A Hybrid Approach to Scalable Name Prefix Lookup

Kun Huang† and Zhaohua Wang†‡
†Institute of Computing Technology, Chinese Academy of Sciences and ‡University of Chinese Academy of Sciences
Beijing, China
{huangkun09, wangzhaohua}@ict.ac.cn

***Abstract*—Name prefix lookup is a core function in Named Data Networking (NDN). It is challenging to perform high-speed name-based longest prefix match lookups against a large amount of variable-length, hierarchical name prefixes in NDN. However, prior work concentrates on software-based name prefix lookup, and can't satisfy the scalability demands of high-speed lookups, low memory cost, and fast incremental updates. In this paper, we propose a hybrid approach to scalable name prefix lookup with hardware and software. We propose SACS, a shape and content search framework with ternary content addressable memories (TCAMs) and static random memory access memories (SRAMs). SACS aims to achieve high-speed lookups and low memory cost, while sustaining fast incremental updates. In SACS, a TCAM-based shape search module is first used to determine a subset of possible matching prefixes, and then a SRM-based content search module is used on the subset to find the longest matching prefix. For SACS, we propose a first shrinking least load algorithm to pack large amounts of shapes of name prefixes in a small TCAM. A shape of a name prefix is a sequence of its component lengths. We also propose a dual fingerprint-based hash table to improve the content search performance in SRAMs. Experimental results demonstrate that SACS outperforms state-of-the-art schemes by achieving up to 2.4X higher lookup throughput, up to 53% lower memory cost, and up to 96% higher insert throughput.**

Keywords—information-centric networks, named data networks, name prefix lookup, ternary content addressable memories

I. INTRODUCTION

Named Data Networking (NDN) [1] is one of new future Internet architectures, which changes the semantics of network services from host-based to content-based delivery. NDN uses a content-centric approach to packet routing and forwarding by using content names instead of host addresses [1, 2, 3]. This approach brings many benefits including content distribution, mobility support, data-centric security, etc. Meanwhile, it also introduces a great challenge of high-speed name-based packet forwarding in NDN.

Name prefix lookup is a core function in an NDN router [4, 5], which makes a forwarding decision for every packet. For the arrival of a packet, a NDN router carries out name prefix lookup by using the content name carried in the packet header as key to perform a name-based longest prefix match (LPM) lookup on a large amount of variable-length, hierarchical name prefixes. For instance, a content name */a/b/c/d* consists of four components delimited by '/'. We assume that it matches three name prefixes */a/**, */a/b/**, and */a/b/c/**, where '*' is a wildcard. So the longest prefix */a/b/c/** is returned as the result by LPM-based name prefix lookup.

Name prefix lookup poses a scalability challenge and needs to meet three key demands of high-speed lookups, low memory costs, and fast incremental updates. First, unlike fixed-length IP addresses, NDN names have a hierarchical structure with an unbounded number of variable-length name components. This feature makes previous approaches to high-speed IP lookup not work well for name prefix lookup in NDN. Second, the number of prefixes in NDN Forwarding Information Bases (FIBs) is expected to be one to two orders of magnitude larger than IP FIBs. So it is challenging to design fast and memory-efficient data structures for name prefix lookup in NDN FIBs. Third, NDN forwarding tables need to support fast dynamic updates. For an Interest or Data packet, the packet forwarding process in NDN usually involves the insertions and/or deletions on the Pending Interest Tables (PITs) or Content Stores (CSs). This requires fast incremental updates on the FIBs, PITs, and CSs in NDN, while achieving high-speed name prefix lookups.

However, prior work concentrates on software-based name prefix lookup, but can't meet these above demands. Trie-based schemes [6, 7, 8, 9, 10] use a trie data structure to implement memory-efficient name prefix lookup. But they achieve slow updates and lookups with O(w) memory accesses for a search on name prefixes with up to w components. Hash-based linear search (LS) schemes (i.e., CCNx [11] and NFD [12]) perform a name-based LPM search on hash tables organized by the prefix lengths from the longest to the shortest. LS has fast updates and low memory cost, but slow lookups because it requires worst-case O(w) hash lookups per search. Furthermore, hash-based binary search (BS) schemes [4, 13, 14, 15] improve the lookup throughput by cutting down the number of hash lookups per search to O(log(w)). They have slow updates and high memory cost due to adding additional markers, and can't sustain high-speed lookups on larger amounts of name prefixes with longer lengths. Therefore, existing schemes can't achieve scalable and high-speed name prefix lookups when scaling to large amounts of variable-length name prefixes.

In this paper, we propose a hybrid approach to scalable name prefix lookup with hardware and software. Our goal is to achieve high-speed name prefix lookups and low memory cost, while sustaining fast incremental updates. To achieve this goal, we leverage high-speed ternary content addressable memories (TCAMs) that are widely used in Internet routers, and combine with static random memory access memories (SRAMs). We propose SACS, a Shape And Content Search framework with TCAMs and SRAMs to accelerate name prefix lookups. In SACS, a TCAM-based shape search module is first used to determine a subset of possible matching prefixes, and then a

This work was supported in part by the National Science and Technology Major Project of China under Grant No. 2017ZX03001013-002 and the Huawei Technical Cooperation Program under Grant No. YB2015110041.

978-1-5386-2542-2/18/$31.00 ©2018 IEEE

SRAM-based content search module is used on the subset to find the longest matching prefix. A shape of a name prefix is a sequence of the lengths of its components. The key to SACS is encoding the shapes rather than the contents of name prefixes in a small TCAM. This is because name prefixes have a large number of variable-length components, so their contents can't be stored in a small TCAM with limited widths. We use a hash table in SRAMs to store a subset of name prefixes with the same shape, allowing for fast incremental updates. One TCAM entry encodes a shape that points to a SRAM-based hash table containing a subset of name prefixes. Therefore, SACS requires O(1) hash lookup per search in average case, achieving higher lookup performance than previous schemes.

In addition, we propose two novel techniques to improve the performance of SACS. First, we propose a first shrinking least load algorithm to pack large amounts of shapes in a small TCAM by shrinking longer least-load shapes into shorter ones. Second, we propose a dual fingerprint-based hash table (DFHT) to improve the performance of SRAM-based content search by using content and shape fingerprints in a slot of a hash table.

We conducted simulation experiments on real and synthetic datasets to evaluate SACS and compare with previous schemes including NFD [12], BS [14], and BFAST [23]. Experimental results demonstrate that SACS outperforms previous schemes in lookup throughput. Compared to state-of-the-art BS, SACS achieves up to 2.4X higher lookup throughput, up to 53% lower memory cost, and up to 96% higher insert throughput as well as up to 133% higher delete throughput.

