies to all caching schemes discussed in Section II. The specific implementations of the cache and the main table can be in software or hardware.

There are three types of workflows on the system framework. First, the lookup results in a cache hit and returns the cached rule without consulting the main table. Second, the lookup results in a cache miss, either true or false, and goes to search in the main table. In this case, if a better match is found in the main table, it is also admitted to the cache. If the cache is full, LRU is used for cache replacement. Third, the rule update may need to update the cache (i.e., modify or delete some entries) for cache consistency. In summary, the PBC cache has one source of lookups for packet forwarding, and two sources of updates for cache replacement and consistency maintenance.

Most name lookup algorithms, either hardware-based or software-based, usually encode the original human-readable segments or names to fixed size strings for storage efficiency and lookup performance. The conversion can be done at the forwarding nodes such as CCNx [9] and NFD [10], or by the senders such as hICN [26]. In the system implementation, the encoded names are used, but we still explain the algorithm with the human-readable names for convenience.

B. Software-based Implementation

The cache indeed stores some of the name prefixes, so it can be implemented like a small main table. Although TCAM-based implementations can realize constant prefix matching, we focus on a software-based implementation as it is more flexible and widely used in the current NDN deployment.

Since hash-based name lookup is faster than trie-based ones in general, we use a hash-based PBC cache as our name cache data structure, as shown in Fig. 4. The main table data structure is a pbTrie which maintains a bitmap for each intermediate node, as shown in Fig. 5.

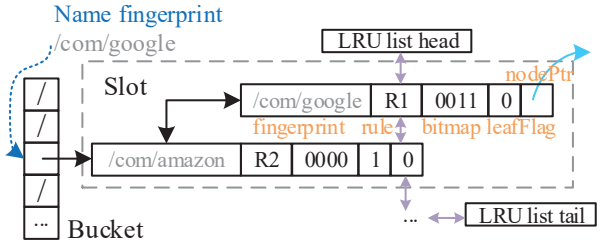


Fig. 4: Structure of the Hash-based PBC

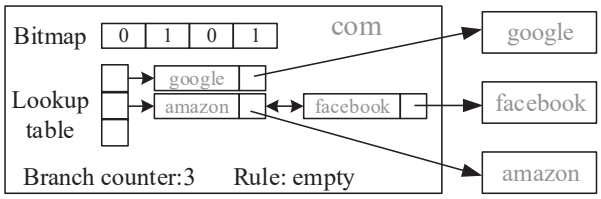


Fig. 5: Structure of the pbTrie node

Each name is encoded into two ordered lists. The first list is called $fpSeg$, which contains the fingerprints of its name segments. A fingerprint is a fixed-size hash value.The second list is called $fpPrf$, which contains the fingerprints of all the prefixes of the name. For example, for the name *com/google/map*, its $fpSeg$ is $[H_1(\text{“}com\text{”}), H_1(\text{“}google\text{”}), H_1(\text{“}map\text{”})]$, and its $fpPrf$ is $[H_2(\text{“}com\text{”}), H_2(\text{“}com\text{”}, \text{“}google\text{”}), H_2(\text{“}com\text{”}, \text{“}google\text{”}, \text{“}map\text{”})]$, in which H_1 and H_2 are two hash functions. Assuming a small number of fingerprint collisions are tolerable for our system, we let $H_2(S_1, S_2, ..., S_n) = H_1(S_1) \oplus H_1(S_2) \oplus ... \oplus H_1(S_n)$ for simplicity, so $fpPrf$ can be derived from $fpSeg$. For systems that cannot tolerate any collision, the hash-table structure in [27] can be used.

1) Hash-based PBC cache: It is essentially a linked hash table as shown in Fig. 4. The key to the hash table is a prefix fingerprint, and the value is a pointer to a cache entry. A cache entry contains five fields including a prefix fingerprint (for match verification), a rule, a bitmap, a leaf flag, and a pointer to the corresponding node in the pbTrie.

All the cache entries are doubly linked so it is easy to implement the LRU cache replacement policy. Each hit entry is relinked to the head. A new cache entry is always inserted to the head, and an old cache entry is removed from the tail if necessary. The cache entries hashed to a same slot are also doubly linked by a slot list.

