A Fast and Memory-Efficient Trie Structure for Name-based Packet Forwarding

Chavoosh Ghasemi[1], Hamed Yousefi[2], Kang G. Shin[2], and Beichuan Zhang[1]
[1]Department of Computer Science, The University of Arizona
[2]Department of Electrical Engineering and Computer Science, The University of Michigan

***Abstract*—Name lookup is an essential function, but a performance bottleneck in both today's and future network architectures. Variable-length and unbounded names rather than fixed-length addresses, as well as much larger and more dynamic forwarding tables call for a careful re-engineering of lookup structures for fast, memory-efficient, and scalable packet forwarding.**

We propose a novel data structure, called `NameTrie`, to store and index forwarding table entries efficiently and to support fast name lookups and updates. Its novelty lies in the optimized design and implementation of a character-trie structure. The nodes of `NameTrie` are stored compactly, improving cache efficiency and speeding up packet processing. Its edges are implemented using a hash table, facilitating fast name lookups and updates. A new scheme is used to encode control information without consuming additional memory. Running on conventional commodity hardware and using large-scale real-world name datasets, our implementation of `NameTrie` in software achieves 2.82∼3.56, 3.48∼3.72, and 2.73∼3.25 million name insertions, lookups, and removals per second, respectively, for various datasets while requiring a small memory footprint. We have conducted a comprehensive performance evaluation against the state-of-the-art of named data networking (NDN) as a typical use-case. It is shown to require at least 35% less memory and runs at least 3x faster for name table lookups and updates than two well-known trie-based schemes in NDN.

I. Introduction

Numerous emerging applications, such as firewalls, intrusion detection systems, load balancers, search engines, peer-to-peer file-sharing and content distribution networks, rely heavily on name-based services for packet forwarding, filtering, mapping, etc. Moreover, significant changes in the Internet usage have led to the rapid development of *information-centric networking* (ICN) architectures, such as *named data networking* (NDN) [1] and *content centric networking* (CCN) [2] (§II.A). NDN/CCN makes a fundamental shift in the semantics of network service from delivering the packet to a given destination IP address to fetching data identified by a given name. Thus, name lookup and update (i.e., insertion and removal of names) are core functions [3], [4], [5], [6], [7]. This paper focuses on packet forwarding tables, where name lookup hinges on longest-prefix matching (LPM), as it is easily applicable to other name-based services, such as filtering, mapping, etc.

Unlike IP addresses of fixed length (32 or 128 bits), names are variable-length with no upper bound, thus making it necessary to solve two main issues in designing packet forwarding engine: memory usage and name processing speed.

Memory Usage: The size of a name-forwarding table is, in general, much larger than that of an IP-forwarding table, because each name prefix is likely to be significantly longer than an IP prefix. Moreover, the number of name prefixes is orders-of-magnitude greater than that of IP prefixes [8]. (Taking the domain names as an example, there are already 330 million registered domain names in the Internet as of September 2017.) These factors together result in the forwarding tables with a much larger memory footprint than the current IP-forwarding tables with up to 700K IP prefix entries. Therefore, a memory-efficient data structure for the name-forwarding table is essential.

Speed: Since names are of variable and unbounded length, lookups of and updates to the name-forwarding table may take longer than those to the IP-forwarding table. More frequent updates to the name tables due to updates of forwarding rules or publication and withdrawal of contents also make name processing a bottleneck in the forwarding engine (§II.A). To support the service at a global scale, wire-speed lookup, insertion, and removal of names in large forwarding tables remains a great challenge.

Trie is an excellent candidate data structure for the LPM and has been widely used for name lookups and updates [6], [7], [8], [9], [10], [11], [12], [13] (§II.B). Tries, especially Patricia tries, improve memory efficiency by removing the redundant information among name prefixes. If two names[1] share a common sequence of characters (or bits or components depending on the granularity of trie) starting from the beginning of names, the common sequence will be stored only once in the trie. However, traversing a trie can be slow because accessing each level of the trie requires a memory access that cannot exploit CPU caches. To make it worse, we cannot always rely on special hardware, such as TCAM (Ternary Content Addressable Memory) and GPU (Graphics Processing Unit), to achieve high speed name-processing for two reasons. First, TCAM and GPU-based solutions suffer from power consumption and heating issues. Second, virtual network functions (VNFs) are now implemented more efficiently and flexibly using general-purpose CPU and in software [14]. Thus, implementing the trie structure in software is an attractive option.

[1]Without loss of generality, we assume the names are hierarchically structured and composed of multiple components separated by delimiters (usually slashes '/'). Flat names are a special case of hierarchical names with only one component.

We design and implement `NameTrie`, a novel character-trie structure that significantly improves both the memory efficiency and lookup/update speed of forwarding tables. A character-level implementation incurs less structural overhead than a bit-level trie [6], and removes more redundant information than a component-level trie [9], while providing the forwarding engine with the higher name-processing speed of the other two trie granularities (§III). The nodes in `NameTrie` are stored compactly to reduce the memory footprint, as well as to facilitate the use of CPU caches, which will reduce the number of memory accesses and speed up name lookups (§IV.B). The edges in `NameTrie` are stored in a hash table, and optimized for memory and speed (§IV.B). `NameTrie` also uses a new scheme to encode some control information without requiring additional memory (§IV.A). `NameTrie` solves the major problems of conventional trie-based approaches: deep trie structure, slow child lookup at each node, and inefficient use of pointers (§III).

In summary, this paper makes the right choice of trie granularity, proposes a new character encoding scheme, and optimizes the design and implementation of trie nodes and edges. We have conducted extensive experiments to evaluate the performance of `NameTrie` while comparing it against the state-of-the-art of NDN (which is an important use-case), demonstrating its significant benefits (§V). We used three large-scale datasets containing millions of URL-like names extracted from Alexa [15], Blacklist [16], and DMOZ [17], plus a 5-month-long traffic trace from a tier-3 ISP router. Running on commodity hardware, our software implementation of `NameTrie` achieved 2.82∼3.56, 3.48∼3.72, and 2.73∼3.25 million names per second in insertion, lookup, and removal, respectively, on different datasets while yielding a small memory footprint. We compared the performance of `NameTrie` against two well-known trie-based software solutions in NDN: Name Prefix Trie (NPT) [9] and Name Component Encoding (NCE) [10]. The lookup speedup is 11x NPT and 3x NCE, and the memory saving is 87% of NPT and 35% of NCE. The effect of multi-threading and URLs parsing for construction of name datasets are also demonstrated. These results corroborate our design as a practical solution for high-performance name lookups and updates, hence demonstrating its potential for broad impact on many areas beyond NDN/CCN, such as search engines, content filtering, and intrusion prevention/detection.

II. Background and Related Work

A. NDN and Applications

As a prominent design under the ICN paradigm, NDN [1] has gained significant momentum with participation from academia and industry. Companies including Huawei [18], [19] and Cisco [20], [21], [22] have made substantial R&D efforts on different aspects of NDN in recent years. Last year, Cisco acquired the PARC's CCN[2] (NDN's predecessor) platform to accelerate ICN insertion in 5G. The research community has also demonstrated its effectiveness for many real-world applications: IoT [23], [24], Hadoop and data centers [25], [26], sensor networks [27], and autonomous vehicles [28], [29], etc. The NDN community maintains a global testbed [30] to facilitate the development of NDN applications.

[2]NDN and CCN are two very similar ICN architectures whose differences do not affect the description and the considerations of this paper, so they will be used interchangeably in the paper.