This paper makes three key contributions as follows:

- We propose a Shape And Content Search framework (SACS) for high-speed name prefix lookups. SACS is a hybrid approach with TCAMs and SRAMs, where a TCAM-based shape search module is first used to determine a subset of possible matching prefixes, and then a SRAM-based content search module is used on the subset to find the longest matching prefix.
- We propose two techniques to improve the performance of SACS. First, we propose a first shrinking least load algorithm to pack large amounts of shapes in a small TCAM. Second, we propose a dual fingerprint-based hash table to improve the performance of SRAM-based content search.
- We conduct simulation experiments to compare SACS with previous schemes. The results show that SACS outperforms previous schemes in lookup throughput. Compared to state-of-the-art BS, SACS improves up to 2.4X lookup throughput, reduces up to 53% memory cost, and improves up to 96% insert throughput.

The rest of this paper is organized as follows. We introduce the background and related work on NDN name prefix lookup in Section 2. Section 3 describes the framework and algorithms of SACS, and Section 4 provides experimental evaluation of SACS. Finally, Section 5 concludes this paper.

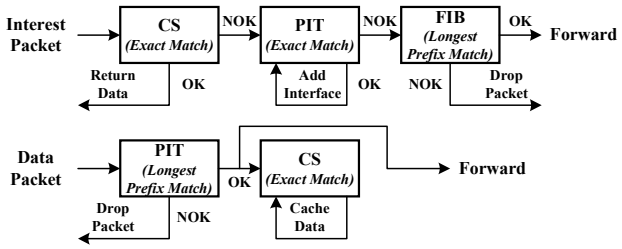


Figure 1. NDN packet forwarding process.

II. Background and Related Work

In this section, we first present the background on NDN packet forwarding, and then review the related work on name prefix lookup proposed in recent years.

A. Background on NDN Packet Forwarding

NDN is a receiver-driven, content-centric communication paradigm, where packet forwarding decisions are made based on content names. NDN uses two types of packets: Interest and Data, both of which carry a content name that identifies a piece of data transmitted in the Data packet. To forward Interest and Data packets, an NDN router maintains three forwarding tables: a Pending Interest Table (PIT), a Content Store (CS), and a Forwarding Information Base (FIB). The PIT stores all the Interests that are not satisfied yet, where each entry records a content name carried in an Interest packet, together with its incoming and outgoing interfaces. The CS is a cache memory that stores a copy of previously processed Data to answer re-requested Interests. The FIB stores a set of name prefix rules each with outgoing interfaces as the next hop.

Fig. 1 illustrates the NDN packet forwarding process. We assume that a data consumer sends an Interest packet with a content name */a/b/c* to an NDN router. The router first searches the CS based on exact match (EM) to find the matching Data whose name starts with */a/b/c* (i.e., */a/b/c* and */a/b/c/1*). If it exists, the Data in the CS is returned to the data consumer. Otherwise, the router proceeds to search the PIT based on EM. If there exists a PIT entry that matches */a/b/c*, the router adds the incoming interface of the Interest packet into the PIT entry. If there is no match in the PIT, the router forwards the Interest packet to the next hop by searching the FIB based on longest prefix match (LPM). There are several matching name prefixes (i.e., */a/**, */a/b/**, and */a/b/c/**) in the FIB, and the router returns the longest prefix (i.e., */a/b/c/**) as the result. If no match is found in the FIB, the Interest packet is dropped.

When a Data packet with a content name */a/b/c/1* arrives, an NDN router first searches the PIT based on LPM to find the matching entry whose name is a prefix of */a/b/c/1* (i.e., */a/b/c/1*, */a/b/c*, */a/b*, and */a*). If it exists, then the router forward the Data packet to all downstream interfaces listed in the PIT entry, and removes the entry from the PIT. Meanwhile, the Data packet is cached in the CS, which involves an EM lookup. If no match is found in the PIT, the Data packet is dropped.

This paper focuses on name-based LPM lookup, because it is the performance bottleneck of NDN packet forwarding. Like

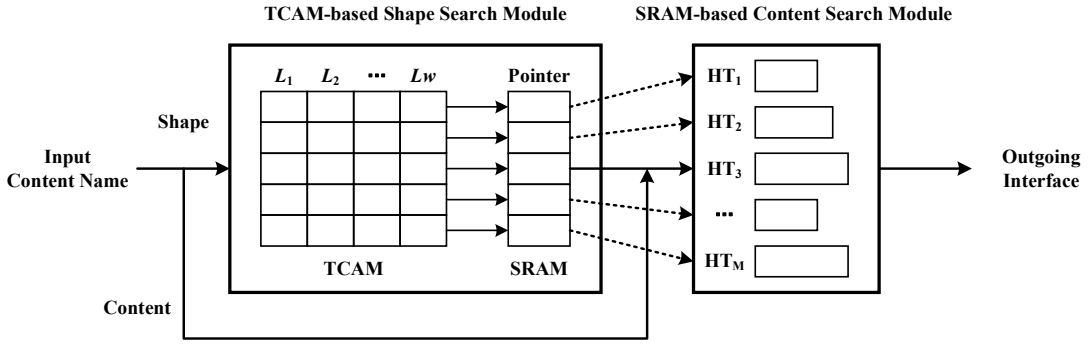


Figure 2. A SACS framework with TCAMs and SRAMs for name prefix lookup.

prior work [7, 16], we consolidate three separate tables of PIT, CS, and FIB into a common table to improve the performance of NDN packet forwarding. The consolidated table reduces the total memory cost required by the three tables, and improves lookup performance by incorporating LPM and EM in a single search. Note that EM is a special case of LPM in this paper.

B. Related Work on Name Prefix Lookup

Name prefix lookup is a core function in NDN, which uses a content name carried in the packet header as key to perform a name-based LPM lookup to find the longest matching prefix. Recently, several software-based name prefix lookup schemes have been proposed to improve the scalability and performance. These schemes are usually classified into three categories: trie-based, hash-based, and Bloom filter-based.

1) Trie-based schemes: Trie-based schemes [6, 7, 8, 9, 10] use a trie data structure to implement name prefix lookup. For instance, NCE [6] builds a memory-efficient character trie by encoding name components. A tokenized Patricia trie [8, 9] is proposed to further improve memory efficiency. Also, a multi-stride character trie [10] is implemented on GPU for parallel name prefix lookup. However, existing trie-based schemes do not achieve high-speed lookups on larger amounts of name prefixes. This is because they require O(w) memory accesses for a search in worst case by traversing the trie data structure, where w is the maximum number of components of prefixes. In addition, they have high overhead for frequent per-packet updates, which impedes normal name prefix lookups.

2) Hash-based schemes: Hash-based schemes [4, 11, 12, 13, 14] use a sequence of hash tables organized by the number of components of name prefixes. For instance, CCNx [11] and NFD [12] use linear search (LS) on these hash tables to find the longest matching prefix from the longest to the shortest, until the first match is found. LS achieves fast incremental updates and low memory cost, but not high-speed lookups, because it requires worst-case O(w) hash lookups per search. Furthermore, binary search (BS) schemes [4, 13, 14, 15] are proposed to perform a LPM lookup in a binary search tree on these hash tables. BS achieves faster lookups over LS because it requires worst-case O(log(w)) hash lookups per search, but has higher memory cost and slower updates. This is because additional markers of shorter lengths that point to prefixes of longer lengths are required by BS to ensure that the search can find a longer matching prefix [14, 15]. Nevertheless, existing hash-based schemes are designed for software so that they can't keep up with line rates when scaling to billions of name prefixes in the future.

