2017 ACM/IEEE Symposium on Architectures for Networking and Communications

Enhancing Scalable Name-Based Forwarding

Haowei Yuan
Washington University
Department of CSE
St. Louis, Missouri 63130
hyuan@wustl.edu

Patrick Crowley
Washington University
Department of CSE
St. Louis, Missouri 63130
pcrowley@wustl.edu

Tian Song
Beijing Institute of Technology
School of Computer Science
Beijing, China 10081
songtian@bit.edu.cn

ABSTRACT

Name-based forwarding is a core component in information-centric networks. Designing scalable name-based forwarding solutions is challenging because name prefixes are of variable length and the forwarding tables can be much longer than seen with IP. Recently, the speculative forwarding method has been proposed, in which the forwarding structure size is proportional to the information-theoretic differences between the name prefixes rather than their lengths. In this paper, our goal is to enhance name-based forwarding performance with memory- and time-efficient data structures. We first define the string differentiation problem, based on the behavior of speculative forwarding in core networks, and then propose fingerprint-based solutions for both trie-based and hash table-based data structures. We experimentally demonstrate that the proposed solutions reduce the lookup latency and memory requirements. The proposed fingerprint-based Patricia trie decreases the average leaf-node depth and thus reduces the lookup latency. The proposed fingerprint-based hash table design requires only 3.2 GB of memory to store 1 billion names where each name has only one name component, and the measured lookup latency of the software-based single-threaded implementation is 0.29 microseconds. What's more, the distributed forwarding scheme presented in this paper makes name-based forwarding truly scalable.

1. INTRODUCTION

Information-centric networking (ICN) [1] has been proposed to address the mismatch between the original design of the Internet and its current usage. Several recently proposed network architectures, such as Named Data Networking (NDN) [2], follow the information-centric approach. In these proposals, each content has a unique name, and packets are forwarded according to name-based lookup results. Content names have variable lengths, and they could be human-readable hierarchical names, such as URLs, or flat self-certifying names [3]. Regardless of the naming scheme, scalable name-based forwarding is always a core component in these proposals.

Name-based forwarding is challenging for the following two reasons. Names are of variable and unbounded length, and the number of rules, which are name prefixes in the forwarding information base (FIB), is large. Variable and unbounded length affects lookup speed because the complexity of most of the traditional longest prefix matching methods designed for IP is proportional to the length of the rules. The number of rules affects the memory requirements because rules are typically stored in the forwarding structure to support the prefix matching algorithms. For instance, the software-based name prefix lookup engine proposed in [4] provides a lookup performance baseline of 10 Gbps with one billion name prefixes, where the system requires more than 100 GB of memory just for storing the name prefix strings. A smaller memory requirement always benefits both hardware-based and software-based solutions.

Recently, the speculative forwarding method [5] has been proposed to reduce the memory requirement and enable scalable name-based forwarding in core networks, and it works extremely well for forwarding rules that contain a large number of name prefixes with only one name component. In speculative forwarding, for the first-level name components (e.g., the first-level name component is `/a` for the name prefix `/a/b/c`), the core routers use a Patricia-trie like data structure, which stores only the information differences among the first-level name components rather than the complete name component strings. This way, the size of the forwarding structure is proportional to the information-theoretic difference among the rules, rather than the length of the names. As a result, the memory requirements of the forwarding rules that mostly have only one name component are significantly reduced. As shown in [5], the sizes of the forwarding structure in speculative forwarding for real-world datasets, such as the Alexa [6] and Dmoz [7] domain names, are much smaller than the memory requirements of the trie structures that store the complete first-level name components.

Our goal is to further improve the name-based forwarding performance by exploring time- and memory-efficient algorithms and data structures. To achieve this, we formulate the *string differentiation* (SD) problem, which is based on the speculative forwarding behavior in core networks, and identify the advantages that allow us to find efficient solutions to the problem. We focus on *exact string differentiation* (ESD), a special case of the string differentiation problem where no proper prefixes exist in the ruleset. Unlike exact string matching, strings in ESD only need to be differentiated rather than matched. Following the information-theoretic difference approach, we propose fingerprint-based methods to improve the forwarding performance. In essence, by transferring the information differences among name prefixes to fixed-length fingerprints, the differences are expressed more concisely. We propose a fingerprint-based Patricia trie (FPT) that reduces the lookup latency of the original Patricia trie-based methods proposed in speculative forwarding [5]. In terms of memory requirements, fingerprints are

978-1-5090-6386-4/17 $31.00 © 2017 IEEE
DOI 10.1109/ANCS.2017.16

more compact than name prefixes, giving opportunities for memory-efficient solutions to name-based forwarding. We propose a fingerprint-based hash table (FHT) that stores only the fingerprint of the name, reducing the memory requirements considerably. When the physical resources on a single machine cannot meet the requirements of large rulesets, a distributed forwarding scheme is needed. We study the distributed string differentiation problem, and evaluate the memory requirements of applying the proposed data structures to support distributed name-based forwarding. Finally, we have implemented the proposed FPT and FHT designs in software and applied software optimizations to better support large rulesets. We demonstrate the proposed solutions reduce the lookup latency and memory requirements with one billion names that each contains only one name component.

In this paper, we make the following contributions:

1. We formally define the *string differentiation* problem and propose fingerprint-based methods as a general approach to solve the problem.

2. We propose a fingerprint-based Patricia-trie design that reduces the average trie height (i.e., leaf-node depth) compared with the speculative Patricia trie presented in [5], thus the lookup latency is reduced. The FPT could also reduce the number of pipeline stages for hardware-based pipelined implementations.

3. To reduce the memory requirements further, a fingerprint-based hash table is proposed. Our proposed FHT design requires only 3.2 GB to store 1 billion names, reducing the memory requirements by 57% compared with Patricia trie-based methods. We experimentally show that the software-based single-threaded FHT implementation supports line-rate lookups. With 1 billion names, the lookup latency of FHT is 0.29 μs, and the throughput of the single-threaded program is expected to achieve 3.5 million packets per second.

4. To support larger forwarding rulesets, we propose distributed forwarding, where each router stores only a subset of the forwarding rules. We show that the proposed FPT and FHT works well with distributed forwarding in terms of memory requirements.