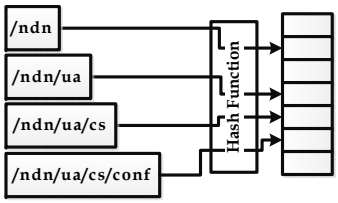
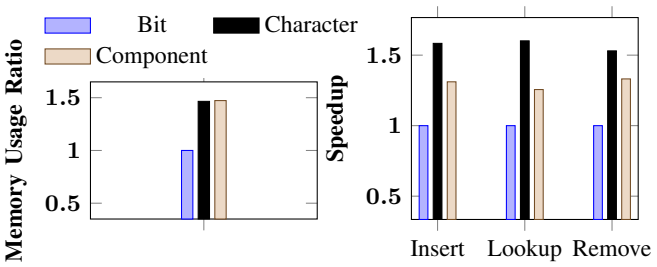


Fig. 1: Looking up */ndn/ua/cs/conf* in hash-table-based FIB. Since hash tables only support exact matches, they incur high overheads (both in memory usage and number of memory accesses) to find the LPM.

NDN uses content names rather than host addresses as the identifier at the network layer. Each request packet, called *Interest*, carries the name of the content that the application wishes to retrieve. The response packet, called *Data*, carries the content and its associated name. Neither type of packets contain the source or destination address, and thus the network is free to retrieve the content from anywhere, be it a cache, a server replica, or a mobile node passing by, as long as the content is what the user wants, i.e., Interest name matches Data name. While NDN brings many benefits such as mobility support [31], data-centric security [32], [33], in-network caching and multicast [34], there remain many challenges, one of which is name-based packet forwarding.

Nodes in NDN forward packets based on their names. (NDN design assumes hierarchically-structured names, akin to URLs composed of multiple components delimited by '/'.) By running a routing protocol [35], [36] or a flooding-based self-learning [37], each node populates its forwarding table (FIB) with rules consisting of name prefixes (e.g., */ndn* and */ndn/ua/cs*) and next hops. When an Interest packet needs to be forwarded, the node uses the Interest name (e.g., */ndn/ua/cs/conf*) as the key to look up FIB to find the longest-prefix match (LPM) (e.g., */ndn/ua/cs*), and forwards the packet to the corresponding next hop. The NDN packet forwarding process needs to store/insert a large number of names in FIB, look them up, and remove expired records. The forwarding table needs to be updated much more frequently than that of today's Internet. Unlike in IP networks, the FIB in NDN is under the control of a smart and adaptive forwarding engine which frequently updates the forwarding entries not only by consulting the routing plane, but also through continuous performance monitoring of the interfaces of each node. Furthermore, when contents are published (or deleted), name prefixes may need to be inserted into (or deleted from) FIBs. To deploy NDN at a global scale, the key problem is how to perform name-based LPMs fast and with a small memory footprint.



(a) Memory usage ratio **(b)** Processing speedup

Fig. 2: On the granularity of trie-based structures. We use the performance of the bit-level trie as the baseline for the comparison. According to our evaluation results, we choose the character-level trie as the basis of NameTrie since it provides a more balanced trade-off than the other two alternatives.

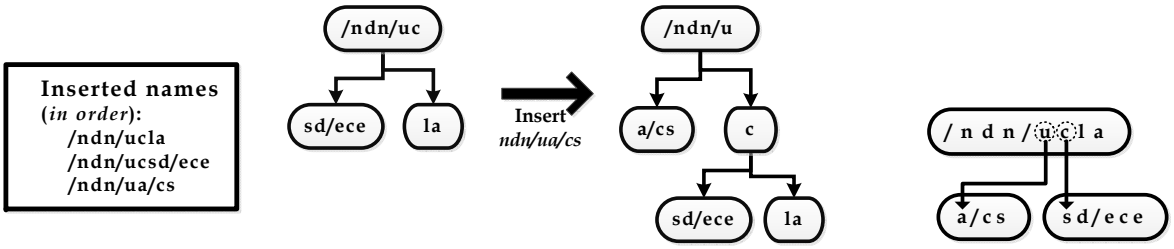
B. Related Work

Initial work on name lookups used data structures based on hash tables [3], [4], [5], [38]. Fig. 1 shows a simple example of using a hash-table-based approach. The FIB is stored in a hash table whose key is the hash of the name prefixes. Since the hash table only supports exact matching, it needs multiple lookups to find the LPM—this problem is often referred to as "prefix seeking." For example, for an incoming Interest with name */ndn/ua/cs/conf*, the LPM process can start from the shortest prefix (i.e., */ndn*) and increment by one component at a time, start from the longest (i.e., */ndn/ua/cs/conf*) and decrement by one component at a time, start from a particular length and then continue to shorter prefixes or restart at a larger prefix [39], or even start from the middle and do a binary search for the longest matching prefix [40]. Besides the prefix seeking, another problem of the hash-table-based approach is its large memory footprint. Since hash tables need to store the name strings for the purpose of collision resolution and verification, the names often need to be stored in their entirety. Even when multiple names share the same prefixes, these shared prefixes are still stored multiple times, thus consuming more memory. Several schemes [41], [42], [43] used bloom filters along with hash tables to improve the lookup speed and memory usage. Their benefit of reduced memory usage often comes at the expense of key verification and collision resolution. False positives will prevent the packets from being forwarded to the right destination. Any possibility of hash collision will undermine the integrity of the basic functions of the NDN routers. [44]. Finally, it is unattractive to incur such overheads (both in memory usage and number of memory accesses) to the system in order to use an exact matching structure for LPM. This inherent mismatch causes structural complexity for higher performance.

Trie, especially Patricia trie, is widely used in forwarding table implementation for its memory efficiency. Using this structure, searching for names that share the same prefix only needs to save one copy of the shared part. It also naturally supports both exact match and LPM. The traditional LPM designed for IP cannot be directly applied to name-based forwarding due to its unique characteristics and requirements. There are several existing proposals that implement name-based FIBs using trie structures. Parallel Name Lookup (PNL) [9] established component-level Name Prefix Tree (NPT) while speeding up lookups by selectively grouping trie nodes and allocating them to parallel memories. Name Component Encoding (NCE) [10] improves NPT by encoding name components and then looking up these codes to find the LPM. It uses the State Transition Arrays (STA) to implement both the Component Character Trie (CCT) and Encoded Name Prefix Trie (ENPT), thus effectively reducing the memory requirement while accelerating lookup speed. He *et al.* [11] used ENPT along with an improved version of STA, called *Simplified STA*, and a hash-table-based code allocation function. However, the encoding function is the performance bottleneck in both designs. Wang *et al.* [8] dealt with this bottleneck by leveraging the massive parallel processing power of GPU to achieve faster table lookups. Quan *et al.* [13] proposed an Adaptive Prefix Bloom filter (NLAPB) in which the first part of a name is matched by a bloom filter, and the later part is processed by a trie. Recently, Song *et al.* [6] proposed a bit-level Patricia trie that can compress the FIB significantly in order to fit it into high-speed memory modules. To achieve this, they also utilized "speculative forwarding", where instead of storing the names, only their differentiating bits are recorded. This technique, albeit effective, introduces false positives in lookups, in which case every packet is always forwarded to the next-hop, regardless whether the content name matches a FIB entry or not. Yuan *et al.* [7] adopted the main idea of [6] and proposed hash-table-based and trie-based data structures. However, like the prior paper, some packets may get wrongly forwarded. Besides, most existing work focuses on high-speed table lookup without considering table updates. So, they may suffer from slow name insertion and removal, which are important to the network performance.