3) Bloom filter-based schemes: Bloom filters (BFs) are a space-efficient randomized data structure for fast membership queries with a small false positive probability [16, 17]. Recent efforts have proposed BF-based schemes [5, 18, 19, 20, 21, 22, 23, 24, 25] to accelerate name prefix lookup by adding a BF to avoid unnecessary hash lookups. For instance, PBF [5] uses a cache-line sized BF to identify the length of longest matching prefix. NameFilter [19] uses two-stage BFs to determine both the length of the longest matching prefix and the outgoing interfaces. BFAST [23] uses multiple counting Bloom filters (CBFs) that use a counter to replace a bit of BFs to balance the bucket load in a hash table. However, because BFs yield false positives, these schemes suffer either additional hash lookup overhead or a bandwidth waste of additional traffic.

To summarize, prior work concentrates on software-based solutions, and can't sustain high-speed lookups when scaling to large amounts of name prefixes. In contrast, we propose SACS, a hybrid approach based on hardware and software to achieve high-speed lookups and low memory cost while sustaining fast incremental updates. In fact, SACS uses TCAMs to accelerate name prefix lookup, outperforming previous schemes.

III. Shape And Content Search (SACS)

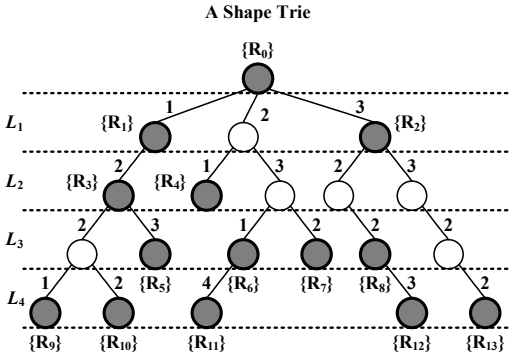
We propose a hybrid approach toward scalable name prefix lookup based on hardware and software. Our goal is to achieve high-speed lookups and low memory cost, while sustaining fast incremental updates. In this section, we first present SACS, a Shape And Content Search framework combining TCAMs and SRAMs. Then, we elaborate the design of TCAM-based shape search and SRAM-based content search in SACS. Finally, we present the incremental update algorithm of SACS.

A. SACS Framework

We propose a SACS framework to accelerate NDN name prefix lookup with off-the-shelf TCAMs that are widely used

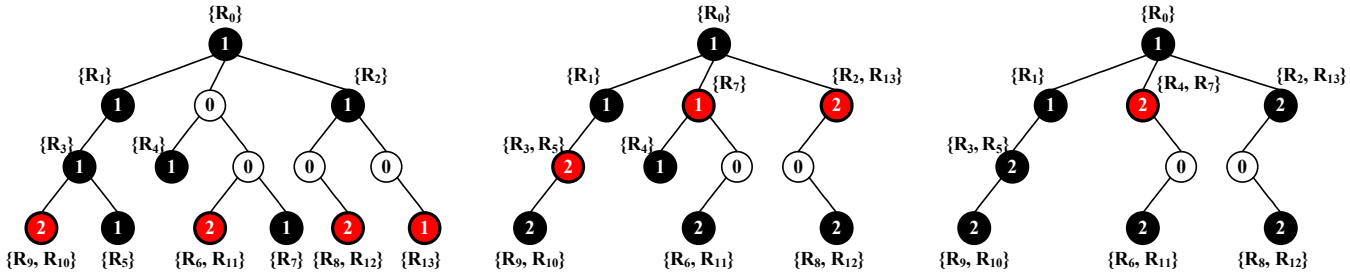
	Name Prefix	Shape	Next Hop
R_0	/*	/*	P_0
R_1	/a/*	/1/*	P_1
R_2	/abc/*	/3/*	P_2
R_3	/a/ab/*	/1/2/*	P_1
R_4	/ab/a/*	/2/1/*	P_3
R_5	/a/ab/abc/*	/1/2/3/*	P_4
R_6	/ab/abc/a/*	/2/3/1/*	P_2
R_7	/ab/abc/ab/*	/2/3/2/*	P_3
R_8	/abc/ab/ab/*	/3/2/2/*	P_4
R_9	/a/ab/ab/a/*	/1/2/2/1/*	P_1
R_{10}	/a/ab/ab/ab/*	/1/2/2/2/*	P_3
R_{11}	/ab/abc/a/abcd/*	/2/3/1/4/*	P_4
R_{12}	/abc/ab/ab/abc/*	/3/2/2/3/*	P_2
R_{13}	/abc/abc/ab/ab/*	/3/3/2/2/*	P_3

(a) A set of name prefix rules and a set of shapes.



(b) A shape trie.

Figure 3. Illustration of a shape trie for an example set of name prefix rules.



(a) Step 1: shrinking pruned leaf nodes.

(b) Step 2: shrinking prefix nodes with load 1.

(c) Step 3: shrinking other prefix nodes.

Figure 4. Illustration of the first shrinking least load algorithm on an example shape trie.

in Internet routers. TCAMs are high-speed associate memories that allow ternary states for each cell in a TCAM entry: '0', '1', or '*' (wildcard). A TCAM performs a parallel search against all occupied TCAM entries within a single memory access and a fixed number of clock cycles, and returns the first matching (highest-priority) entry.

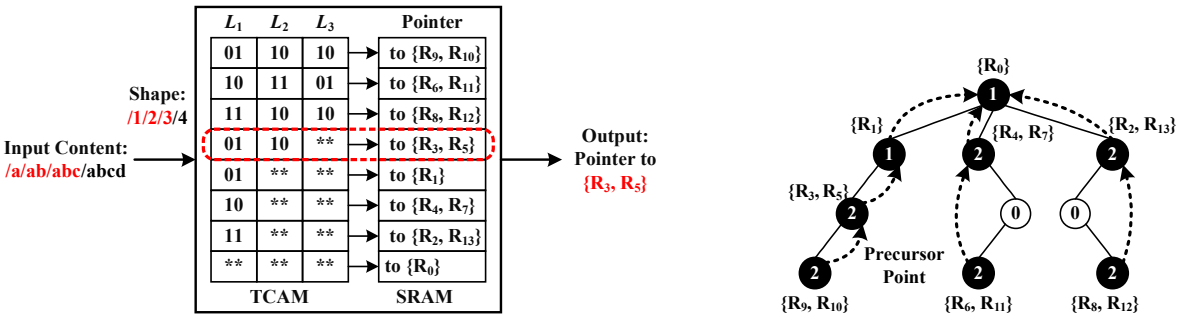
We exploit the above unique features of TCAMs to design our SACS framework. SACS consists of two components: a TCAM-based shape search module and a SRAM-based content search module as shown in Fig. 2. The basic idea behind SACS is that a TCAM-based shape search module is first used as index to determine a subset of possible matching prefixes, and then a SRAM-based content search module is used on the subset to find the longest matching prefix. In practice, we use a TCAM table to implement a shape search, and use a sequence of hash tables in SRAMs to implement a content search. Each TCAM entry points to a hash table that stores a subset of name prefixes with the same shape. A shape of a name prefix is a sequence of its component lengths.