The rest of the paper is organized as follows. We present the background of name-based forwarding and speculative forwarding in Section 2. In Section 3, we define the exact string differentiation problem and discuss fingerprint-based approaches in general. We present the proposed fingerprint-based Patricia trie and fingerprint-based hash table in Section 4 and Section 5, respectively. Section 6 presents the proposed distributed forwarding and Section 7 discusses FIB update operations. In Section 8, we discuss software optimization techniques and present the performance evaluation for large datasets. Existing trie- and hash table-based solutions for name-based forwarding and IP forwarding are reviewed in Section 9. We conclude the paper in Section 10.

2. BACKGROUND

In this section, we review the challenges in name-based forwarding and illustrate the speculative forwarding method.

2.1 Challenges in Name-Based Forwarding

There are two primary challenges in scalable name-based forwarding. First, content names are of variable and unbounded length. IP addresses, which have a fixed 32-bit length, can be looked up efficiently using trie-based data structures. However, these methods cannot be directly applied to name-based forwarding because the complexity of trie-based methods is proportional to the lengths of the rules. The second challenge is that the forwarding table is much larger. Taking the domain names as an example, there are already 334 million registered domain names in the Internet as of 2016. What's more, there are 1.8 billion host names and 172 million active sites in the network as of January 2017 [8]. To understand the challenge, a plain text file that contains one billion domain names needs tens of GB of storage. Many recently proposed name-based forwarding designs store the complete forwarding rules in the data structures [4, 9, 10, 11], and thus have large memory requirements.

2.2 Speculative Forwarding

The speculative forwarding method [5] significantly reduces the memory requirements of name-based forwarding for datasets that have large numbers of prefixes with only one name component. Unlike [4], which provides a performance baseline that works regardless of the specific characteristics of the forwarding rules, speculative forwarding focuses on and works extremely well for forwarding rules that most have only one name component.

Speculative forwarding employs a Patricia-trie like data structure, which performs the path compression as the original Patricia-trie [12]. The original Patricia-trie still has large memory cost because names are stored to support prefix matching. Each name may have multiple components, for instance, name `/a/b/c` has components `/a`, `/b`, and `/c`. In speculative forwarding, for the first-level name components, only the information differences among the name components are stored in the speculative Binary Patricia-trie (sBPT). Thus, the strings of name prefixes that have only one component are no longer stored in sBPT. To support longest prefix matching, suffix name component strings still need to be stored. For example, if both `/a` and `/a/b/c` are in the FIB, the suffix string `/b/c` needs to be stored in order to properly differentiate name prefixes that match `/a` and `/a/b/c`. Hence, the memory requirements of names that have only one component, such as domain names, can be reduced significantly. Although the matching prefixes are not verified in core routers, the packets that truly have matching prefixes in the FIB are guaranteed to be forwarded correctly. Eventually the matching prefixes are verified in edge routers.

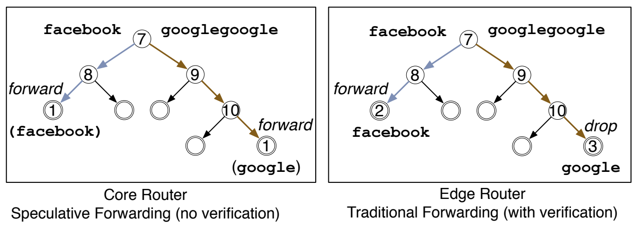


Figure 1: Speculative Forwarding Illustration

Figure 1 shows speculative forwarding in the core router and traditional forwarding in the edge router. In the sBPT shown on the left, only the bit positions that hold the information differences are stored. The name prefixes, as shown in parenthesis, are not stored. The edge router, shown on the right, stores the complete information. As shown in the figure, for a forwarding table that contains `facebook` and `google`, packets with name `facebook`, which is in the FIB, are forwarded correctly in the core router and are verified in the edge router. Packets with name `googlegoogle`, which is not in the FIB, are also forwarded by core routers but eventually get dropped in edge routers. For the Alexa top one million domain names [6], the sBPT requires only 5.58 MB of memory. The memory requirement of one billion synthetic name prefixes that have similar characteristics is 7.32 GB.

3. STRING DIFFERENTIATION PROBLEM

In this section, we define the *string differentiation* problem and propose fingerprint-based methods.

3.1 Problem Statement

The string differentiation problem can be defined as: *Given a set of strings R, for any query string $r_q \in R$, how to differentiate r_q from the strings in $R - r_q$?* The string differentiation problem is similar to the restricted candidate string problem [13]. Compared with the traditional string matching, string differentiation is in a more relaxed form because all the querying strings are assumed to be from the set R. In this paper, we focus on the *exact string differentiation* (ESD) problem, which is a special case of the SD problem, where there are no proper prefixes in R. The proper prefix is defined as follows, when a string r_a is a prefix of string r_b, and $r_a \neq r_b$, then r_a is a proper prefix of r_b. Traditional longest prefix matching methods can be used to support proper prefixes, as shown in Section 2.2. Thus, combining ESD and longest prefix matching effectively solves the SD problem.

3.2 Fingerprint-Based Solutions

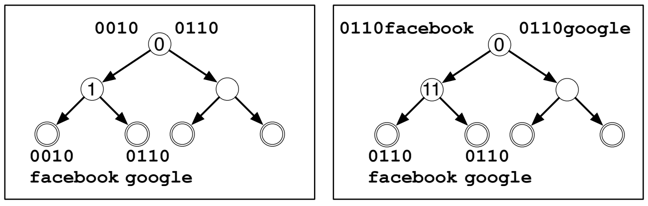
Fingerprints, which are generated by hash functions, are compact representation of variable-length strings. We propose fingerprint-based solutions to the exact string differentiation problem. For each string $r_i \in R$, a fingerprint f_i is generated via hashing. We discuss both perfect and non-perfect hashing in this section, and focus on non-perfect hashing in the rest of this paper.

3.2.1 Perfect Hashing

In perfect hashing, each string r_i is mapped to a unique fingerprint f_i. Figure 2a shows how perfect hashing-based fingerprints can be used in the sBPT. It can be seen that using only the fingerprints is sufficient to differentiate the querying names. Although perfect hash functions can be found [14, 15], it is challenging to support fast updates efficiently and requires additional storage to generate perfect hash values.