To search a name in the cache, we linearly search the hash table by using the $fpPrf$ as key in descending prefix length order. At each hashed slot, we linearly search the linked cache entries by comparing the fingerprints. We repeat the above steps until a matching prefix is found or all the prefix lengths are checked. The pseudo code for the PBC-enabled name lookup is shown in Algorithm 1. Once get a matching entry at line 3, we will proceed to the next operations as follows. If a leaf prefix is matched (i.e., the leaf flag is set) or the name's length equals to the prefix length, we can directly return the result as the best match. Otherwise, for this found entry, we need to check the bitmap to see if it is possible to have a longer match in the main table (line 8 to 13). If the answer is

Algorithm 1: PBC-enabled Name lookup

Input: structure for the lookup name n
Output: LPM rule r, pointer to a pbTrie node np, component number that is matched l

```
for l ← n.length to 1 do
    fe = HashTable.lookup(n.fpPrf[l]);
    if fe then
        if l == n.length || fe.leafFlag then
            /* find a LPM or a leaf prefix */
            LruList.reoder(fe);
            return r ← fe.rule, np ← Null, l
        else
            /* To compare the bit value */
            bitIdx = n.fpCmp[l + 1] % Bits(fe.bitmap);
            if fe.bitmap[bitIdx] == 0 then
                LruList.insertHead(fp);
                return r ← fe.rule, np ← Null, l
            else
                return r ← False, np ← fe.nodePtr, l
return r ← False, np ← Null, l ← 0
```

Algorithm 2: Prefix insertion for PBC

Input: cache entry of the insertion perfix e.

```
index ← e.fingprint % BUCKET_LENGTH;
HashTable[index].insert(e);
LruList.insertHead(e);
if entryNum = CACHE_SIZE then
    LruList.deleteTail();
else
    entryNum ← entryNum + 1;
```

negative, the current matching prefix is returned as the best; If the answer is positive, we need to follow the pointer in the entry to continue the search in the pbTrie. The pointer allows us to start the search from the current matching prefix, called the mid-way search, which improves the lookup performance on pbTrie. Besides, whenever a cache hit happens, the cache entry for the matching prefix is relinked to the head of the entry list and the head of the slot list.

A new prefix insertion happens only when the matching prefix of a lookup name does not exist in the cache. To insert a new cache entry, we first check if the cache is full. If so, we evict a cache entry from the tail of the entry list. Then we insert the new entry to the head of the entry list. At last, we find the target hash table slot for this entry and link it to the head of the slot list. The pseudo code for the prefix insertion process is shown in Algorithm 2.

A name table update may involve modifying or deleting a cache entry for a prefix. To modify a cache entry, we only need to locate it in the cache by performing an exact match lookup and rewrite some fields. To delete a cache entry for a prefix, we first need to locate it in the cache. If it is found, we simply remove it from the cache entry list and the slot list. The pseudo code for the update process is shown in Algorithm 3.

2) pbTrie: The trie is built on name segments. To support PBC, pbTrie associates a bitmap to each non-leaf node. As

Algorithm 3: Prefix Update for PBC

Input: cache entry of the updated prefix e, update type $type$
Output: Whether there is an influenced cache entry

```
fe = HashTable.lookup(e.fingprint);
if fe then
    if type == MODIFICATION then
        fe.rule ← e.rule ;                    // Insertion
        fe.bitmap ← e.bitmap;
        fe.leafFlag ← e.leafFlag;
    else
        HashTable.delete(fe) ;                // Deletion
        LruList.delete(fe);
        entryNum ← entryNum − 1
    return True;
return False;
```

shown in Fig. 5, each pbTrie node maintains a bitmap, a node lookup table, and some other auxiliary data such as the branch counter. The branch counter is used to determine the node type and the bitmap size. The lookup table uses the accumulated number of '1's in the bitmap as index. All the branches sharing the same index are linked to the corresponding table entry, e.g., *amazon* and *facebook* in Fig. 5. Therefore, to follow a branch, we need to first hash the segment fingerprint to acquire its bit position in the bitmap; By counting the number of '1's in the bitmap before this bit position, we get the node lookup table index; then we can follow the link list to find the pointer to the next child node. In pbTrie, we set the node lookup table's slot number to be the same as the upper limit of the number of '1's for the current bitmap size and FPR threshold configuration. Therefore, it avoids too many empty slots, and keeps the link lists short. This data structure effectively balances the storage size and the lookup performance. Particularly, to accelerate counting the number of '1's, pbTrie splits a long bitmap to several 64-bit segments. Each segment is associated with a *before* field to record the total number of '1' before this segment. In each segment, we adopt the hardware instruction *POPCNT* [28] to count the number of '1's for any offset in a single CPU cycle. By summing the two numbers, we can get the slot index quickly.