The SACS framework works as follows. For an incoming NDN packet, SACS uses the shape of a content name carried in the packet as key to perform a shape search in a TCAM table, and returns the first matching TCAM entry. Then, SACS uses the content name as key to perform a content search in the hash table pointed by the matching TCAM entry. Therefore, SACS uses high-speed TCAMs to reduce the number of hash lookups in SRAMs, improving name prefix lookup.

B. TCAM-based Shape Search Module

We describe in detail the design of a TCAM-based shape search module. Each entry in a TCAM table contains two parts: a TCAM part and a SRAM part. The TCAM part stores shapes of name prefixes, while the SRAM part stores pointers to a sequence of hash tables, each storing a subset of name prefixes with the same shape. We note that a shape of a name prefix is a sequence $/L_1/L_2/.../L_w/*$ of lengths of its w components, where '/' is a delimited symbol, '*' is a wildcard, and L_i is the length of i-th name component for $1 \le i \le w$. For example, a name prefix */a/ab/** has a shape /1/2/*, where 1 and 2 are the first-level and second-level component lengths. Fig. 3(a) illustrates a set of shapes for an example set of 14 name prefix rules $\{R_0, R_1, \ldots, R_{13}\}$. For these shapes, we build a shape trie with up to w depth (see Fig. 3(b)), where a node denotes a component of a name prefix and an edge denotes the component length. We note that a name shape corresponds to a prefix node (a black node in Fig. 3(b)) in the shape trie, and other nodes are intermediate nodes (white nodes in Fig. 3(b)).

We encode each prefix node of a shape trie in one entry of a TCAM-based shape search module. Shapes are encoded in a TCAM table in descending order of the shape length due to the



(a) Search a content name /a/ab/abc/abcd in a TCAM.

(b) Precursor points in a shape trie.

Figure 5. Illustration of a TCAM-based shape search module in SACS.

first matching feature of TCAMs. However, it is challenging to encode large amounts of shapes in a small TCAM. Today's TCAMs have the limited width (i.e., 36, 72, 144, and 288 bits) and limited number of entries (i.e., 5K to 50K). On the other hand, there are ever-increasing name prefixes (i.e., 1M to 10M), which generates large amounts of unique shapes (i.e., 100K to 1M), beyond the available TCAM space capacity. Therefore, it is critical to pack large amounts of shapes into a small TCAM.

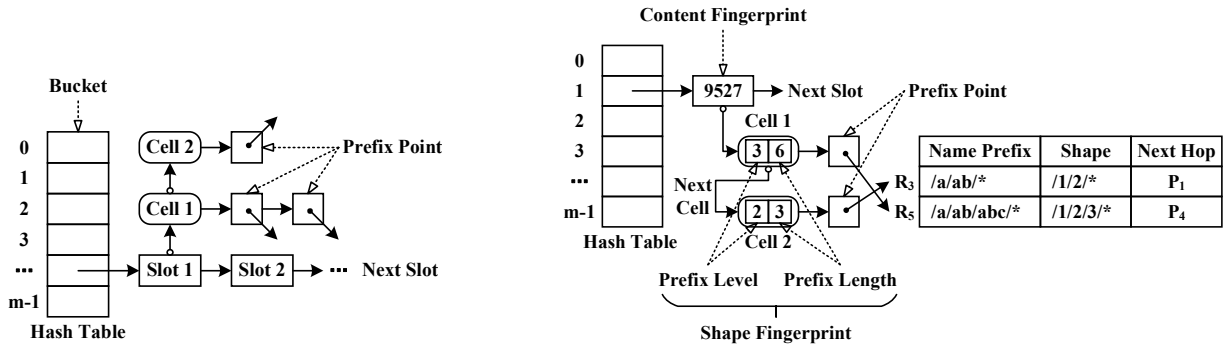
Algorithm 1: Packing a shape trie in a TCAM with FSLL.

```
Function PackShapeTrieInTCAM(shape_trie, tcam_table)
// shrink pruned prefix nodes each with the depth more than the TCAM width.
1:  for (each prefix_node with the depth > tcam_table.width) do
2:      merge prefix_node into its precursor_node;
3:  end for
// shrink longer prefix nodes with the least load into shorter prefix nodes.
4:  while (shape_trie.num_prefix_nodes > tcam_table.num_entries) do
5:      find least_load_prefix_node with the maximum length;
6:      if (least_load_prefix_node's precursor_prefix_node != root) then
7:          merge least_load_prefix_node into its precursor_prefix_node;
8:      else // precursor_prefix_node is the root of shape_trie.
9:          merge least_load_prefix_node into its first_level_precursor_node;
10:     end if
11: end while
// encode each prefix node in a TCAM entry in descending order of the length.
12: for (each prefix_node with the maximum depth in shape_trie) do
13:     encode prefix_node in one entry of tcam_table;
14: end for
```

To overcome the above challenge, we propose a novel bin packing algorithm called First Shrinking Least Load (FSLL). FSLL first shrinks longer prefix nodes with the least load into shorter prefix nodes in a shape trie. The purpose of FSLL is to pack all shapes into a small TCAM table, while balancing the load among hash tables in SRAMs. With FSLL, we alleviate the imbalance of SRAM-based hash tables, and improve worst-case lookup performance. Algorithm 1 shows the algorithm of FSLL. First, we prune leaf prefix nodes each with the depth more than the TCAM width, and shrink them into their own precursor (prefix or non-prefix) nodes in a shape trie (Lines 1 to 3). Second, we iteratively shrink longer prefix nodes with the least load into shorter prefix nodes (or the first-level nodes but not the root for avoiding too heavy load of the root) (Lines 4 to 11). Finally, we build a TCAM table by encoding each prefix node into one TCAM entry in descending order of the node width (Lines 12 to 14). The time complexity of FSLL is O(n), where n is the number of unique shapes for a given set of name prefixes. Our experimental results show that FSLL is fast, and requires up to 0.3 seconds for 1M name prefixes and up to 35 seconds for 10M name prefixes (see Table 5).