3.2.2 Non-Perfect Hashing

Non-perfect hashing can be computed more efficiently, but fingerprint collisions occur. We present two collision resolution methods to address this issue. The first method employs a separate collision table (CT) to store the fingerprint-



(a) Perfect Hashing (b) Non-Perfect Hashing

Figure 2: Perfect Hashing and Non-Perfect Hashing

colliding strings. If inserting a string causes a fingerprint collision, this string is inserted into the collision table. As a result, each lookup requires querying the collision table. The size of the collision table is determined by the fingerprint collision rate. The second approach stores additional information for the fingerprint-colliding strings in the original data structure. Each colliding string is assigned a local fingerprint generated by a different hash function, so that colliding strings can be differentiated by examining the local fingerprints. In a special case, the local fingerprint can be the original strings, as shown in Figure 2b. As can be seen, `facebook` and `google` share the same 4-bit fingerprint, and they are differentiated by examining the 11th bit position in the input string, i.e., bit position 7 in the original strings.

4. FINGERPRINT-BASED PATRICIA-TRIE

In this section, we present the fingerprint-based method to enhance the Patricia trie-based name prefix lookup design.

4.1 The FPT Design

Using fingerprints rather than name strings does not affect the Patricia-trie memory requirements because the number of nodes in the trie is always proportional to the number of the rules. Hence, we focus on reducing the lookup latency of the Patricia trie-based designs. The lookup performance of trie-based data structures is determined by the number of nodes visited during each query. As a result, decreasing the height of the trie could reduce the lookup latency. Additionally, for hardware-based pipelined implementations, the number of pipeline stages can also be reduced with smaller trie height values. The original Patricia-trie does not generate a minimum-height trie because it scans from the beginning of the name strings and split whenever there is a difference among the rules at a bit location. Consequently, the original sBPT may work well for randomly generated synthetic rules but not for real-world datasets, because the information differences may not be distributed evenly in practice. A better splitting approach would always select the bit that has the maximum entropy, which is similar to a decision tree. However, generating a minimum-height decision tree has been proven to be NP-complete [16].

We measured the leaf-node depths in the sBPT with the real-world Alexa [6] and Dmoz [7] domain name lists. Because we focus on the exact string differentiation problem, only the unique domain names are extracted from the datasets. The major characteristics of these two datasets are listed in Table 1. The average depths of the Patricia-trie structures with the Alexa and Dmoz datasets are 30 and 52 levels,

Table 1: Real-World Dataset Characteristics

Dataset	Rules	Domain Names	File Size
Alexa	1,000,000	990,821	15 MB
Dmoz	3,707,458	2,887,847	95 MB

respectively. Both are much larger than the optimal value $log(n)$, where n is the number of unique domain names.

Fingerprints are expected to have more balanced information difference distribution because they are generated by hash functions. To resolve fingerprint collisions, we could either use a collision table or prepend fingerprints to the names. For Patricia-trie, prepending fingerprints to the names is the simplest approach. It is worth noting that prepending a hash value to a fixed-length flow ID has been explored to reduce the average depth of level-compression tries [17], while we take the same approach on variable-length names to improve name-based forwarding performance. The height of the Patricia trie can also be reduced by dividing the complete datasets into smaller groups via hashing and then construct a subtrie for each group [5], however, prepending fingerprints is an orthogonal approach and can be applied to reduce the depth of each individual subtrie.

4.2 Experiments with Real-World Datasets

The original speculative Patricia-trie data structure can be used directly as a fingerprint-based Patricia-trie (FPT). The difference is that, in each insertion, deletion, and lookup operation, a fingerprint is prepended to the original name. In this paper, fingerprints are computed using the CityHash function [18].

We measured the impact of the fingerprint length on the leaf-node depth distribution. The entire Alexa or Dmoz domain names were inserted into the FPT, and the average Patricia-trie depth are shown in Figure 3. Note that the memory requirements of the original sBPT and the proposed FPT are the same because these two trie structures have the same number of nodes. As the fingerprint length increases, the average leaf-node depth decreases. When the fingerprint length is greater than $log(n)$, the average depth becomes stable. Take the Dmoz dataset for example, after prepending

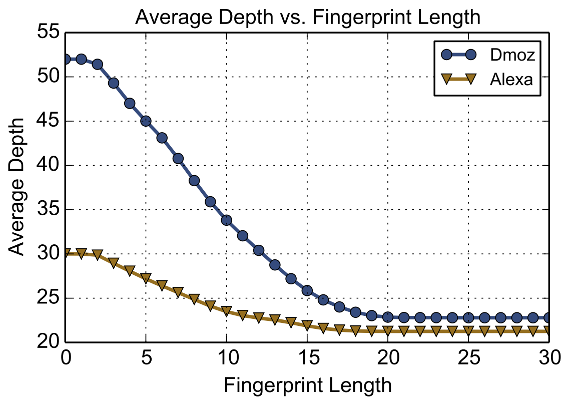


Figure 3: Reduced Trie Depth with Longer Fingerprints

20-bit long fingerprints, the average depth is reduced from 52 to 24.

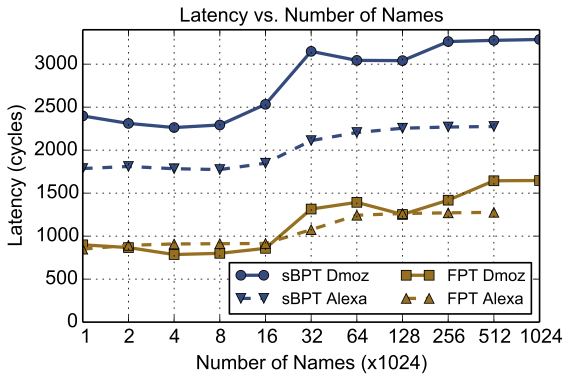


Figure 4: Patricia-Trie Variants Lookup Performance

We have also evaluated the lookup latency of fingerprint-based Patricia-trie in software. The experiment was performed on a machine equipped with 12 Intel Xeon E5-2630 cores operating at 2.3 GHz, 15 MB of L3 cache, and 192 GB of DDR3 memory. To measure the lookup latency, the entire Alexa or Dmoz domain names were inserted into the FPT, and then the names in the same dataset were looked up. The lookup latency was measured using the `RDTSC` instruction, which provides high precision and low processing overhead. We started with looking up $1,024$ names, then doubled the number of names each time, and eventually half a million names were looked up for Alexa and one million names were looked up for Dmoz. Each experiment run 100 times, and the average latency per lookup is shown in Figure 4. As can be seen, the lookup latency of the FPT outperforms the sBPT, which is expected because the number of memory references is reduced. Note that only the small-scale experimental results with real-world datasets are presented here, and the large scale experiments, which require further software optimizations, are presented in Section 8.