Searching in a pbTrie for LPM is straightforward. Now we describe the pbTrie update process and explain how it can affect the cache using examples shown in Fig. 6.

To insert a prefix to a pbTrie, we come across two cases: 1) The prefix matches an existing trie node. If the trie node is also a prefix node, we need to modify it. We also need to search the cache for the prefix and modify it if it is in cache. Otherwise, if the trie node is not a prefix node, we just update it but we do not need to search the cache for it (see case a in Fig. 6). 2) The inserted prefix needs to create new trie nodes. In this case, we do not need to check the cache for this prefix since it is impossible to have been cached. However, if the new prefix causes its nearest sub-prefix node to add a new branch and change the bitmap, we need to check the cache for that sub-prefix (see case b in Fig. 6).

To delete a prefix in a pbTrie, we come across two cases as

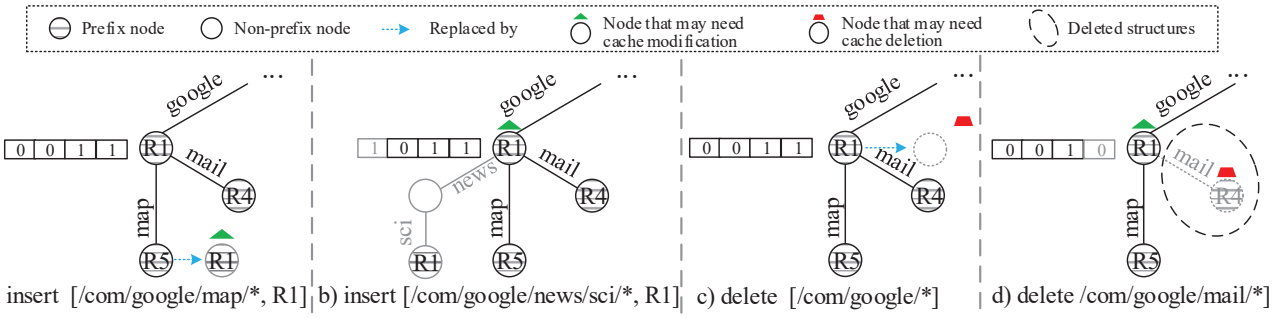


Fig. 6: pbTrie insertion and deletion examples: a) the inserted prefix matches a prefix node; b) the inserted prefix generates new prefix nodes and changes the bitmap of its sub-prefix node; c) the deleted prefix matches a non-leaf node; d) the deleted prefix deletes the leaf node and changes the bitmap of its sub-prefix node.

well: 1) The deleted prefix node is a non-leaf node. We simply clean the rule in this node (see case c in Fig. 6). As the prefix may be cached, we should delete it from the cache if so. 2) The deleted prefix is a leaf node. In this case we need to delete the leaf node first and then recursively delete the parent node until the parent node is a prefix node or it has other child nodes. This operation may also affect some prefix's bitmap. So we need to search the cache for both the original prefix and the affected prefix, and then delete the original one and update the affected one, if any exists (see case d in Fig. 6).

From the above discussion, we can see for any update on pbTrie, the PBC cache needs to do prefix updates from zero to two times to maintain the cache consistency.

V. Performance Evaluation

We conduct experiments on real and synthetic data sets to get the cache hit ratio of different schemes, and the lookup and update performance of systems applying these schemes.

A. Experiment Setup and Data Sets

1) Name Prefix tables: We generate a real name prefix table from the Internet domain name in three steps. First, we obtain 3 million names from DMOZ [21]. Second, we transform them into NDN-style names by reversing the segment order. Finally, we assign a random rule for each name. In addition, we synthesize a name table VT16 from [4], which contains 3.55 million prefixes with each having 3$\sim$16 segments. Statistically, the average segment number is 2.25 for DMOZ prefixes, and 9.50 for VT16 prefixes. The two tables are representative and comprehensive for scheme evaluation and comparison.