Fig. 4 illustrates the FSLL algorithm on an example shape trie in Fig. 3. The number in a prefix node denotes the number of name prefixes stored in the node. We assume that a TCAM table contains up to 8 entries each with 6 bits width. We see that each level in the shape trie requires 2 bits to represent the maximum component length as shown in Fig 3. So the TCAM table contains the first three levels L_1, L_2, and L_3 of the shape trie. At step 1, FSLL prunes all leaf prefix nodes at level 4, and then shrink them into their own precursor nodes at level 3, as shown in Fig. 4(a). For example, two prefix nodes {R_9} and {R_{10}} are shrank up to a new prefix node {R_9, R_{10}} (a red node in Fig. 4(a)). At step 2, FSLL shrinks longer prefix nodes with the least load into their precursor prefix nodes, as shown in Fig. 4(b). For example, a prefix node {R_5} with load 1 at level 3 is shrank into its precursor prefix node {R_3} at level 2. At step 3, FSLL proceeds to shrink longer prefix nodes with the lead load, until the number of prefix nodes is no more than the maximum number of TCAM entries. Fig. 4(c) shows that there remain just 8 prefix nodes that the TCAM table contains, after a prefix node {R_4} is shrank into its precursor prefix node {R_7}. Finally, FSLL encodes 8 prefix nodes in Fig. 4(c) into a TCAM table of 8 entries in descending order of the node width (see Fig. 5(a)). We assume that there is a content name */a/ab/abc/abcd* as input to be searched in the TCAM table (in Fig. 5(a)). We use its three-level shape /1/2/3 as key to search the TCAM table, because the TCAM table allows the first three levels of shapes. We find the first matching TCAM entry 0110** that points to a hash table containing two name prefixes {R_3, R_5}. Finally, we find the longest matching prefix R_5 (*/a/ab/abc/**) by using a content search in the hash table (see the next subsection).

However, the TCAM-based shape search module may yield false positives, where a shape of a content name matches that of a name prefix even though the content name does not match the name prefix. For example, a content name */a/cd/abc* is feed to the TCAM table in Fig. 5(a). We use the shape /1/2/3 as key to search the TCAM table and find the first matching TCAM entry 0110** pointing to a hash table {R_3, R_5}. But the content name */a/cd/abc* does not match R_3 (/a/ab/*) or R_5 (/a/ab/abc/*), yielding a false positive error. To eliminate the false positives and guarantee correct lookups, we add a precursor pointer into each prefix node (or each hash table) in a shape trie as shown in Fig. 5(b). When a content name does not match any name



(a) Data structure of a dual fingerprint-based hash table.

(b) A dual fingerprint-based hash table for two name prefix rules R_3 and R_5.

Figure 6. Illustration of a SRAM-based content search module in SACS.

prefix in a hash table, SACS jumps to its precursor hash table pointed by its precursor pointer for the longest matching prefix. As shown in Fig. 5(b), a prefix node $\{R_3, R_5\}$ maintains a precursor pointer to a prefix node $\{R_1\}$ in the shape trie. For a content name */a/cd/abc*, there is no match found in a hash table $\{R_3, R_5\}$. We proceed to search it in the precursor hash table $\{R_1\}$, and finally find the longest matching prefix R_1 (/a/*).

C. SRAM-based Content Search Module

We describe the design of a SRAM-based content search module. This module aims to find the longest matching prefix in a hash table pointed by the matching entry in TCAMs. Like prior work [4, 12, 13, 14], we use a fingerprint-based hash table (FHT) to implement a SRAM-based content search. In a FHT, a search requires a string comparison between a content name and a name prefix when their fingerprints match. However, a FHT has a key issue of the cluster effect, where too many name prefixes with the same shape and same fingerprint congregate in a slot of a hash table. For example, three name prefixes */a/ab/**, */a/ab/abc/**, and */a/ab/abcd/** have the same two-level shape /1/2. These prefixes are mapped to a slot in a hash table because they have the same fingerprint of their two-level string /a/ab. Given that a content name */a/ab/ab/abc* is searched, it is mapped to the same slot that stores above three name prefixes because it has the same two-level string */a/ab* as these prefixes. The search requires three string comparisons to find the longest matching prefix */a/ab/**, which slows down the performance.

To address the above issue, we propose a dual fingerprint-based hash table (DFHT) for alleviating the cluster effect and improving the content search performance. In a DFHT, each slot maintains two types of fingerprints: a content fingerprint and a shape fingerprint. The content fingerprint is a hash value of a name prefix or a content name, while the shape fingerprint consists of the prefix level and prefix length. Fig. 6 shows the DFHT data structure. A DFHT is an array of m buckets each with a linked list of slots. Each slot has a sorted list of cells in descending order of the prefix level, each of which has a linked list of prefix pointers to name prefix rules mapped to the slot, as shown in Fig. 6(a). We note that each slot contains a content fingerprint and a slot pointer, while each cell in a slot contains a shape fingerprint (the prefix level and prefix length) and a cell pointer. Fig. 6(b) shows an example DFHT for two name prefix R3 and R5 with the same two-level shape /1/2. R3 and R5 are mapped to bucket 1, and share the same content fingerprint 9527 because they have same two-level string /a/ab. There are two sorted cells that contain R5 and R3 respectively. A search of a content name /a/ab/ab requires only one string comparison in the DFHT of Fig. 6(b). This is because the content name has the three-level length of 5 that is not equal to the prefix length of 6 in cell 1, and has the same two-level length of 3 with cell 2.

Algorithm 2: The lookup of a content name in SRAMs

```
Function Lookup(x)
// x: a content name carried in a packet.
// hash_table: a hash table that matches x's name_shape in a TCAM.
1:  while ((prefix_rule == SearchHashTable(hash_table, x)) == NULL) do
2:      hash_table = hash_table's precursor_hash_table;
3:  end while
4:  return prefix_rule;
Function SearchHashTable(hash_table, x)
5:  bucket = hash_table[hash(x)];
6:  for (each slot in bucket) do
7:      if (x.fingerprint == slot.fingerprint) then
8:          for (each cell in slot) do
9:              if (x.name_level == cell.prefix_level) and
                    (x.name_length == cell.prefix_length) then
10:                 return matching_prefix_rule in cell;
11:             end if
12:         end for
13:     end if
14: end for
15: return NULL;
```

Algorithm 2 shows the lookup algorithm of DFHT. For a content name, we first map it to locate a bucket in a DFHT by a single hash function, and then check each slot in the bucket to find the matching slot by comparing their content fingerprints. Next, we check each cell in the slot to find the matching cell by comparing their prefix levels and prefix lengths, and return the matching prefix rule in the cell. If no matching prefix rule is found in the hash table, we proceed to search the content name in its precursor hash table iteratively. We see that a search of DFHT achieves average-case O(1) hash lookup, which is much less than state-of-the-art BS with O(log$w$) [14, 15], where w is the maximum number of components of name prefixes.

TABLE 1. STATISTICAL CHARACTERISTICS OF NDN DATASETS.