III. Design Rationale

Given its memory efficiency and the support of LPM, trie, especially Patricia trie, is an excellent candidate structure for name-based packet forwarding. Although the binary trie implementation is often adopted by IP routing tables, LPM in name-forwarding tables is usually of component granularity. However, this should not be translated to using component-level trie as the best choice since it does not offer the best performance. Thus, before optimizing its memory usage and speed, we need to decide which trie granularity—component-, character-, or bit-level—to use. The factors to be considered for this decision are *compactness*, *depth*, and *internal overhead* of the trie structure. Compactness is concerned with how much of redundant information a data structure should avoid. For example, to store two name prefixes */ndn* and */ndn/ua*, a component-level trie can store the string */ndn* only once. If the name prefixes are */ndn* and */ndx*, a character-level trie will be able to store */nd* only once, but the component-level trie needs to store both names in their entirety because the trie nodes must fall on the boundary of name components. Therefore, in terms of compactness, bit-level $>$ character-level $>$ component-level where $>$ means "better than." Bit-level tries have the largest depth, which may lead to longer



(a) Inserting */ndn/ua/cs* in a conventional character-level trie, which causes node partitioning.

(b) Inserting */ndn/ua/cs* in an optimized character-level trie.

Fig. 3: Examples of trie-based FIB structures. A combination of several techniques is necessary to implement the optimized trie structure (we store nodes compactly in arrays; we implement pointers in branching characters and store them in a hash table; and, we utilize a new encoding scheme to embed control information in characters). The dotted circles show the branching points.

lookup times as traversing each level usually requires to access the main memory once. There are internal overheads including memory and processing. For example, if the trie is implemented by pointers from parent node to child nodes, then all the leaf nodes will have null pointers, which is a significant memory overhead. Another major contributor to the overhead is the processing required to decide which branch to go to at each level of the trie. In a bit-level trie, each node has up to 2 children, and checking which child node to go to is a quick, single bit operation. However, in a component-level trie, the number of children of each node could be unbounded, thus figuring out which child to branch to will require more time and potentially more memory.

In summary, the highest memory compression in the bit-level trie is achieved at the cost of maximal trie depth, which can degrade the speed. Moreover, the component-level trie reduces the trie depth at the cost of potential sparseness. This can lead to the generation of very fat tries since each node may be associated with a large number of children, incurring a higher internal overhead and slowing down the processing speed. Fig. 2 shows the comparison of non-optimized implementations of bit-, character-, and component-level tries in terms of memory usage and processing speed. There we used a dataset of 4M names (§V) and used the performance of the bit-level trie as the baseline for comparison. One can see that the bit-level trie consumes much less memory thanks to its compactness, but is also the slowest due to its trie depth. Component-and character-level tries are similar in terms of memory, but the former is slower due to its internal processing overhead at each tree level. Based on these findings, we decided to use a character-level trie since it gives a more balanced trade-off than the other two alternatives. Note that the absolute performance numbers may change under different implementations of the structures, but they preserve the relative merits.

Then, the main challenge lies in the optimization of the character-level trie. While we cannot change the compactness of the structure, we can improve its depth and internal overhead for smaller memory footprint and faster lookup speed. Fig. 3(a) shows an example character-level trie implemented using the conventional pointer technique. Inserting a name into an existing trie would make the trie one-level deeper and increase the number of nodes, both resulting in higher memory consumption and slower lookups—this problem is referred to as *node partitioning*, and is one of the major drawbacks of the conventional designs. Fig. 3(b) demonstrates the idea of `NameTrie`, which avoids partitioning existing nodes (i.e., */ndn/ucla*) in the trie, but adding pointers to branching characters (i.e., *u* and *c* in *ucla*). In this example, to store the same set of names, `NameTrie` has smaller depth and fewer nodes, meaning less memory usage and faster lookups. For example, LPM on the name */ndn/ucla* needs three memory accesses to fetch the corresponding nodes in a conventional trie, but only once in an optimized design.

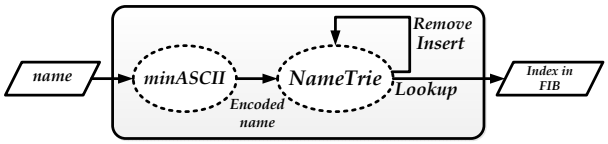


Fig. 4: NameTrie Overview. The main design of NameTrie consists of the *minASCII* name encoding and the NameTrie data structure supporting lookup, insertion, and removal of names.

We reduce the internal overhead in two ways. First, we encode the names with a new *minASCII* encoding, which uses the unused bits in ASCII characters to encode control information for our trie structure. Second, instead of including a number of pointers in each individual node, we use a hash table to store all the pointers of the trie. Not only does this solve the conventional implementation's waste of memory for storing the pointers of leaf nodes, but also enables fast decisions on which child node to branch to during lookups (as it pinpoints the corresponding child by sending a query to the hash table without iterating through children). The combination of these techniques makes it possible to implement the idea illustrated in Fig. 3(b) with minimal overhead.

Next we will discuss the architecture and implementation of this optimized trie structure.

IV. Design of NameTrie

The main design of `NameTrie` consists of (i) the *minASCII* name encoding and (ii) the `NameTrie` data structure. As illustrated in Fig. 4, the input to minASCII is a standard NDN/CCN name or name prefix, and the output is a sequence of bytes (minASCII codes). These minASCII codes are stored in the `NameTrie` data structure, facilitating the lookup, insertion, and removal of names.

minASCII	Definition	minASCII	Definition
0	EOP	32	US-ASCII Printable Characters
1	EOP (forwarding point)	... ...	
2	Slash		
3	Slash (forwarding point)	126	
4	Commonly Occurring Group of Characters (e.g., ndn)	130	[0-126]+128 (MSB ON for Branching)
...		...	
31		254	

Fig. 5: *minASCII* divides possible codes in NameTrie into four categories while encoding some control information in ASCII codes without requiring additional memory.

A. minASCII Encoding

An NDN/CCN name is a URL-like hierarchical name containing a sequence of components delimited by /. The current *de facto* NDN URI adopts the percent-encoding method, so NDN names include US-ASCII upper- and lower-case letters (A-Z and a-z), digits (0-9), and four special characters: PLUS (+), PERIOD (.), UNDERSCORE (_), and HYPHEN (-), as well as the PERCENT (%) symbol [45].

The Most Significant Bit (MSB) of all characters in the NDN URI scheme is found unused, as the ASCII code range of printable characters is 32–126. So, codes outside of this range (i.e., 0–31 & 128–255) can be used to facilitate the `NameTrie` data structure. As shown in Fig. 5, minASCII divides possible codes in `NameTrie` into four categories:

- Ordinary codes are in the range 32–126 for US-ASCII printable characters.
- Codes in the range of 4–31 can be assigned to frequently occurring group of characters (e.g., ndn, edu, etc.) to improve memory usage. Utilizing this opportunity could consume only a single byte for each group of characters, instead of a sequence of bytes. However, to simplify the presentation, we leave this optimization of encoding as our future work.
- The range 0–3 is reserved for *End Of Piece (EOP)* and delimiter (/).
- Codes greater than 128 are a simple repetition of the previous ones, but with their MSB set to 1 to indicate that this byte is also a *branching point*. A branching point is a character such as *u* and *c* in Fig. 3(b) as shown by dotted circles.