2) Name Trace: A name lookup trace is a sequence of names fed into the lookup engine. Each name in the trace is composed of a name prefix from a prefix table and a suffix. To generate a trace, we first randomly select 100K prefixes from a prefix table. Then, we pick up a prefix and append it with different suffixes to generate names according to the Zipf distribution ($\alpha = 0.9$). At last, we shuffle the names and finally get a trace for tests. Since the performance of some caching schemes relies on the prefix type, we ensure each trace has a different ratio of intermediate prefix to leaf prefix. Similarly, we also generate an update trace, and each prefix in the trace is an insertion or a deletion.

3) Platform: We run the experiments on a workstation with an Intel CPU Core i7-6700 (4x 3.4GHz cores) and 32GB DDR4 (2.4GHz) memory. Each CPU core has an 8MB three-level cache. The workstation OS is Ubuntu-16.04-LTS.

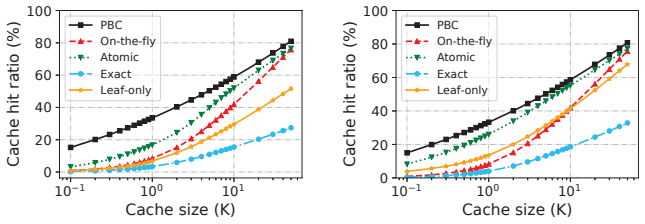
B. Cache Hit Ratio

On evaluation of the cache hit ratio, we compare PBC against the other caching schemes: on-the fly, atomic, exact, and leaf-only. We vary the cache size, intermediate prefix ratio, and suffix set size individually to test the schemes on the prefix tables VT16 and DMOZ. For PBC, we also evaluate the influence of FPR on the cache and system performance.

1) Cache Size: We test how the caching schemes perform under different cache sizes. The results are illustrated in Fig. 7. For both prefix tables and all the cache sizes, PBC performs the best. When the cache size is small, PBC's advantage is more noticeable. For example, on VT16, the cache hit ratio of PBC is 16.8% and 6.8% higher than that of the second best, atomic caching, when the cache size is 1K (account for 1% of active prefixes) and 10K (account for 10% of active prefixes), respectively. When the cache size increases, every scheme performs better. If we keep increasing the cache size, at some point some other caching schemes will outperform PBC due to the false cache misses of PBC. However, in reality, cache is a costly resource and its size is always limited.

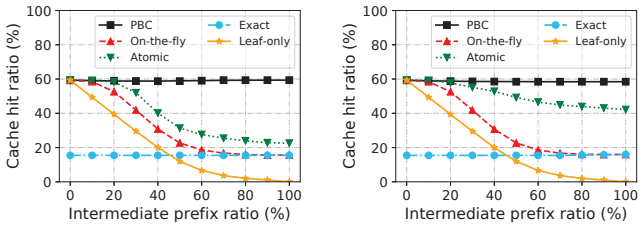
2) Intermediate Prefix Ratio: The atomic and on-the-fly caching schemes are inefficient for caching intermediate prefixes. The leaf-only caching scheme does not cache these prefixes. So the intermediate prefix ratio in a trace may significantly affect the cache performance. The results are illustrated in Fig. 8. The PBC and exact caching schemes show a flat cache hit ratio, which means they are insensitive to the prefix type, because they both only cache one prefix per cache replacement operation. By contrast, the atomic, on-the-fly and leaf-only caching schemes see a decreasing trend on their cache hit ratio as the intermediate prefix ratio increases. Note that, except for the exact caching scheme, the other caching schemes have the same hit ratio when all the hit prefixes are leaves, which complies with our theoretical analysis.