NDN Dataset	Name Prefix Set				Name String Trace			
	Number of Prefixes	Avg Prefix Length (Byte)	Avg Prefix Levels	Max Prefix Levels	Number of Strings	Avg String Length (Byte)	Avg String Levels	Max String Levels
Blacklist	**933,846**	**40.0**	**4.6**	**14**	**9,338,460**	**102.6**	**10.1**	**26**
DMOZ	**3,152,589**	**21.8**	**2.8**	**18**	**31,525,890**	**68.0**	**6.8**	**26**
UNIF8	**10,000,000**	**47.5**	**5.5**	**8**	**100,000,000**	**107.8**	**10.5**	**18**
UNIF16	**10,000,000**	**82.5**	**9.5**	**16**	**100,000,000**	**142.5**	**14.5**	**26**

D. Incremental Updates

When a network changes or fails, SACS needs to support fast incremental updates by allowing dynamic insertions and deletions of name prefix rules. We next separately present the insertion and deletion algorithms of SACS.

Algorithm 3: The insertion of a name prefix rule in SACS.

```
Function Insert(x)
// x: a new name prefix rule inserted into a set.
// insert x's name_shape into tcam_table.
1:  for (each x's name_shape from the longest to the shortest) do
2:      find first_matching_entry in tcam_table;
3:      insert a new entry for x into tcam_table with spare space;
4:      break;
5:  end for
// insert x into hash_table pointed by first_matching_entry.
6:  InsertHashTable(hash_table, x);
Function InsertHashTable(hash_table, x)
7:  bucket = hash_table[hash(x)];
8:  for (each slot in bucket) do
9:      if (x.fingerprint == slot.fingerprint) then
10:         insert a new cell for x into slot;
11:         return true;
12:     end if
13: end for
```

Algorithm 4: The deletion of a name prefix rule in SACS.

```
Function Delete(x)
// x: an existing name prefix rule deleted from a set.
// delete x's name_shape from tcam_table.
1:  for (each x's name_shape from the longest to the shortest) do
2:      find first_matching_entry in tcam_table;
3:      delete first_matching_entry from tcam_table if no rule in the entry;
4:      break;
5:  end for
// delete x from hash_table pointed by first_matching_entry.
6:  DeleteHashTable(hash_table, x);
Function DeleteHashTable(hash_table, x)
7:  bucket = hash_table[hash(x)];
8:  for (each slot in bucket) do
9:      if (x.fingerprint == slot.fingerprint) then
10:         for (each cell in slot) do
11:             if (x.prefix_level == cell.prefix_level) and
                   (x.prefix_length == cell.prefix_length) then
12:                 delete x from cell;
13:                 return true;
14:             end if
15:         end for
16:     end if
17: end for
```

Algorithm 3 shows the insertion algorithm of SACS. When a new name prefix rule is inserted, we use its shape from the longest to the shortest to search a TCAM table for finding the first matching entry except the default entry with all wildcards.

TABLE 2. TCAM SPACE IN MB AND LOOKUP LATENCY IN NS

# Entries	Width	Space (Mb)	Latency (ns)
1K	**72**	**0.070**	**1.3**
5K	**72**	**0.352**	**1.8**
10K	**72**	**0.703**	**2.2**
50K	**72**	**3.52**	**2.9**
100K	**72**	**7.03**	**3.3**

Then, we construct a new TCAM entry for the name prefix rule, and insert it into a TCAM table when there is spare space in a TCAM. Next, we need to insert the name prefix rule into a hash table pointed by the matching TCAM entry. We map it to a bucket in the hash table, and then search the bucket to find the matching slot whose fingerprint matches that of the name prefix rule. Finally, we construct a new cell for the name prefix rule, and insert it into a sorted list of cells in the slot.

Algorithm 4 shows the deletion algorithm of SACS. When an existing name prefix rule is deleted, we use its shape from the longest to the shortest to search a TCAM table for finding the first matching entry. Then, we delete the matching entry from the TCAM table if there is no rule in the entry. Next, we need to delete the name prefix rule from a hash table pointed by the matching TCAM entry. We map it to a bucket in the hash table, and then search the bucket to find the matching slot, in which we find the matching cell. Finally, we delete the name prefix rule from the cell.

IV. EXPERIMENTAL EVALUATION

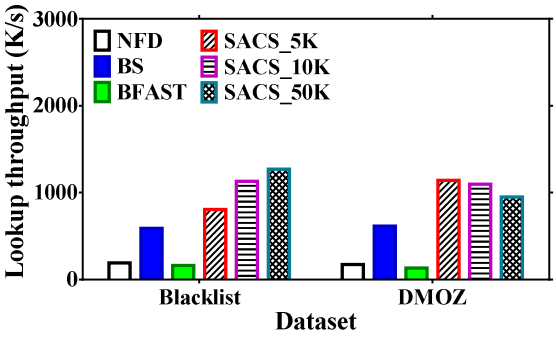
We conducted simulation experiments on real and synthetic datasets to evaluate the performance of SACS. We compare SACS with previous three hashed-based name prefix lookup schemes: NFD [12], BS [14], and BFAST [23]. In this section, we present the experimental methodology, and then provide the performance results.

A. Experimental Methodology

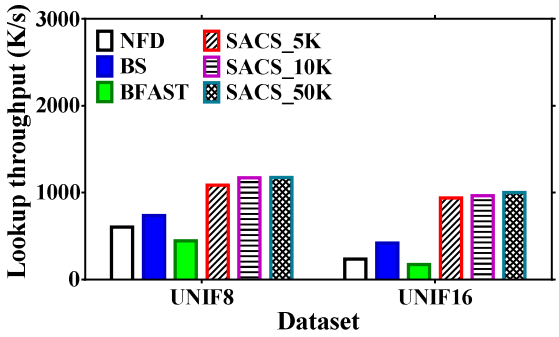
We use four sets of name prefix rules to simulate the FIBs, PITs, and CSs in NDN routers as shown in Table 1. We collect two real name prefix sets named Blacklist and DMOZ from the public URL datasets [30, 31]. Blacklist contains about 0.9M name prefixes each with the average length of 40 bytes and average level of 4.6. DMOZ contains about 3M name prefixes each with the average length of 21.8 bytes and average level of 2.8. Also, we generate two synthetic name prefix sets named UNIF8 and UNIF16 by combining random numbers of English words in a dictionary to synthetize name prefixes. Both UNIF8 and UNIF16 contain 10M name prefixes following a uniform

TABLE 3. NUMBER OF HASH LOOKUPS PER SEARCH AND NUMBER OF MEMORY ACCESSES PER SEARCH.

Solution	Blacklist		DMOZ		UNIF8		UNIF16	
	# Hash Lookups	# Memory Accesses	# Hash Lookups	# Memory Accesses	# Hash Lookups	# Memory Accesses	# Hash Lookups	# Memory Accesses
NFD	9.7	9.7	14.5	14.5	3.3	3.3	6.9	6.9
BS	3.2	3.2	3.7	3.7	2.3	2.3	3.2	3.2
BFAST	10.4	64.5	16.2	80.1	3.5	33.0	7.5	41.0
SACS_5K	1.1	1.1	1.8	1.8	1.1	1.1	1.0	1.0
SACS_10K	1.2	1.2	2.0	2.0	1.1	1.1	1.0	1.0
SACS_50K	1.4	1.4	2.3	2.3	1.2	1.2	1.1	1.1



(a) On Blacklist and DMOZ.