5. HASH TABLE-BASED METHODS

For software-based platforms, although FPT has smaller lookup latencies compared with the original sBPT, trie-based schemes still require considerable more memory references. As a result, the Patricia trie-based solutions are more likely to be implemented in purpose-built hardware with pipelining. To further reduce the lookup latency, in this section, we present hash-based designs for the exact string differentiation problem. The presented data structure can be used in both software-based and hardware-based implementations.

5.1 The FHT Design

Hash tables have been used widely in network applications [19], but traditional hash tables store the entire key strings, which contribute a large portion of the memory usage. Fingerprint-only hash tables [20, 21], which store only fingerprints of the keys, have been proposed for approximate membership querying and tolerate a small number of fingerprint collisions, i.e., false positives.

Unlike fingerprint-only hash table-based approximate membership querying designs, fingerprint collisions have to be resolved for the exact string differentiation problem because each lookup key needs to be uniquely identified. We propose a *collision free fingerprint-based hash table* (FHT) , which has small memory requirements with no false positives. Because the query strings in the ESD problem are from a known string set R, thus employing a collision table to store the colliding strings in the full string format resolves fingerprint collisions. In addition, the collision table can also store overflowed keys from the hash table, increasing the hash table load factors. Note that when the query string is not in R, i.e., packets with names that do not have any matching prefixes in the FIB, false positives could occur. This is a common issue faced by packet forwarding solutions that do not store the complete forwarding rules [22]. What's more, in name-based forwarding, the false positives can be eliminated by the edge routers. The edge routers store the full first-level name strings, and they will drop the packets with no matching first-level name strings or generate NACK responses to the requests [5].

The collision-free fingerprint-based hash table is a simple modification of the original fingerprint-based hash tables [20]. The only difference is that a collision table that stores full strings is introduced. During an insertion, if there is a fingerprint collision in the fingerprint-based hash table, then the complete string of the inserting key is stored in the CT. Because the CT stores full strings, keys with the same fingerprint can still be differentiated. During a lookup, the CT is queried first, and then the hash tables are looked up only if there is no match in the CT. As for deletion, the CT needs to be examined first. If the string is not found in the CS, the fingerprint can be deleted from the fingerprint-based hash table.

Intuitively, it is also possible to organize the FHT operation logic so that the fingerprint-based hash table is queried first, and the CT is looked up only when a fingerprint collision occurs. This way, the lookup latency can be further reduced because most queries require only lookups on the main hash table. However, this approach requires an additional bit for each fingerprint entry to indicate if there is a collision. What's more, it is also complicated to populate and update the hash tables. When a fingerprint collision happens, both the current inserting string and the previously inserted string with the same fingerprint need to be stored in the CT. However, because only fingerprints are stored in the main hash table, an additional insertion pass is required so that all colliding strings are stored in the CT. As a result, we choose to query the CT first for each lookup key. In the evaluation, we report the lookup latency with and without querying the CT to quantify the CT lookup overhead.

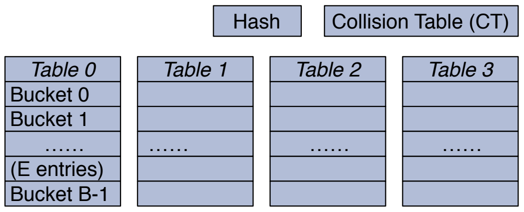


Figure 5: Fingerprint-Based Hash Table

Figure 5 shows the collision-free fingerprint-based hash table design. The FHT is based on a d-left hash table [19], where there are d subtables. Each subtable has B buckets, and each bucket has E entries. In our experiments, d = 4 subtables are used, and each hash bucket holds E = 8 entries. The load factor ld of the hash table is set to 93% to accommodate our largest experiment configuration where one billion (10^9) names are stored in a hash table with 2^{30} entries. Each entry stores a fingerprint and an outgoing port. The size of the CT, denoted as S_{CT}, is determined by the fingerprint collision rates and bucket overflow rates.

Because large-scale NDN forwarding rules are not available yet, the workload used in this paper is randomly generated. The workload contains one billion names, and each name has only one name component as we focus on the exact string differentiation problem. The average length of the name is about 30 bytes. We measured the memory requirements of storing one billion names with different fingerprint lengths w. The measured number of entries in the collision table (S_{CT}), collision table memory requirements (M_{CT}), hash table memory size (M_{HT}), and total memory requirements of the FHT (M_T) are listed in Table 2. The fingerprint-based hash table requires less memory storage than the fingerprint-base Patricia trie (7.32 GB as reported in [5]) because FHT also stores only the information difference among the rules, and what's more, there is no pointer storage.

Table 2: FHT Memory Requirements

w	**12**	**16**	**20**	**24**
S_{CT}	3.69E+6	3.24E+5	1.13E+5	1.01E+5
M_{CT}(MB)	1.76E+2	1.55E+1	5.41E+0	4.81E+0
M_{HT}(GB)	2.63	3.13	3.63	4.13
M_T(GB)	2.80	3.14	3.63	4.13

From Table 2, even in the case where 16-bit fingerprints are used, there are only 324K items stored in the collision table. The CT is also implemented as a 4-left hash table, and it is configured with 75% load so that no CT overflow occurs. Each CT entry stores a 32-bit fingerprint and a 64-bit name prefix pointer. Because each name string is about 30 bytes long, the estimated CT memory requirements is less than 16 MB, thus it can fit into a single SRAM chip. The total memory size of the FHT is 3.14 GB, which reduces the memory requirements of the original speculative Binary Patricia-trie (7.32 GB [5]) by 57%.