3) Suffix Set Size: We test the impact of suffix set size on the cache hit ratio and show the results in Fig. 9. PBC remains stable and efficient. The atomic and leaf-only caching



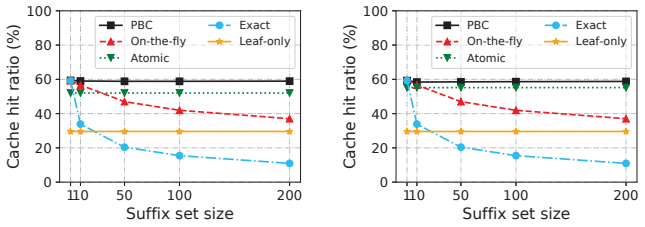
(a) VT16 (b) DMOZ

Fig. 7: Cache hit ratios with the increasing cache size. The intermediate prefix ratio is 20% and the suffix set size is 100.



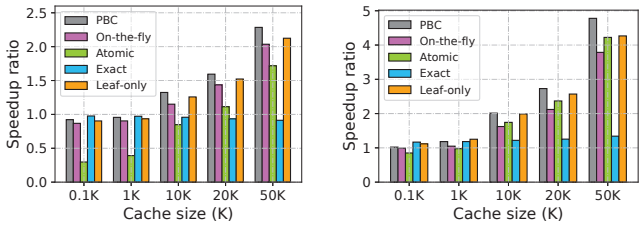
(a) VT16 (b) DMOZ

Fig. 8: Cache hit ratios with the increasing intermediate prefix ratio. The cache size is 10K and the suffix set size is 100.



(a) VT16 (b) DMOZ

Fig. 9: Cache hit ratios with different suffix set sizes. The intermediate prefix ratio is 20% and the cache size is 10K.



(a) VT16 (b) DMOZ

Fig. 10: Speedup ratios for lookup with a single thread. The intermediate prefix ratio is 20% and the suffix set size is 100.

TABLE III: FCMR and average bitmap length (in bits) on VT16 and DMOZ with different θ. The cache size is 10K and the intermediate prefix ratio is 0.3.

θ	Branch # of the 1st range	FCMR		AVG bitmap length	
		VT16	DMOZ	VT16	DMOZ
12.5%	1-8	3.32%	2.70%	64.47	89.64
6.25%	1-4	2.60%	1.89%	65.72	109.81
3.13%	1-2	1.56%	1.30%	79.43	146.46
1.56%	1	0.67%	0.83%	75.67	208.87

schemes are also stable, because they only cache prefixes, but their cache hit ratios are lower than PBC. The on-the-fly and exact caching schemes are influenced by the suffix set size significantly. The exact caching scheme is especially suffered, because it only caches exact names including the suffix.

4) FPR: The cache hit ratio of PBC is related to the FCMR. We set the threshold FPR θ for the adaptive bitmap to influence the FCMR. We show that the FCMR and the average bitmap length under different θ values in Table III. Even when the threshold is high, the FCMR is still low, thanks to the high leaf node hit ratio (70%). The average bitmap length is small and increases slowly when θ decreases, reflecting the efficiency of the adaptive bitmap.

C. Lookup Performance

1) Single Thread: We run the name lookup engine based on different caching schemes on a single thread. The lookup performance on different cache sizes is shown in Fig. 10. The y-ordinate represents the speedup due to the use of a caching scheme on top of a pure pbTrie. For very small cache sizes, caching fails to show any benefit and the atomic caching scheme has even negative impact, because the cache hit ratio is low and most of cache lookups are in vein. When the cache

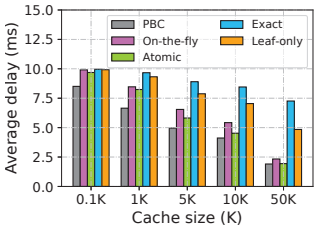


Fig. 11: Average delay in the distributed system. $\lambda = 1000$ and $RTT = 10ms$.

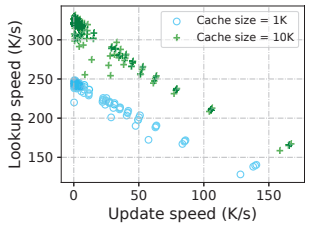


Fig. 12: The single-thread lookup and update performance for PBC on VT16.

size is relatively large, all schemes perform better. PBC shows the best performance, which can improve the speed by up to 4.78 times on DMOZ and 2.28 times on VT16. Atomic caching remains the worst and the system performs better without it. Besides, the caching schemes have a better effect on DMOZ as its pure lookup speed is slower. Due to space, we omit the results for other intermediate prefix ratio and suffix set size configurations. The results show similar trends as in Fig. 8 and Fig. 9.