A node in `NameTrie` is also called a *name piece*, e.g., */ndn/ucla*, *sd/ece*, and *a/cs* in Fig. 3(b), each of which is stored as a sequence of bytes in an array. Unlike a name component, which is delimited by the special character /, a name piece can be any continuous sequence of bytes from a name. Fig. 6 shows `NameTrie` for a set of input names. Note that `NameTrie` is a general data structure in which the names do not necessarily share the same starting character. Thus, a dummy node can be used to connect multiple tries, each of which starts with a different character. However, in NDN, the root node of the trie contains byte /, which is a name piece shared by all NDN names.

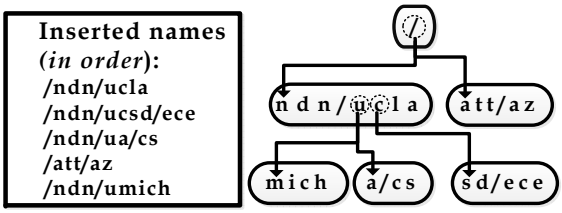


Fig. 6: NameTrie for a set of input names. The dotted circles show the branching points.

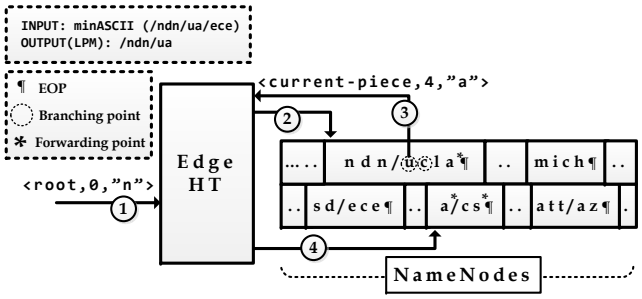


Fig. 7: Lookup of the name */ndn/ua/ece* in the NameTrie. For simplicity, characters are used instead of minASCII codes. *NameNodes* and *EdgeHT* have already stored the name pieces and their branching information according to Fig. 6.

Since we neither maintain name component boundaries nor partition nodes during insertion, to perform name lookup on `NameTrie`, we need to know (1) whether a byte is a *branching point*, i.e., the start of a trie edge; (2) whether a byte is a *forwarding point*, i.e., it marks the end of a prefix that has a corresponding entry in FIB; and (3) whether a byte is the end of a name piece (EOP). Using minASCII, we encode all this information without any memory overhead. For branching points, we set their MSB to 1. A forwarding point has to be either EOP or a component delimiter /. We use ASCII code 0 for EOP if it is not a forwarding point, and ASCII code 1 if it is a forwarding point. Similarly, we use ASCII code 2 for / if it is not a forwarding point, and ASCII code 3 if it is a forwarding point. For example, assuming a FIB has only two names: */ndn/ua/cs* and */ndn/ua*, we will use ASCII code 3 for the third slash, and add a byte of ASCII code 1 to the end as EOP. The first two slashes will use ASCII code 0.

B. NameTrie Data Structure

For a name, the characters of each piece are stored adjacently in an array, but the memory for different pieces are dynamically allocated when needed, so they may not be adjacent in memory. We call the memory space occupied by all `NameTrie` nodes collectively *NameNodes*. Fig. 7 shows how *NameNodes* store the trie nodes in memory. Note that to simplify the figure(s), we fill the *NameNodes* with characters, not their associated minASCII codes. As shown in this figure, */ndn/ucla* is the first inserted name and is stored as a whole in a single piece. A later inserted name, */ndn/ua/cs*, which shares the same prefix, is broken up and a new piece *a/cs* is inserted into the structure.

Avoiding node partitioning yields important benefits, such as less off-chip memory accesses, a higher CPU cache hit ratio, and a smaller memory footprint. The remaining problem is then how to implement the edges in the trie. `NameTrie` has many "edges," which connect a branching point in a parent node to the first byte of a child node. For example, we need an edge to point from *u* in *ndn/ucla* to *a* in *a/cs*. Following these edges, we can traverse the trie to look up a name. In a conventional trie implementation, these edges are implemented as pointers inside each node. For example, the node *ndn/ucla* will need to have a pointer to the node *a/cs*, and will need to record that this pointer starts from the letter *u*. Each name piece may have multiple branching points (e.g., *u* and *c* in *ndn/ucla*), and at each branching point, there can be multiple edges (e.g., *u* in *ndn/ucla* can be followed by pieces *mich* or *a/cs* in the next level of the trie) up to 128. Therefore, implementing these edges as in-node pointers would be complicated. In addition, if they are pre-allocated, they can waste memory, especially for leaf nodes. Furthermore, searching through multiple pointers within a single node can be slow.

Our idea is to collect all the edges of `NameTrie` and store them in a hash table, *EdgeHT*. The keys of this hash table are the tuples of <*parent node*, *offset*, *next byte*> to uniquely identify an edge in the trie. The *parent node* is the memory address of the node from which the edge originates, i.e., the address of the parent node. Within this node, the *offset* identifies the byte in the name piece where the edge starts. The *next byte* is the first character of the child node of this edge. Thus, this 3-tuple uniquely identifies an edge and serves as the key of *EdgeHT*. The return value of *EdgeHT* is the child node's address.

Fig. 8 shows two main components of `NameTrie`: (1) *NameNodes* which stores the trie's nodes, and (2) *EdgeHT* which implements the trie's edges. The processing functions are executed iteratively between these two components. By accessing *NameNodes*, the bytes of a name piece are fetched into the CPU cache for comparison with the encoded input name. As soon as a byte-mismatch is found, a query will be sent to *EdgeHT* to resolve the next piece, if available, and continue the name processing. *EdgeHT* plays an important role in speeding up `NameTrie`, as it directs the search towards the next name piece, if available. This is very different from conventional tries which must search the children of each matched node at a different level during the trie traversal. For example, in Fig. 3(a), if we search for the name */ndn/umich*, both the children of */ndn/u* will be checked to find the LPM. Instead, `NameTrie` can determine this by sending a single query to *EdgeHT*, which returns NULL, and will find LPM as */ndn*.

Hash table optimization: An important advantage of using a hash table is the high lookup speed for the entire key (a "fixed-length" entry substituting for the original variable-length string), using a hash function to compute its index in an array of buckets. However, HT lookup may face collisions. To solve the collision problem, we implement chaining in a cache-friendly way with embedded arrays instead of linked lists, i.e., each bucket is associated with an embedded array. The embedded array maintains the keys and their associated values in the form <*key*, *next piece*>. In case of a collision, the colliding item is added to the embedded array of the related bucket. Utilizing this array yields two main benefits. First, the collision is resolved, since the key of each item is also saved with its value in embedded array. Second, using this cache-friendly structure for collision resolution, frequent off-chip memory accesses are avoided. So, only a single memory access is required to retrieve an array of possible next name pieces. Note that *EdgeHT*'s load factor (i.e., n/m) is kept constant, where n and m are the total number of items and buckets in HT, respectively. Searching and removing an item, therefore, take a constant time.

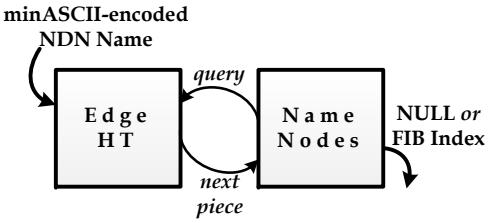


Fig. 8: Components of NameTrie data structure: *NameNodes* stores the trie's nodes, and *EdgeHT* implements the trie's edges. The processing functions are performed iteratively between these two components.

C. Processing Functions

Name Lookup is the key to the forwarding engine, which finds LMP for a given name. The names also need to be inserted in or removed from the forwarding tables frequently due to the updates of forwarding rules or the publishing and withdrawal of contents [8]. We present examples to clarify how `NameTrie` performs these processing functions.