For one billion names, the d-left hash table requires approximately 3 to 4 GB of storage, which needs to be stored in DRAM. In a hardware-based implementation, each query first visits the on-chip SRAM-based collision table, and then looks up the DRAM-based main d-left hash table. In the worst case, each lookup requires d number of DRAM memory accesses; and the average case requires $(1+d)/2$ accesses. Because the d-left hash table is already a compact data structure that stores only fingerprints, previous works [23] on employing on-chip filters to reduce memory accesses cannot be applied. In a hardware-based design, with sufficient resources, it is possible to store each subtable into a separate DRAM modules, so that these d memory accesses can be pipelined or parallelized. In software-based designs, general purpose processors can maintain multiple memory requests to hide the access latency, and we present the impact of two known software optimization techniques in Section 8.

5.2 Experiments with Real-World Datasets

To evaluate the performance of software-based FHT, we used the same experimental setup as what we did with FPT to measure the lookup latency. Since the FHT load factor is kept as 93%, we started with looking up $1024 \times 93\% \approx 954$ names, then the number of names were doubled each time, and eventually $976,562$ names were queried. The performance results are shown in Figure 6. Because the collision table lookup can be offloaded in a hardware-assisted design, we present the FHT lookup latency with both querying the CT (denoted as w/ct) and skipping the CT (denoted as w/oct). We also include the previous FPT results for comparison.

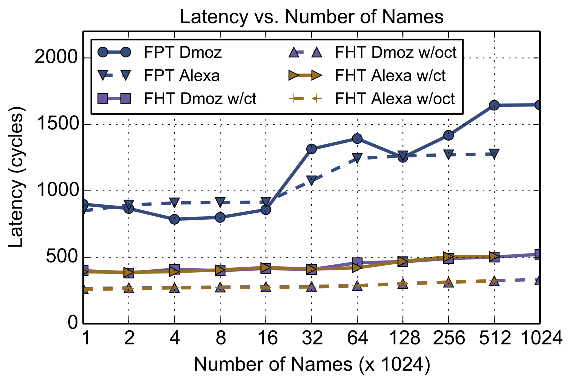


Figure 6: Hash Table Lookup Performance

From Figure 6, the hash table-based design outperforms the fingerprint-based Patricia-trie considerably. Although the CT size is small, there is still overhead associated with querying the CT before looking up the main d-left hash table in software.

6. DISTRIBUTED FORWARDING

When datasets are large, the memory requirements of a string differentiation problem may exceed the physical resources available on a single device. Hence, it needs to support a distributed scheme that solves the string differentiation problem collectively using a cluster of devices. In such schemes, each device stores a subset of the complete set R, and the subset is denoted as R_S.

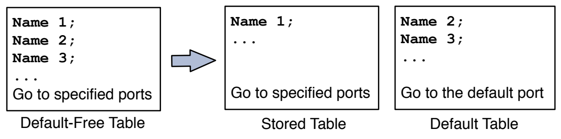


Figure 7: Distributed Name-Based Forwarding

Figure 7 illustrates the distributed name-based forwarding application. In this example, the complete FIB table, i.e., the default-free table, is divided into two tables, a *stored* table and a *default* table. The stored table contains the forwarding rules stored in the router, and each rule has a specific outgoing port. The default table contains the rules that are not stored, and packets with these name prefixes need to be recognized and forwarded to a default port. The distributed string differentiation problem is about differentiating any string $r_i \in R_S$, and recognizing strings in $R - R_S$, i.e., membership testing. To reduce the memory requirements, the data structure is typically built with only strings in R_S. However, when the data structure stores only the information differences among R_S, strings in $R - R_S$ are not guaranteed to be recognized. Our approach is to store a small amount of additional information for membership testing. We use the Patricia-trie to illustrate the approach.

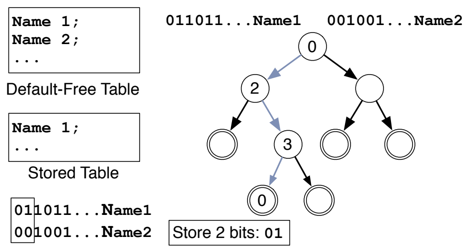


Figure 8: Distributed Forwarding Memory Optimization

In Figure 8, a Patricia-trie is built with the strings in the stored table, therefore, `Name1` is stored and `Name2` is not. When a packet with prefix `Name2` arrives, it is possible for `Name2` to reach the matching node of `Name1` if the bit values of the examined positions happen to match. For a FIB of size m, and a stored table of size n, the average number of collisions is expected to be m/n. Thus, $m/n - 1$ strings need to be recognized from the *owner* string of a matching node. In Figure 8, the least number of bits required to recognize the *owner* name are stored in the matching node. In this example, the first two bits of `Name1`'s fingerprint are stored.

6.1 Patricia Trie-based Approach

In distributed forwarding, additional information, such as local fingerprints, is stored in the Patricia-trie leaf nodes to recognize the owner name prefix. Because the size of the trie and the number of nodes visited in each query are not affected, we focus on the memory overhead of the additional information.

Figure 9 shows the memory overhead for distributed forwarding with the Alexa dataset. In our design, local fingerprints were generated by up to two hash functions. In the One Hash case, only one fingerprint was generated, and the number of bits to be stored at each node was noted. In the Two Hash case, two fingerprints were generated for each name, and then the fingerprint that required less number of additional bits is chosen. In addition, one more bit is required to indicate which hash function to use. We present both the average and the maximum number of additional bits. From Figure 9, as the number of colliding names per leaf node increases, the number of additional bits increases. Although the average number is close to $log(m/n)$, the maximum number is much larger and it determines the memory overhead if all of the leaf nodes have the same memory lay-

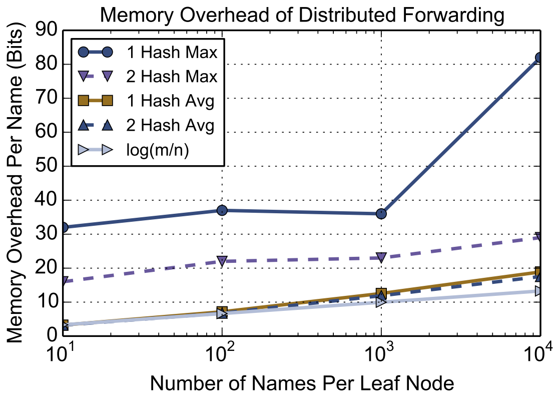


Figure 9: Memory Overhead of Distributed Forwarding

out. To reduce the memory requirements in practice, as most of the leaf nodes require close to $log(m/n)$ of bits, a threshold can be set, such as $log(m/n) + k$, where all the leaf nodes can store up to $log(m/n) + k$ bits, and then the names that require more bits would be inserted into the FPT so that they are differentiated by the trie structure.