2) Distributed System: To highlight the PBC performance in distributed system where the cache runs in edge nodes and the pbTrie runs in a remote controller, we simulate a simple scenario. We set one edge node to receive name lookup requests, and the transmission delay between the edge node and the controller is 5ms. The lookup requests obey a Poisson distribution whose λ is 1K, i.e., the average request frequency is 1K per second. We evaluate the system delay which is defined as the time used to get a name lookup result from either local cache or remote controller. As shown in Fig. 11, PBC still performs the best due to its high cache hit ratio, reducing up to 80% delays, and is closely followed by the on-the-fly and atomic caching schemes.

D. Update Performance

1) Cache Replacement Performance: The atomic caching scheme needs considerable operations on cache replacement, especially under a deep name prefix table, small-sized cache and high intermediate prefix ratio. E.g., for VT16, the average operations are 19.7 per lookup, if the cache size is 1K and the intermediate prefix ratio is 0.5. The other schemes all admit a new prefix/name when a cache miss occurs. If the cache is full, a prefix/name eviction is also needed. Hence, their expected cache replacement operations are approximately equal to $2(1-h)$. PBC needs the fewest operations on cache replacement due to its highest cache hit ratio.

2) Cache Consistence Performance: When a table update arrives, the trie needs to be updated and the cache needs to be checked for consistency. The trie update can be time-consuming, so we examine its performance first. The update performance of pbTrie is closely related to the table structure. It has lower update performance on the flat table such as DMOZ. For example, the prefix insertion speed is 288K/s on VT16 but only 3 K/s on DMOZ. This is because the complexity of bitmap reconstruction increases rapidly with the number of branches. The prefix deletion is much faster on VT16, up to 514 K/s, but only about 1 K/s on DMOZ as many deletions involve nodes with large bitmaps.

3) Comprehensive Performance: An update can interrupt the lookup processes. We examine the comprehensive system performance of PBC on VT16 by mixing the lookups and updates, which can reflect the true behavior when deploying it in a distributed system. Fig. 12 shows the updates and lookups are negatively correlated on a single thread. A larger cache size boosts the system performance due to the large cache hit ratio. In a distributed system, if we parallelize the cache lookup and the main table update, the lookup interruption time can be reduced. However, to ensure cache consistency, in this case we need to prohibit the mid-way search on the pbTrie. To simulate such a system, we assign the cache and the main table in two threads with parallel lookups and updates. When a cache miss or table update occurs, the two threads communicate with each other. Different from the above setting on a single thread, we control the ratio of lookups and updates. If the ratio is low, i.e., the main table is busy with updates and has not enough time for cache miss, the lookup delay will increase. If the ratio is high, the main table may be idle on waiting for cache misses. The results show that if there are 5 table lookups per update on average, the system on VT16 can achieve up to 41 K/s update speed, which is high enough referred to the IP prefix updates in CAIDA [29], with almost uninterrupted 2.3 M/s lookups.

VI. Conclusion

PBC deploys an adaptive bitmap for name cache to facilitate efficient LPM. Theoretical analysis and experiments show that PBC achieves the highest hit ratio among existing name caching schemes. In the future, we plan to develop PBC to other networks, especially the SDN-enabled IP networks.

Acknowledgement

This work is supported by Guangdong Basic and Applied Basic Research Foundation (2019B1515120031) and NSFC (68172213,61432009). The corresponding author is Bin Liu (lmyujie@gmail.com).

References

[1] L. Zhang, A. Afanasyev, J. Burke *et al.*, "Named data networking," *ACM SIGCOMM CCR*, vol. 44, no. 3, pp. 66–73, 2014.

[2] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: Fast and dos-resistant forwarding with hash tables," in *Proceeding of ACM/IEEE ANCS*, 2013, pp. 215–226.

[3] L. Zhang, D. Estrin *et al.*, "Named data networking (ndn) project," 2010.

[4] K. Huang and Z. Wang, "A hybrid approach to scalable name prefix lookup," in *Proceeding of IEEE IWQoS*, 2018.