Lookup: Fig. 7 shows the sequential steps in looking up the name */ndn/ua/ece* in `NameTrie`. We start from the root of the trie, which contains the character */*, followed by EOP. At each node, we try to match as many characters as possible before encountering a mismatching byte or EOP. (In this lookup, we encounter the EOP.) By checking the MSB (=1) of */*, we find edges to a lower level of the trie. So, we look up the tuple <*root node*, 0, *n*> in *EdgeHT*, which returns the next node's address, *ndn/ucla*. We then continue for a match between the input name and the name pieces in the trie node. We stop at the letter *c*, where a mismatch occurs. Again, we check the MSB of the last matching byte, *u*, to see if it is a branching point (these points are marked by dotted circle). *u* is shown to be a branching point from which some edges originate. Therefore, we look in *EdgeHT* to find the next node *a/cs*. For this name piece, the match will stop at letter *c*. The last matching byte */* has 0 in its MSB, meaning that there is no edge from that byte, thus terminating the lookup process. To determine the result of the lookup, we need to find the LPM along the lookup path. In other words, we need to check all the matching */* to determine if any of them is a forwarding point (i.e., code 3). In this case, the */* in *a/cs* denotes the end of a FIB entry name

and is the longest match, and hence the final result of this lookup is */ndn/ua*. Note that the LPM information is recorded as we go down the trie, so it is readily available when the lookup terminates. When the LPM is found, we look up the tuple <*node*, *offset*, *EOP*> in *EdgeHT* to obtain the pointer to the actual FIB entry. The pointer to the FIB entry is treated as another edge from a `NameTrie` node to a FIB entry.

Insert: is similar to lookup. Once the lookup terminates, instead of returning the LPM, we allocate a new node to store the remaining unmatched bytes as a new name piece. For example, to insert the name */ndn/ua/ece*, we need to allocate memory to create a new node to store */ece*, and insert an edge in *EdgeHT* to connect from *a/cs* to */ece*. Besides, while inserting names into our structure, forwarding points will also be marked. For example, consider insertion of the name */ndn/ua/cs* into our structure in Fig. 7, while we know the FIB table has two records for prefixes */ndn/ua* and */ndn/ua/cs*. After performing LPM and finding the last matched character (i.e., *c* in */ndn/ucla*), we add a new piece (i.e., *a/cs*) to the structure. However, there is some forwarding information about prefix */ndn/ua* in the FIB table, so that the last / is encoded with ASCII code 3, instead of code 2 to indicate that FIB knows how to forward the packets with this prefix. Similarly, as we have a FIB record for */ndn/ua/cs*, EOP is encoded with ASCII code 1, instead of code 0. These two bytes are marked with * in Fig. 7.

Remove: is the reverse process of insertion. It starts by looking up the name. Once an exact match is found, we delete the node and the edge from its parent node while single child nodes, if any, will be merged with their parents. This will continue recursively from lower to upper levels in the trie, as long as the name piece being deleted is not shared with other names. If the name piece is shared, for example, when removing */ndn/ucla* in Fig. 7, then we will still keep the node and the name piece, but will update the encoding of the bytes so that it does not represent a FIB entry name (i.e., forwarding point). In such a case, the code of EOP (which is marked by *) will change from 1 to 0, as the associated FIB record of this prefix is deleted.

We also analyze both the time and space complexities of `NameTrie` for all the processing functions in terms of the number of off-chip memory accesses and the memory footprint. Due to the space limit, we omit the details here and summarize the complexities as well as the notations used for complexity analysis in Table I.

TABLE I: Complexity analysis

	Definition
N	Number of names
d	Average number of pieces per name
d_r	Average number of pieces removed from each name
n	Average length of pieces
p	Average length of shared prefixes
	Time Complexity
Lookup	$O(2Nd)$
Insertion	$O(2Nd)$
Removal	$O(2N(d+d_r))$
	Space Complexity
NameTrie	$O(N((nd-p)+12d))$

V. Evaluation

We have conducted extensive experiments to evaluate the performance of `NameTrie` in terms of memory usage and name insertion, lookup, and removal speed. These results are then used to compare `NameTrie` with two well-known trie-based schemes in NDN/CCN: NPT [9] and NCE [10]. For a fair comparison, our implementation of `NameTrie` in software is not compared against those approaches that rely on massively parallel hardware (e.g., [8]), or do not guarantee forwarding correctness (e.g., speculative forwarding [6], [7] and lossy data structures [44], [13]). Although the focus of this paper is on the trie structures, for the purpose of comparison, we have also implemented a HT-based solution using the same strategy as in NFD (NDN's Forwarding Daemon) prototype [46], which is referred to as HT in what follows. Since to date, no real router hardware is available to run `NameTrie`, we have implemented and run `NameTrie` on a commodity PC, and compared the performance of different lookup structures on the same PC platform. This comparison will preserve the relative merits of different lookup designs.

A. Experimental Setup

We conducted experiments on a PC with an Intel Core i7 2.1∼2.8 GHz CPU, 6MB cache, and 8GB DRAM. This CPU consists of 4 real and 4 virtual cores, based on Intel's hyper-threading technology. All data structures are multi-threaded and implemented in C. We utilized real-world URLs, extracted from ALEXA [15], DMOZ [17], and Blacklist [16], as well as a 5-month-long traffic trace from a tier-3 ISP router, to construct three distinct name tables of 4,012,113, 9,347,389, and 12,012,255 entries for our experiments. For brevity, they will henceforth be referred to as 4M, 9M, and 12M datasets, respectively. Note that feeding the ordered names to `NameTrie` enables them to cache name prefixes, thus enhancing performance. To obtain unbiased results, the names were randomly shuffled, and therefore do not follow any particular order while being fed to `NameTrie`. Table II shows the specifications of each dataset. To extract more information from the datasets, the distribution of the number of characters in each component and the number of components in each name for each dataset are plotted in Fig. 9. For example, the 9M dataset is shown to have the longest average name size since most components have 12 characters and most names are composed of 5 components.

B. Results

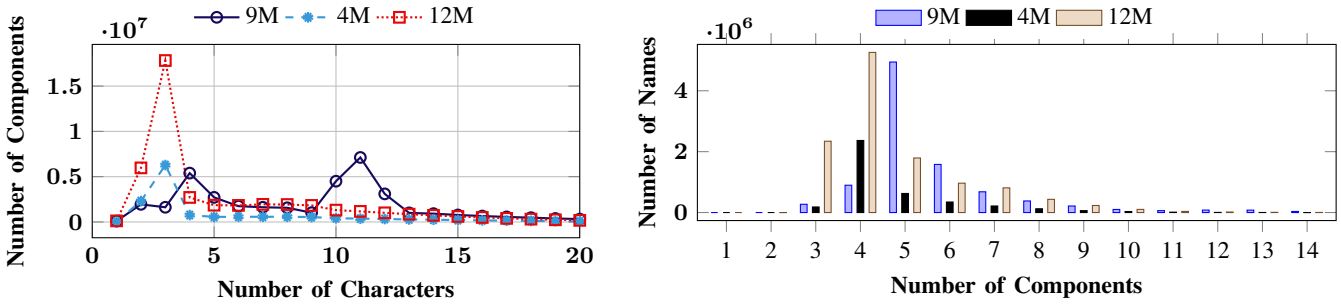
We first present the results related to the memory usage and the speed of different name processing functions. Then, the effect of multi-threading and URLs parsing for construction of name datasets are presented.