6.2 Hash Table-Based Approach

For distributed forwarding, the fingerprint-based hash table can be more memory-efficient because a fingerprint is already stored in each entry. We use the dataset with one billion randomly generated names to evaluate the memory requirements of the distributed FHT. The FHT is configured with 4 subtables, 8 entries per bucket, 16-bit long fingerprints, and the load factor is kept at 93%. The FHT stores S_p percent of the one billion dataset, and the hash table size N was varied from 64M to 512M. In the first phase, $S_p \times N \times 93\%$ names were randomly chosen and inserted. In the second phase, the rest of the names were queried, and names with fingerprints matched in the FHT are stored in the collision table. The memory requirements of this distributed table (M) and the percentage of total memory (M_p) are listed in Table 3.

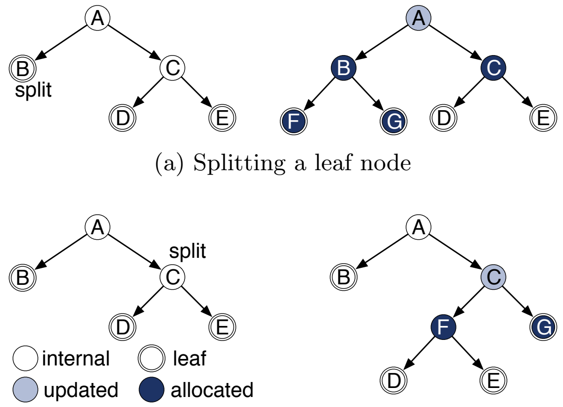
Table 3: Distributed Forwarding for One Billion Names

N	64M	128M	256M	512M
S_p(%)	6.25	12.50	25.01	50.01
M_p(%)	6.88	13.09	25.51	50.34
M(MB)	221.33	420.96	820.23	1619.69

From Table 3, the memory requirement of the subtable is always proportional to the number of rules stored in the table. Thus, FHT can be used to support distributed name-based forwarding with large datasets.

7. FIB UPDATES

The FIB table needs to be updated when routes change. When lossy data structures such as sBPT, FPT, and FHT are used, the FIB does not have sufficient information to process route updates directly. This problem also occurs in today's IP network when FIB compression schemes are used [24]. In practice, the FIB table can be incrementally updated with the assistance of a route controller (RC), which is employed in modern router architectures. The RC maintains the Routing Information Base (RIB), handles route updates, and generates FIB tables. The forwarding engines are distributed to multiple line cards, where they download the FIB table from the RC and then perform fast packet forwarding [25]. On the arrival of route update messages, the RC updates the RIB following the routing protocol and generates the new FIB tables. In general, the RC generates a sequence of *update instructions* to incrementally update the FIB structures in each forwarding engine. In this section, we describe the required instructions for both trie- and hash table-based FIB structures, and present the measured RIB update performance.



(a) Splitting a leaf node

(b) Splitting an internal node

Figure 10: Patricia-Trie Update

7.1 Patricia-Trie Updates

The original Patricia-trie maintains the complete information for the forwarding entries and supports online updates. Updating the forwarding information for an existing entry requires performing a lookup in the trie and modifying the related fields. Insertions and deletions are relatively complicated since nodes need to be allocated or deallocated in the process. Because Patricia-trie is a full binary trie, each insertion or deletion always requires creating or deleting two nodes.

The instruction generation for the lossy Patricia-trie (sBPT, FPT) needs to consider the node memory layouts. Each Patricia-trie node has two children, and maintaining both child pointers has a high memory cost. Thus optimized methods have been proposed [5], such as the single-memory address. In addition, internal nodes and leaf nodes have different memory layouts, so if the node type is changed, both this node and its sibling need to be reallocated, and their parent must be updated with the child's new address. In this paper, we use the single-memory address scheme because it reduces the memory cost. Here, each node stores the left child pointer, and the right child always stays next to the left child in memory.

Figure 10 shows the two cases of inserting an entry into the Patricia-trie. In Figure 10a, Node B needs to be split because the inserted entry differs from the entry stored in Node B. As a result, leaf nodes F and G are allocated to store these two entries. Node B now becomes an internal node, thus it needs to be reallocated. Because siblings always stay next to each other, Node C is also reallocated. In the end, Node A is updated with the Node B's new address. In Figure 10b, Node C needs to be split, although its node type is unchanged. In this case, Node F and G are created first: Node F copies the information from Node C, and Node G stores the inserted entry. Node C is then updated with the new bit position and child's address.

7.2 Hash Table Updates

Hash tables can be updated easily. The route controller maintains a similar hash table that stores the entire names for the occupied hash buckets. The RC generates the instructions to update the hash buckets or the collision table. When a new entry is inserted into the FHT, the entry eventually is stored either in a previously empty hash bucket or in the collision table. Deletions and updates have similar effects: either a hash bucket or a collision table entry is affected.

7.3 Update Performance

We use the insertion latency as the metric to evaluate the update performance. To see the trend of the latency as datasets become larger, we started with approximately one million ($2^{20} \times 93\% \approx 9.77 \times 10^5$) names, then doubled the dataset size at each step, and eventually reached one billion (10^9) names. Figure 11 shows the measured insertion latency.