[5] Y. Wang, Y. Zu, T. Zhang *et al.*, "Wire speed name lookup: A gpu-based approach." in *Proceeding of USENIX NSDI*, 2013, pp. 199–212.

[6] Y. Wang, H. Dai *et al.*, "Gpu-accelerated name lookup with component encoding," *Computer Networks*, vol. 57, no. 16, pp. 3165–3177, 2013.

[7] Y. Wang *et al.*, "Scalable name lookup in ndn using effective name component encoding," in *Proceeding of ICDCS*, 2012, pp. 688–697.

[8] T. Song *et al.*, "Scalable name-based packet forwarding: From millions to billions," in *Proceedings of ACM ICN*, 2015, pp. 19–28.

[9] Ccnx. [Online]. Available: http://blogs.parc.com/ccnx/.

[10] Nfd. [Online]. Available: http://nameddata.net/doc/NFD

[11] Y. Wang, B. Xu, D. Tai *et al.*, "Fast name lookup for named data networking," in *Proceeding of IEEE IWQoS*, 2014, pp. 198–207.

[12] H. Yuan and P. Crowley, "Reliably scalable name prefix lookup," in *Proceedings of ACM/IEEE ANCS*, 2015, pp. 111–121.

[13] H. Dai *et al.*, "Bfast: Unified and scalable index for ndn forwarding architecture." in *Proceeding of IEEE INFOCOM*, 2015, pp. 2290–2298.

[14] L. Breslau, P. Cao, L. Fan, *et al.*, "Web caching and zipf-like distributions: Evidence and implications," in *Proceeding of IEEE INFOCOM*, vol. 1, 1999, pp. 126–134.

[15] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: a view from the edge," in *Proceedings of ACM SIGCOMM conference on Internet measurement*, 2007, pp. 15–28.

[16] X. Chen *et al.*, "Investigating route cache in named data networking," *IEEE Communications Letters*, vol. 22, no. 2, pp. 296–299, 2018.

[17] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang, "A fast and memory-efficient trie structure for name-based packet forwarding," in *Proceeding of IEEE ICNP*, 2018, pp. 302–312.

[18] H. Liu, "Routing prefix caching in network processor design." in *Proceeding of ICCCN*, 2001, pp. 18–23.

[19] Y. Liu *et al.*, "Efficient fib caching using minimal non-overlapping prefixes," *Computer Networks*, vol. 43, no. 1, pp. 14–21, 2013.

[20] L. Fan *et al.*, "Summary cache: a scalable wide-area web cache sharing protocol," *IEEE/ACM TON*, vol. 8, no. 3, pp. 281–293, 2000.

[21] Dmoz-open directory project. [Online]. Available: http://www.dmoz.org/

[22] H. Che, Y. Tung, and Z. Wang, "Hierarchical web caching systems: Modeling, design and experimental results," *IEEE JSAC*, vol. 20, no. 7, pp. 1305–1314, 2002.

[23] A. Dan and D. Towsley, *An approximate analysis of the LRU and FIFO buffer replacement schemes*. ACM, 1990, vol. 18, no. 1.

[24] C. Fricker, P. Robert, and J. Roberts, "A versatile and accurate approximation for lru cache performance," in *Proceedings of Teletraffic Congress,*, 2012, pp. 1–8.

[25] M. Garetto, E. Leonardi, and V. Martina, "A unified approach to the performance analysis of caching systems," *ACM ToMPECS*, vol. 1, no. 3, p. 12, 2016.

[26] M. Sardara, L. Muscariello, and A. Compagno, "A transport layer and socket api for (h) icn: design, implementation and performance analysis," in *Proceedings of ACM ICN*, 2018, pp. 137–147.

[27] H. Yuan, P. Crowley, and T. Song, "Enhancing scalable name-based forwarding," in *Proceedings of ACM/IEEE ANCS*, 2017, pp. 60–69.

[28] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software ip routing table lookup," *ACM SIGCOMM Computer Communication Review*, vol. 45, no. 4, pp. 57–70, 2015.

[29] C. Zhang, Y. Feng, H. Song *et al.*, "Obma: Minimizing bitmap data structure with fast and uninterrupted update processing," in *Proceeding of IEEE IWQoS*, 2018, pp. 1–6.