1) Memory Usage:

Table III shows the memory usage by different data structures, as well as `NameTrie`'s improvements over the others for various datasets. The overall results for the 12M dataset are

TABLE II: Dataset specifications

Dataset	4M	9M	12M
Sources	Dmoz(2015)	ISP Router Traffic	Dmoz(2015), ALEXA(2017), Blacklist(2016)
Number of names	4,012,113	9,347,389	12,012,255
Number of components	18,897,821	48,025,182	55,929,775
Number of components per name	4.74	5.64	4.65
Average component length (Byte)	4.55	6.73	4.56
Average prefix length (Byte)	15.38	8.74	13.21



(a) Distribution of length of components

(b) Distribution of number of components

Fig. 9: Dataset statistics

also plotted in Fig. 10, as names are inserted gradually (1M-by-1M names) into the data structures. (For smaller datasets, we omit the related figures due to space limitation.) The structures are shown to have a larger memory footprint for a larger dataset. The growth rate of `NameTrie` is, however, shown to be much slower than all the other structures, i.e., `NameTrie` scales well and can thus be used to store a very large number of names. It outperforms HT, NPT, and NCE by an average of 84.9%, 88.9%, and 40.9%, respectively. There are five main reasons for such a significant improvement. First, it does not use pre-allocated in-node pointers to implement trie edges, and thus prevents memory waste, especially in leaf nodes. Second, it employs embedded arrays, instead of linked lists, to resolve collisions, thus eliminating the need to store several pointers. Third, it does not store any part of the names in *EdgeHT*. Moreover, using fixed-length keys in *EdgeHT*, instead of variable-length string keys, it reduces memory consumption. Fourth, it prevents the insertion of redundant name prefixes by exploiting the character-level granularity. Finally, using minASCII code prevents any extra memory usage for control purposes. Note that `NameTrie`'s memory usage could be improved further if minASCII also compresses the commonly occurring groups of characters, each into a one-byte code (codes in the range of 4–31 as shown in minASCII table). However, utilizing the full power of minASCII is a part of our future work.

Due to longer names and components in the 9M dataset, as well as the smaller average prefix length, each name requires more memory to be stored (on average $\frac{423.09}{9}$ bytes per name, as shown in Table III). A longer average prefix length results in insertion of less redundant characters, and hence a reduction in memory usage. `NameTrie` and NCE are more memory-efficient, especially for the 9M dataset, because they use the character-level granularity during insertion, which can exploit the benefit of average prefix length. However, NPT has the worst/largest memory usage, because it stores all the components of each name (except for those matched in the prefix) in addition to several pointers in every node. Furthermore, to prevent false positives, HT saves a series of components as keys, resulting in a rapid increase of memory usage as more names are added to its structure.

2) Speed:

Fig. 11 compares the speeds of `NameTrie`, HT, NPT, and NCE for different processing functions when using the 12M dataset. The speed of a function is defined as the processing rate of input names in terms of millions of names per second (MNPS). To illustrate the improvements in each function's performance, we gradually insert the names into the structure. To measure lookup performance, the previously-inserted names are randomly selected, and for each name, either the whole name, a part of it, or an extended version of it (e.g., */ndn/../...* instead of */ndn*) is looked up. As shown in Fig. 11(a), NPT has the slowest name lookup due to its high internal processing overheads at each level, leading to high time complexity. Moreover, partitioning the nodes and utilizing the pointers result in a cache-unfriendly deep trie structure, incurring more off-chip memory accesses. NCE performs faster by using a state transition array (STA) to implement its modules (ENPT and CCT) for name lookup. It uses a binary search, reducing internal processing overheads and thus lookup speed significantly. HT is faster than NCE and NPT as it incurs less internal processing overheads by performing exact matches in each round. However, to prevent false positives, it requires several strings to be matched inside every bucket, thus limiting its speed.

`NameTrie` significantly outperforms all other data structures for several reasons. First, it avoids node partitioning while storing the name pieces compactly in *NameNodes*. This keeps the resulting trie's height shorter, thus increasing the speed of name lookups and updates. Second, it uses a hash table to implement edges, avoiding iteration through multiple pointers to reach a specific child at each level. Moreover, using a fixed key for this hash table makes significant speedup over variable-length string keys, because of its simple hash

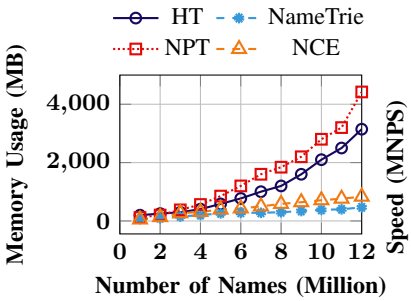
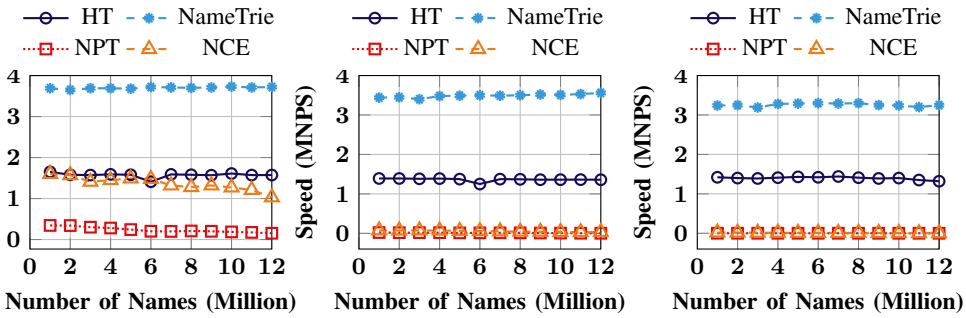


Fig. 10: Memory usage of different structures for 12M dataset. NameTrie outperforms HT, NPT, and NCE by an average of 84.9%, 88.9%, and 40.9%, respectively.



(a) Lookup **(b)** Insertion **(c)** Removal

Fig. 11: Name processing speed in different structures for 12M dataset. NameTrie achieved 2.82∼3.56, 3.48∼3.72, and 2.73∼3.25 million names per second in insertion, lookup, and removal, respectively, while significantly outperforming HT, NPT, and NCE.

TABLE III: Memory usage and Memory compression

Dataset	Memory Usage (MB)				Memory Compression		
	HT	NameTrie	NPT	NCE	NameTrie vs. HT	NameTrie vs. NPT	NameTrie vs. NCE
4M	357.81	143.2	530.84	291.7	85.15%	89.45%	43.87%
9M	2761.23	423.09	3491.62	651.07	84.67%	87.88%	35.02%
12M	3145.1	467.33	4421.42	832.9	85.14%	89.43%	43.89%

TABLE IV: Speed and Speedup

Dataset	Speed (MNPS)							
	Insert		Lookup				Remove	
	HT	NameTrie	HT	NameTrie	NPT	NCE	HT	NameTrie
4M	0.73	3.31	0.89	3.62	0.31	1.41	0.71	3.11
9M	1.25	2.82	1.55	3.48	0.16	1.02	1.38	2.73
12M	1.35	3.56	1.57	3.72	0.15	1.03	1.32	3.25

Dataset	Speedup				
	Insert	Lookup			Remove
	NameTrie vs. HT	NameTrie vs. HT	NameTrie vs. NPT	NameTrie vs. NCE	NameTrie vs. HT
4M	2.16x	2.14x	11.67x	2.56x	2.05x
9M	2.25x	2.24x	21.75x	3.41x	1.97x
12M	2.61x	2.36x	24.31x	3.61x	2.46x

computation and key comparison. (The comparison of fixed-length keys is much faster than that of variable-length keys). Third, it uses embedded arrays to implement chaining for collision resolution in a cache-friendly and memory-efficient manner, reducing the number of off-chip memory accesses. Fourth, it can effectively leverage multiple threads for parallel name processing.