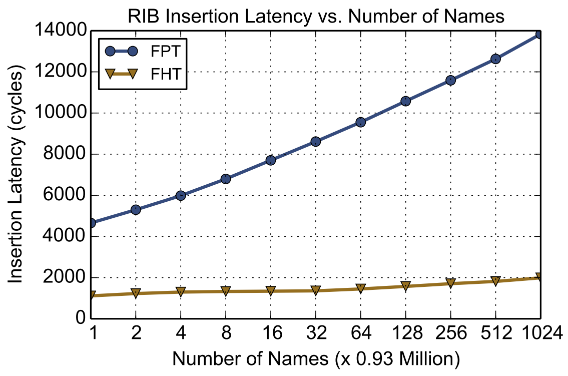


Figure 11: RIB Insertion Performance

For one billion names, the FHT requires 1,991 cycles to insert a name. As the frequency of the processor is 2.3GHz, thus FHT supports processing approximately 1.2 million packets per second (MPPS). Our Patricia-trie implementation is not yet heavily optimized. The FPT has a much higher latency due to more memory accesses, and with the largest dataset, on average 13,838 cycles are required for each insertion. Obviously, FHT supports faster updates, while a single FPT structure has limited scalability in terms of updates. In practice, large datasets can be split into smaller ones to be constructed independently for FPT.

8. EXPERIMENTAL EVALUATION

In this section, we evaluate the performance of the proposed designs with large datasets in software.

8.1 Software Optimization for Large Datasets

The characteristics of general-purpose multicore processors allow further optimizations for applications that require frequent memory accesses and large amount of memory storage. Specifically, software memory prefech instructions and large page sizes have been exploited to improve packet forwarding performance [10, 26, 21, 4]. We apply both software prefetching and large page sizes for the experiments with large datasets. Software memory prefetch instructions can be employed when the locations of future memory references can be known. Thus, it is straightforward to improve the performance of hash tables but not trie-based designs. With memory prefetching, all of the d subtables in the FHT are fetched at the same time, and the processor is able to handle multiple memory requests efficiently. Larger page sizes reduce translation lookaside buffer (TLB) misses, and we use both the 4KB and 2MB pages for FPT and FHT.

8.2 Lookup Latency

We measured the lookup latency of the proposed data structures with the same set of one billion synthetic names. Similar as the FIB update experiments, we started with approximately 1 million names, then the number of names in the dataset was doubled at each step. At each step, the forwarding data structure was built, and then all the names were looked up. We ran each experiment three times and recorded the average lookup latency. The measured results for the FPT are shown in Figure 12 and the results for FHT are shown in Figure 13.

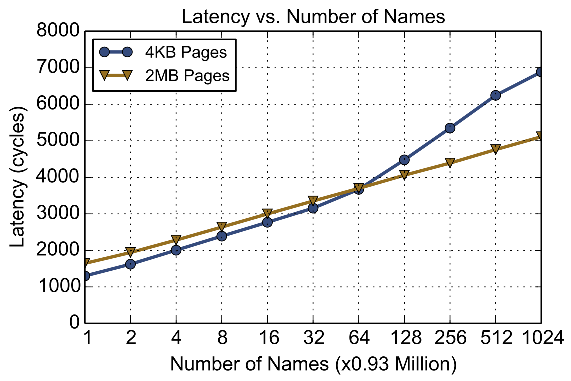


Figure 12: Fingerprint-based Patricia Trie Lookup Performance

According to Figure 12, the lookup latency of FPT increases linearly as the number of names doubles each time. When the datasets are small, i.e., less than 64 million names, the performance with 4KB pages performs slightly better than the one with 2MB pages. As the datasets become larger, the lookup latency with 2MB pages outperforms the one with 4KB pages. With one billion names, 5,112 cycles are required for each lookup. The processor's frequency is 2.3 GHz, thus roughly 0.44 million packets can be processed in each second. It is obvious that when the datasets are large, the software-based implementation of the FPT does not yield impressive results, and therefore, FPT would likely rely on hardware-based implementations.

As expected, the FHT performs much better than FPT

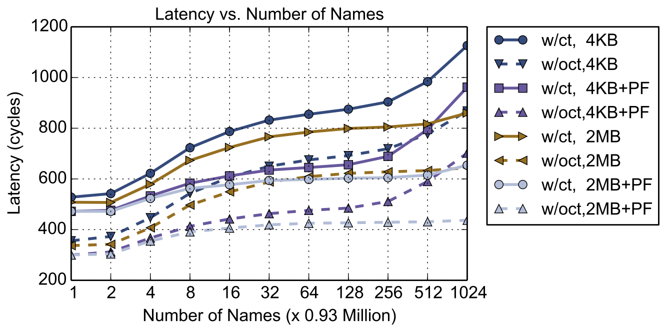


Figure 13: Fingerprint-based Hash Table Lookup Performance

in software. As shown in Figure 13, each lookup requires only 1,126 cycles without any additional optimization. With 4KB pages, the lookup latency increases sharply when the datasets become large, e.g., 256 million names or more, due to more TLB misses. With 2MB pages, the lookup latency increases moderately. Regardless of the configuration, querying the collision table always introduce additional latency. An interesting observation is that, when either only software prefetching or only 2MB pages is employed, the performance with software prefetching outperforms the one with 2MB pages when the datasets are relatively small, i.e., less than 512 million names. Eventually, the performance with 2MB pages outperforms the one of software prefetching with one billion names. This is what we expected as more TLB misses are likely to occur when the memory size becomes large.

The best lookup performance is achieved when both software prefetching and large pages are used. When the CT is queried, each lookup requires 654 cycles (0.29μs); when the CT lookup is offloaded, each lookup requires 437 cycles (0.19μs). Correspondingly, the single-threaded program is expected to be able to process 3.5 MPPS and 5.2 MPPS, respectively. Assuming each packet is 256 bytes long, the single-thread packet forwarding throughput is expected to achieve 7 Gbps and 10 Gbps, receptively. Because the FIB is used mostly for lookup operations, the throughputs of software-based implementations are expected to be improved further with multi-threading on multicore platforms.