(b) On UNIF8 and UNIF16.

Figure 7. Lookup throughput on Blacklist, DMOZ, UNIF8, and UNIF16.

distribution among the levels of from 1 to 8 (or from 1 to 16). Each prefix in UNIF8 has the average length of 47.5 bytes and maximum level of 8. Each prefix in UNIF16 has the average length of 82.5 bytes and maximum level of 16. In addition, we generate four name string traces for the above name prefix sets by mimicking the content names carried in the NDN packets. We synthesize a name string by appending randomly generated suffixes to a name prefix from one of above name prefix sets. These traces are used to test the lookup and update throughput.

We run simulation experiments to separately test a TCAM-based shape search module and a SRAM-based content search module of SACS. First, we use a public TCAM modeling tool [32] to test the performance of TCAM-based shape search. We assume that our TCAM chip is manufactured with a 0.18*um* processor to calculate the latency of a single TCAM lookup. In the experiments, the total number of TCAM entries varies from 5K to 50K, and each entry has the fixed width of 72 bits. Table 2 shows the TCAM space and lookup latency. Second, we run experiments to test the performance of SRAM-based content search on a server with Intel Xeon CPU E5-2640×2 (8 cores, 2 threads/core, 2.6 GHz, L3 cache 2 MB) and 96 GB of main memory. For fair evaluation, we implement all schemes using chaining hash tables each with a MurmurHash hash function [27]. In addition, we set the false positive rate of CBFs as 0.1% in BFAST [23] to achieve optimal lookup performance. In the above experiments, we run ten trials with a single thread to measure three key performance metrics including the lookup throughput, memory consumption, and update throughput. We also measure the TCAM packing time of SACS. The results in the paper are averaged on ten runs.

TABLE 4. MEMORY CONSUMPTION IN MB.

Solution	Memory Size (MB)			
	Blacklist	DMOZ	UNIF8	UNIF16
NFD	32.8	110.8	351.6	351.6
BS	73.6	189.6	1020.5	1232.6
BFAST	296.0	999.2	3169.3	3169.3
SACS_5K	49.6	181.2	582.1	581.6
SACS_10K	51.6	182.5	583.9	583.4
SACS_50K	54.9	185.1	586.7	586.6

B. *Results on Lookup Throughput*

Fig. 7 shows the comparisons of lookup throughput in kilo operations per second (KOPS). We see that SACS outperforms previous schemes in lookup throughput. As shown in Fig. 7(a), SACS achieves the highest lookup throughput of up to 1271 KOPS on Blacklist and up to 1142 KOPS on DMOZ when the total number of TCAM entries varies from 5K to 50K. SACS significantly improves the lookup throughput by 4.2X to 6.6X over NFD, by 1.4X to 2.2X over BS, and by 4.9X to 8.7X over BFAST. Similarly, as shown in Fig. 7(b), SACS also achieves the highest lookup throughput of up to 1171 KOPS on UNIF8 and up to 1001 KOPS on UNFI16, which improves 1.8X to 4.3X over NFD, 1.4X to 2.4X over BS, and 2.4X to 5.7X over BFAST. Therefore, SACS achieves significant improvements in lookup throughput by up to 6.6X over NFD, by up to 2.4X over BS, and by up to 8.7X over BFAST.

To clarify why SACS is faster than previous schemes, we also measure the number of hash lookups and the number of memory accesses for a search as shown in Table 3. We see that

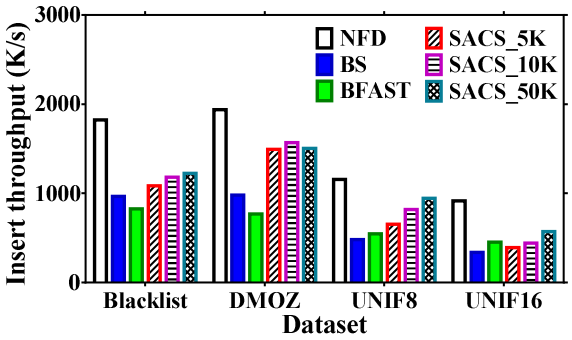


Figure 8. Insert throughput on Blacklist, DMOZ, UNIF8, and UNIF16.

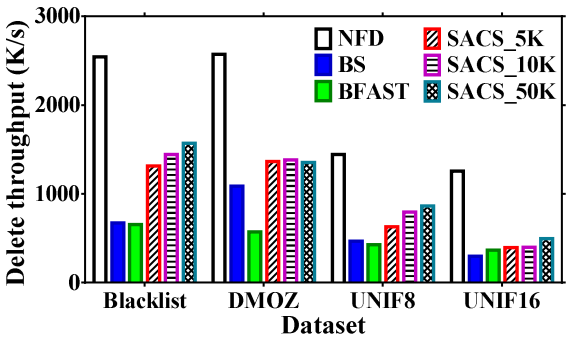


Figure 9. Delete throughput on Blacklist, DMOZ, UNIF8, and UNIF16.

SACS requires much fewer hash lookups and memory accesses per search than previous schemes. SACS requires 1.0 to 2.3 hash lookups per search, which is fewer than state-of-the-art BS by up to 65% on Blacklist, by up to 50% on DMOZ, by up to 55% on UNIF8, and by up to 68% on UNIF16. We note that the number of hash lookups per search increases with the increase of the number of TCAM entries from 5K to 50K. This is because a TCAM-based shape search module may yield more false positive lookups when the number of TCAM entries increases. Even so, with more TCAM entries, SACS achieves higher lookup throughput as shown in Fig. 7. This is because the lookup throughput of SACS is a function of the number of hash lookups and the time for hash lookups. When there are enough TCAM entries to pack the entire shape trie, the cluster effect in SRAM-based hash tables occurs with low probability, imposing less time for hash lookups.

C. Results on Memory Consumption

Table 4 shows the comparisons of memory consumption in SRAMs. We show the memory consumption of SRAM-based hash tables in SACS, since previous schemes are proposed for software. We see that SACS requires less memory than BS and BFAST by up to 53% and by up to 83% respectively. Also, when the number of TCAM entries varies from 5K to 50K, the memory consumption of SACS almost keeps constant because SRAM-based hash tables of SACS have different organizations but the same memory usage. We note that BFAST has the highest memory cost because of its additional CBFs. Moreover, SACS requires more memory than NFD, because SACS needs to maintain dual fingerprints and cells in a slot of hash tables. Nevertheless, SACS is up to 6.6X faster than NFD (see Fig. 7), at the cost of moderate memory consumption.