As shown in Fig. 11(b-c), the update (insertion and removal) speed is very slow for NPT and NCE, because NPT needs to sequentially traverse nodes at each level while modifying several nodes pointers as well as their properties to insert new nodes or remove existing ones. Moreover, NCE needs to keep the STA updated, which requires a group of arrays and their entities to be resized and rearranged, degrading performance.

Fast updates, comparable to lookups, are an important benefit of NameTrie. Their operation speeds are lookup > insertion > removal, where '>' means faster. Name lookup is the initial step of insertion and removal, thus the fastest processing function. Name insertion starts in case of a mismatch during lookup and continues to store the rest of the name, thus slower. Name removal needs to go all the way down the trie, find the correct leaf nodes, and then remove the nodes backward, thus being the slowest.

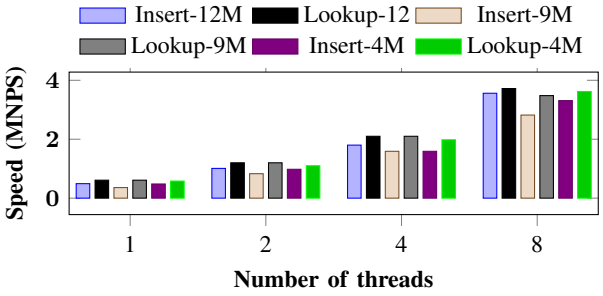


Fig. 12: NameTrie: lookup/update speed vs. number of threads.

The overall results and speedup of NameTrie are provided in Table IV. Since the number of insertions and removals in NCE and NPT is very low (below 0.01 MNPS for more than 4M names), they were not included in Table IV. For the 12M dataset, the lookup in NameTrie is 2.36x, 24.31x, 3.61x faster than HT, NPT, and NCE, respectively. These results indicate that our new data structure can meet the high-rate table update requirements of the NDN forwarding engine.

3) Multi-threading:

It is important for a software data structure to utilize the full capabilities of CPUs, not only caching but also multi-

threading.

As mentioned earlier, NameTrie is a cache-friendly data structure. To handle the cache stalling problem, we employ arrays to ensure that more relevant data will be fetched from memory. In particular, NameTrie uses *NameNodes* to preserve the unity of a name and reduce memory stall cycles during name traversal. Moreover, each bucket is mapped to an embedded array which contains all the keys hashed to that bucket. Using the embedded arrays, NameTrie will fetch more than one record with each memory access. Moreover, to further help reduce off-chip memory accesses and speed up the processing functions in NameTrie, we maintain the addresses of some frequently requested parts of *NameNodes* (like root's children) in L2, and subsequently in L3 caches. This is because our L3 cache memory is designed to be inclusive, i.e., the contents of L1 and L2 get replicated in L3.

NameTrie has great potential for multi-threaded implementation. It does not impose any limit, other than the number of threads, on parallel name lookups. Moreover, multiple names can be added to (or removed from) NameTrie independently if branching into different paths. This is because the paths in different sub-tries do not overlap with each other, thus providing opportunities to process names in parallel. This feature reduces time complexity significantly without increasing memory usage. False sharing is a major performance issue in symmetric multi-processor systems, which occurs when threads on different processors modify variables that reside in the same cache line. It invalidates the cache line and forces an update, which degrades performance. We reduce the frequency of false sharing by using local variables instead of global or dynamically-allocated shared data structures. Besides, false sharing decreases if the requested names, handled by parallel threads, are dispersed within the memory. Thus, for a batch of requested names, NameTrie exploits the locality of piece addresses as a heuristic for multi-threading, where a larger difference between two addresses usually implies more dispersion. Moreover, to reduce the cache misses for several names fed to one thread, it tries to find and then pick a name similar to the one processed most recently (two names are said to be *similar* if they share at least a few starting bytes). In this case, at least the starting and possibly more bytes of the name are still in cache, so a cache miss is less likely to occur. Otherwise, a name is selected randomly.

Fig. 12 shows how the speeds of both lookup and insertion in NameTrie are related to the number of threads. By employing 8 threads, we can insert and lookup 2.82∼3.56 and 3.48∼3.72 MNPS, respectively, on different data sets. The speed of each function increases linearly with the number of threads. In addition, Internet traffic is known to be bursty [47]. To reflect this reality in our experiments, we fed NameTrie each time with about 500K names simultaneously. NameTrie is shown to be able to effectively handle bursty requests of names in parallel, easily managing this basic behavior of the Internet.

4) Parsing Method and Performance Trade-offs:

Since NDN/CCN has not yet been standardized, a real FIB table of name entries is thus unknown. This section shows how the method of parsing URLs for constructing the name datasets can affect performance. It virtually changes the order of inserting input names, making a trade-off between processing speed and memory usage. The dissimilarity of consecutive input names has two main advantages. First, it increases the potential for parallel processing, thus speeding up insertion, lookup, and removal. Second, it increases the chance of keeping the name at a shallower level of the trie, so an entire name can be retrieved with a few memory accesses. On the other hand, more similarity of input names can reduce memory usage due to its high potential for aggregation.

TABLE V: Lookup speed vs. memory usage for 12M dataset. The method of parsing URLs for constructing name datasets makes a performance trade-off.

Parsing Method	# of starter pairs	Lookup speed (MNPS)	Memory usage (MB)
Method 1	6	1.13	371
Method 2	13	1.47	402
Method 3	1053	3.72	467

There are three possible ways to parse a name. We can start parsing from: (1) protocol portion of URLs (*http*, *ftp*, *https*, etc.), (2) TLD of URLs (e.g., *com*, *net*, *edu*, etc.), and (3) domain name of URLs (e.g., *google*, *youtube*, etc.). For example, parsing *www.google.com/new* using the third method outputs */google/www.com/new*. Table V shows the number of starting-byte pairs, where more different pairs create higher dissimilarity. The probability of having a batch of similar input names is pretty high using the first method, so we virtually face a sorted dataset, even if it is a random one. For the second method, since the number of starting bytes for TLDs is very limited, the probability of inserting consecutive similar names is still high. Having the maximum diversity of its starting-byte pairs, the third method makes an unsorted dataset of names. Table V also shows the trade-off between memory usage and processing speed. Using the third method results in a larger memory footprint, but makes more than 2x performance improvement over the others in terms of lookup speed. NameTrie can thus look up 3.72 MNPS while using 467MB of memory to store 12M different names. As each name is associated with a packet and assuming each packet is 256 bytes long [7], this lookup speed is translated to 7 Gbps using our software solution. Clearly, this throughput can be improved further on hardware-accelerated platforms.

VI. Conclusion

In this paper, we have designed and implemented NameTrie, a novel character-trie structure for fast and memory-efficient name lookups and updates. A right choice of trie granularity, a new character encoding scheme, and an optimized design and implementation of trie nodes & edges have come together to solve the major problems of conventional trie-based approaches. Our extensive experimentation with various large-scale real-world domain sets has shown NameTrie to achieve 2.82∼3.56, 3.48∼3.72, and 2.73∼3.25 million names per second in insertion, lookup, and removal,

respectively, while consuming a small memory footprint. Moreover, it requires at least 35% less memory and runs at least 3x faster in name lookups and updates than the state-of-the-art trie-based schemes in NDN/CCN, demonstrating the significant potential of `NameTrie`.