9. RELATED WORK

Previous studies have explored both hash-based and trie-based data structures for name-based forwarding. Most hash table-based designs store the complete forwarding rules [27, 10, 11, 4, 9], thus they have large memory requirements when the datasets are large. Trie-based approaches reduce memory requirements as prefixes are shared by design, but the memory requirements of storing the shared name prefix are still considerable. Encoding methods [28] have been explored to reduce the memory requirements, with the cost of additional lookups to generate the encoded names. As for the designs that employ compact but lossy data structures, the bloom filter-based design proposed in [29] may introduce false positives even when there is only one name component in the name. The method proposed in [30] employs the original d-left fingerprint-based hash table with dynamic bit reassignment, but it does not address the fingerprint collision issue. The design proposed in [14] stores only fingerprints in the hash table using perfect hashing, however, the structure that used to generate perfect hash values requires additional storage. In addition [29, 14] do not guarantee to differentiate name prefixes with different numbers of name components, thus they may introduce false positives as hash collisions could occur between name prefixes with different numbers of components. In the recently proposed speculative forwarding [5], the string verification requirement in the core routers is relaxed, and thus reducing the memory requirements significantly. In our work, we focus on exact string differentiation and only names with one name component are considered. Names with multiple name components can be looked up in a separate structure that supports traditional longest name prefix match with no false positives.

Trie- and hash-based data structures have been extensively studied for IP packet processing. As for trie-based approaches, the Patricia-trie data structure has been considered for domain name lookups [31]. Recent works on Cuckoo hash tables [26, 21] have improved software packet processing performance on general purpose multicore processors considerably using software optimization. In this paper, these software optimization techniques are also applied to improve the name lookup performance.

10. CONCLUSION

Name-based forwarding is a core component in information-centric networking. Recently, speculative forwarding has been proposed and reduces the name-based forwarding memory size significantly by relaxing the string verification requirement in core networks for the first-level name components in name prefixes. In this paper, we define the string differentiation problem and propose fingerprint-based solutions to enhance the name-based forwarding performance. The fingerprint-based Patricia-trie effectively reduces the lookup latency of sBPT, and its performance can be improved further with pipelining techniques in practice. The fingerprint-based hash table reduces the memory size and lookup latency further in software. The FHT requires only 3.2 GB to store 1 billion names, with a lookup latency of 0.29 μs. Hence, the single-threaded software implementation of the FHT design supports packet processing rate at 3.5 MPPS, and its throughputs can be improved further on multicore or hardware-accelerated platforms. We also present distributed forwarding, which makes name-based forwarding truly scalable.

Acknowledgment

The authors would like to thank John Dehart and Jyoti Parwatikar for their support on the Open Network Laboratory. This work was supported by National Science Foundation grants CNS-1040643 and CNS-1345282.

11. REFERENCES

[1] Van Jacobson et al. Networking Named Content. In *CoNEXT '09*.
[2] Lixia Zhang et al. Named Data Networking (NDN) Project. Technical Report NDN-0001, NDN, 2010.

[3] Ali Ghodsi, Teemu Koponen, Jarno Rajahalme, Pasi Sarolahti, and Scott Shenker. Naming in Content-Oriented Architectures. In *ICN '11.*

[4] Haowei Yuan and Patrick Crowley. Reliably Scalable Name Prefix Lookup. In *ANCS '15.*

[5] Tian Song, Haowei Yuan, Patrick Crowley, and Beichuan Zhang. Scalable Name-Based Packet Forwarding: From Millions to Billions. *ACM ICN'15.*

[6] Alexa. http://www.alexa.com/topsites/.

[7] Dmoz. http://www.dmoz.org/.

[8] January 2017 Web Server Survey. http://news.netcraft.com/archives/category/web-server-survey/.

[9] Yi Wang et al. Wire Speed Name Lookup: A GPU-Based Approach. In *NSDI '13.*

[10] Won So et al. Named Data Networking on a Router: Fast and DoS-resistant Forwarding with Hash Tables. In *ANCS '13.*

[11] Diego Perino et al. Caesar: A Content Router for High-speed Forwarding on Content Names. In *Proc. of ANCS '14.*

[12] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching.* 1998.

[13] Frank K. H. A. Dehne et al., editors. *Algorithms and Data Structures, Third Workshop, WADS 93.*

[14] Yi Wang et al. Fast Name Lookup for Named Data Networking. In *Prof. of IWQOS '14.*

[15] N. Sertac Artan et al. TriBiCa: Trie Bitmap Content Analyzer for High-Speed Network Intrusion Detection. In *INFOCOM '07.*

[16] Ronald L Rivest. Inferring Decision trees Using the Minimum Description Length Principle. *Inform. Comput*, 80:227–248, 1989.

[17] R. Olsson and S. Nilsson. TRASH A dynamic LC-trie and hash data structure. In *HPSR '07.*

[18] CityHash. https://github.com/google/cityhash.

[19] Adam Kirsch et al. Hash-Based Techniques for High-Speed Packet Processing. In *Algorithms for Next Generation Networks.* 2010.

[20] Flavio Bonomi et al. An Improved Construction for Counting Bloom Filters. In *ESA'06.*

[21] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proc. of CoNEXT '14.*

[22] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations. In *Prof. of CoNEXT '09.*

[23] Sailesh Kumar and Patrick Crowley. Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems. In *Proc. of ANCS '05.*

[24] Zartash Afzal Uzmi et al. SMALTA: Practical and Near-optimal FIB Aggregation. In *CoNEXT '11.*

[25] H. Jonathan Chao. Next Generation Routers. In *Proceedings of the IEEE*, 2002.

[26] Dong Zhou et al. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *CoNEXT '13.*

[27] Haowei Yuan, Tian Song, and Patrick Crowley. Scalable NDN Forwarding: Concepts, Issues and Principles. In *ICCCN '12.*

[28] Yi Wang et al. Scalable Name Lookup in NDN Using Effective Name Component Encoding. In *ICDCS '12.*

[29] Yi Wang et al. NameFilter: Achieving Fast Name Lookup with Low Memory Cost via Applying Two-Stage Bloom Flters. In *INFOCOM '13, mini-conference.*

[30] Christian Esteve, Fabio Verdi, and Mauricio MagalhÃčes. Towards A New Generation of Information-oriented Internetworking Architectures. In *First Workshop on Re-Architecting the Internet*, 2008.

[31] C.A. Shue and M. Gupta. Packet Forwarding: Name-based Vs. Prefix-based. In *IEEE Global Internet Symposium, 2007.*