D. Results on Update Throughput

Fig. 8 shows the comparisons of insert throughput in KOPS. We see that SACS achieves higher insert throughput than BS and BFAST. Compared to BS, SACS improves up to 26% on Blacklist, up to 60% on DMOZ, up to 96% on UNIF8, and up to 69% on UNIF16, respectively. Compared to BFAST, SACS with more TCAM entries (i.e., 50K) also improves up to 48% on Blacklist, up to 104% on DMOZ, up to 72% on UNIF8, and up to 27% on UNIF16. In summary, SACS improves the insert throughput by up to 96% over BS and by up to 104% over BFAST.

TABLE 5. TCAM PACKING TIME OF SACS

Name Prefix Set	# Original Hash Tables	TCAM Packing Time (Second)		
		SACS_5K	SACS_10K	SACS_50K
Blacklist	131,490	0.25	0.24	0.17
DMOZ	127,072	0.26	0.23	0.17
UNIF8	3,684,921	10.8	10.5	10.1
UNIF16	7,563,964	34.9	34.7	28.7

Fig. 9 shows the delete throughput in KOPS. We see that SACS achieves higher delete throughput than BS by up to 133% on Blacklist, by up to 27% on DMOZ, by up to 84% on UNIF8, and by up to 67% on UNIF16. This is because SACS just needs to delete a name prefix from a hash table and does not adjust a sorted list of cells. Compared to BFAST, SACS improves the delete throughput by up to 140% on Blacklist, by up to 142% on DMOZ, by up to 102% on UNIF8, and by up to 36% on UNIF16. In summary, SACS improves the delete throughput by up to 133% over BS and by up to 142% over BFAST.

E. Results on TCAM Packing Time

Table 5 shows the TCAM packing time of SACS. We see that SACS is fast for constructing a TCAM-based shape search module. With FSLL, SACS requires up to 0.25 seconds for TCAM packing on Blacklist, up to 0.26 seconds on DMOZ, up to 10.8 seconds on UNIF8, and up to 34.9 seconds on UNIF16. In addition, when the number of TCAM entries increases (i.e., from 5K to 50K), SACS requires less TCAM packing time as TCAMs are enough large to encode all shapes without FSLL.

V. CONCLUSION

This paper presents a hybrid approach to scalable name prefix lookup with hardware and software. We propose SACS, a Shape And Content Search framework to accelerate name prefix lookups. In SACS, a TCAM-based shape search module is first used to determine a subset of possible matching prefixes, and then a SRAM-based content search module is used on the subset to find the longest matching prefix. In addition, we propose two techniques to improve the performance of SACS:

a first shrinking least load algorithm to pack large amounts of shapes in a small TCAM, and a dual fingerprint-based hash table to improve the content search performance.

Our experiments on real and synthetic datasets demonstrate that SACS outperforms previous schemes in lookup throughput, and requires up to 2.3 hash lookups per search. Compared to BS, SACS improves up to 2.4X lookup throughput, reduces up to 53% memory cost, and improves up to 96% insert throughput as well as up to 133% delete throughput. We show that SACS achieves high-speed lookups and low memory cost, while sustaining fast incremental updates.

REFERENCES

[1] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, K. Claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, "Named data networking," ACM SIGCOMM Computer Communication Review, vol. 44, no. 3, pp. 66-73, 2014.

[2] V. Jacobson, D. Smetters, J. Thornton, M. Plass, N. Briggs, and R. Braynard, "Networking named content," in ACM CoNEXT, 2009.

[3] H. Yuan, T. Song, and P. Crowley, "Scalable NDN forwarding: concepts, issues and principles," in International Conference on Computer Communications and Networks (ICCCN), 2012.

[4] W. So, A. Narayanan, and D. Oran, "Named data networking on a router: fast and DoS-resistant forwarding with hash tables," in ACM/IEEE ANCS, 2013.

[5] D. Perino, M. Varvello, L. Linguaglossa, R. Laufer, and R. Boislaigue, "Caesar: a content router for high-speed forwarding on content names," in ACM/IEEE ANCS, 2014.

[6] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Che, "Scalable name lookup in NDN using effective name component encoding," in IEEE ICDCS, 2012.

[7] H. Dai, B. Liu, Y. Chen, and Y. Wang, "On pending interest table in named data networking," in ACM/IEEE ANCS, 2012.

[8] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: from millions to billions," in ACM ICN, 2015.

[9] H. Yuan, P. Crowley, and T. Song, "Enhancing scalable name-based forwarding," in ACM/IEEE ANCS, 2017.

[10] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, and D. Yang, "Wire speed name lookup: a GPU-based approach," in USENIX NSDI, 2013.

[11] CCNx, http://blogs.parc.com/ccnx/.

[12] NFD – named data networking forwarding daemon, http://named-data.net/doc/NFD/current/.

[13] 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 IWQoS, 2014.

[14] H. Yuan and P. Crowley, "Reliably scalable name prefix lookup," in ACM/IEEE ANCS, 2015.

[15] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high speed IP routing lookups," in ACM SIGCOMM, 1997.

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

[17] A. Broder and M. Mitzenmacher, "Network applications of Bloom filters: a survey," Internet Mathematics, vol. 1, no. 4, pp. 485-509, 2005.

[18] S. Dharmapurikar, P. Krishnamurthy, and D. Taylor, "Longest prefix matching using Bloom filters," in ACM SIGCOMM, 2003.

[19] 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, 2013.

[20] 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.

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

[22] W. Quan, C. Xu, A. V. Vasilakos, J. Guan, H. Zhang, and L. Grieco, "TB2F: tree-bitmap and Bloom-filter for a scalable and efficient name lookup in content-centric networking," in IFIP Networking, 2014.

[23] H. Dai, J. Lu, Y. Wang, and B. Liu, "BFAST: unified and scalable index for NDN forwarding architecture," in IEEE INFOCOM, 2015.

[24] J. Shi, T. Liang, H. Wu, B. Liu, and B. Zhang, "NDN-NIC: name-based filtering on network interface card," in ACM ICN, 2016.

[25] H. Yuan and P. Crowley, "Scalable pending interest table design: from principles to practice," in IEEE INFOCOM, 2014.

[26] A. Kirsch and M. Mitzenmacher, "Less hashing, same performance: building a better Bloom filter," Random Structures & Algorithms, vol. 33, no. 2, pp. 187-218, 2008.

[27] MurmurHash, https://sites.google.com/site/murmurhash.

[28] OpenMP, http://www.openmp.org/.

[29] O. Rottenstreich and I. Keslassy, "The Bloom paradox: when not to use a Bloom filter," in IEEE INFOCOM, 2012.

[30] URL Blacklist, http://urlblacklist.com/.

[31] DMOZ, http://www.dmoz.org/.

[32] B. Agrawal and T. Sherwood, "Ternary cam power and delay model: extensions and uses," IEEE Transactions on VLSI, vol. 16, no. 5, pp. 554-564, 2008.