In conclusion, `NameTrie` is efficient and scalable, and hence applicable to contexts beyond NDN/CCN, such as search engines, content filtering, and intrusion detection.

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. CNS-1629009 and a Huawei grant. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the sponsors.

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 Comput. Commun. Rev.*, vol. 44, no. 3, pp. 66–73, 2014.

[2] "CCNx Project," http://www.ccnx.org/, 2017, [Online].

[3] B. Michel, K. Nikoloudakis, P. Reiher, and L. Zhang, "URL forwarding and compression in adaptive web caching," in *IEEE INFOCOM*, vol. 2, 2000, pp. 670–678.

[4] Z. G. Prodanoff and K. J. Christensen, "Managing routing tables for URL routers in content distribution networks," *International Journal of Network Management*, vol. 14, pp. 177–192, 2004.

[5] Z. Zhou, T. Song, and Y. Jia, "A high-performance URL lookup engine for URL filtering systems," in *IEEE ICC*, 2010, pp. 1–5.

[6] T. Song, H. Yuan, P. Crowley, and B. Zhang, "Scalable name-based packet forwarding: From millions to billions," in *ACM ICN*, 2015, pp. 19–28.

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

[8] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang, "Wire speed name lookup: A GPU-based approach," in *NSDI*, 2013, pp. 199–212.

[9] Y. Wang, H. Dai, J. Jiang, K. He, W. Meng, and B. Liu, "Parallel name lookup for named data networking," in *IEEE GLOBECOM*, 2011, pp. 1–5.

[10] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable name lookup in NDN using effective name component encoding," in *IEEE ICDCS*, 2012, pp. 688–697.

[11] H. Dai, B. Liu, Y. Chen, and Y. Wang, "On pending interest table in named data networking," in *ACM/IEEE ANCS*, 2012, pp. 211–222.

[12] Y. Li, D. Zhang, X. Yu, W. Liang, J. Long, and H. Qiao, "Accelerate NDN name lookup using FPGA: Challenges and a scalable approach," in *IEEE FPL*, 2014, pp. 1–4.

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

[14] H. Asai and Y. Ohara, "Poptrie: A compressed trie with population count for fast and scalable software IP routing table lookup," in *SIGCOMM*, 2015, pp. 57–70.

[15] "Alexa the Web Information Company," http://www.alexa.com/, 2017, [Online].

[16] "Blacklist." http://www.shallalist.de, 2016, [Online].

[17] "ODP–Open Directory Project ," http://www.dmoz.org/, 2015, [Online].

[18] R. Ravindran, A. Chakraborti, S. O. Amin, A. Azgin, and G. Wang, "5G-ICN: Delivering ICN services over 5G using network slicing," *IEEE Communications Magazine*, vol. 55, no. 5, pp. 101–107, 2017.

[19] S. O. Amin, Q. Zheng, R. Ravindran, and G. Wang, "Leveraging ICN for secure content distribution in IP networks," in *ACM Multimedia Conference (MM'16)*, 2016, pp. 765–767.

[20] I. Moiseenko and D. Oran, "Path switching in content centric and named data networks," in *ACM ICN*, 2017, pp. 66–76.

[21] M. Sardara, L. Muscariello, J. Augé, M. Enguehard, A. Compagno, and G. Carofiglio, "Virtualized ICN (vICN): Towards a unified network virtualization framework for ICN experimentation," in *ACM ICN*, 2017, pp. 109–115.

[22] J. Samain, G. Carofiglio, L. Muscariello, M. Papalini, M. Sardara, M. Tortelli, and D. Rossi, "Dynamic adaptive video streaming: Towards a systematic comparison of ICN and TCP/IP," *IEEE Transactions on Multimedia*, vol. 19, no. 10, pp. 2166–2181, 2017.

[23] J. J. Garcia-Luna-Aceves, "ADN: An information-centric networking architecture for the Internet of Things," in *ACM/IEEE IoTDI*, 2017, pp. 27–36.

[24] W. Shang, A. Bannis, T. Liang, Z. Wang, Y. Yu, A. Afanasyev, J. Thompson, J. Burke, B. Zhang, and L. Zhang, "Named data networking of things," in *ACM/IEEE IoTDI*, 2016, pp. 117–128.

[25] M. Gibbens, C. Gniady, L. Ye, and B. Zhang, "Hadoop on named data networking: Experience and results," in *ACM SIGMETRICS*, 2017.

[26] M. Zhu, D. Li, F. Wang, A. Li, K. K. Ramakrishnan, Y. Liu, J. Wu, N. Zhu, and X. Liu, "CCDN: Content-centric data center networks," *IEEE/ACM Transactions on Networking*, vol. 24, no. 6, pp. 3537–3550, 2016.

[27] S. S. Adhatarao, M. Arumaithurai, D. Kutscher, and X. Fu, "ISI: Integrate sensor networks to Internet with ICN," *IEEE Internet of Things Journal*, 2017.

[28] M. Chowdhury, A. Gawande, and L. Wang, "Secure information sharing among autonomous vehicles in NDN," in *ACM/IEEE IoTDI*, 2017, pp. 15–25.

[29] G. Grassi, D. Pesavento, G. Pau, L. Zhang, and S. Fdida, "Navigo: Interest forwarding by geolocations in vehicular named data networking," in *IEEE WoWMoM*, 2015, pp. 1–10.

[30] "NDN Testbed." https://named-data.net/ndn-testbed/, 2018, [Online].

[31] Y. Zhang, A. Afanasyev, J. Burke, and L. Zhang, "A survey of mobility support in named data networking," in *INFOCOM WKSHPS*, 2016, pp. 83–88.

[32] V. Perez, M. T. Garip, S. Lam, and L. Zhang, "Security evaluation of a control system using named data networking," in *IEEE ICNP*, 2013, pp. 1–6.

[33] Y. Yu, A. Afanasyev, D. Clark, k. claffy, V. Jacobson, and L. Zhang, "Schematizing trust in named data networking," in *ACM ICN*, 2015, pp. 177–186.

[34] E. Yeh, T. Ho, Y. Cui, M. Burd, R. Liu, and D. Leong, "VIP: A framework for joint dynamic forwarding and caching in named data networks," in *ACM/IEEE ICN*, 2014, pp. 117–126.

[35] A. K. M. M. Hoque, S. O. Amin, A. Alyyan, B. Zhang, L. Zhang, and L. Wang, "NLSR: Named-data link state routing protocol," in *ACM ICN*, 2013, pp. 15–20.

[36] C. Ghasemi, H. Yousefi, K. G. Shin, and B. Zhang, "MUCA: New routing for named data networking," in *IFIP Networking*, 2018.

[37] J. Shi, E. Newberry, and B. Zhang, "On broadcast-based self-learning in named data networking," in *IFIP Networking*, 2017.

[38] X. Li and W. Feng, "Two effective functions on hashing URLs," *Journal of Software*, vol. 15, pp. 179–184, 2004.

[39] 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, pp. 215–226.

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

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

[42] H. Lim, M. Shim, and J. Lee, "Name prefix matching using bloom filter pre-searching," in *ACM/IEEE ANCS*, 2015, pp. 203–204.

[43] 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, pp. 137–147.

[44] 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 *INFOCOM*, 2013, pp. 95–99.

[45] "NDN Name Format," https://named-data.net/doc/ndn-tlv/name.html\#ndn-name-format, 2017, [Online].

[46] "NFD–NDN's Forwarding Daemon." http://named-data.net/doc/NFD/current/, 2017, [Online].

[47] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of ethernet traffic (extended version)," *IEEE/ACM Transactions on Networking*, vol. 2, pp. 1–15, 1994.