Scalable Name-Based Packet Forwarding: From Millions to Billions

Tian Song
School of Computer Science
and Technology
Beijing Institute of Technology
Beijing, P.R.China 100081
songtian@bit.edu.cn

Haowei Yuan
Dept. of Computer Science
and Engineering
Washington University
St. Louis, MO 63130
hyuan@wustl.edu

Patrick Crowley
Dept. of Computer Science
and Engineering
Washington University
St. Louis, MO 63130
pcrowley@wustl.edu

Beichuan Zhang
Dept. of Computer Science
The University of Arizona
Tucson, Arizona 85721
bzhang@cs.arizona.edu

ABSTRACT

Named-based packet forwarding represents a core characteristic of many information-centric networking architectures. IP-inspired forwarding methods are not suitable because a) name-based forwarding must support variable-length keys of unbounded length, and b) namespaces for data are substantially larger than the global address prefix rulesets used in today's Internet. In this paper, we introduce and evaluate an approach that can realistically scale variable-length name forwarding to billions of prefixes. Our methods are driven by two key insights. First, we show that, represented by binary strings, a name-based forwarding table of several millions of entries can be notably compressed by a Patricia trie to fit in contemporary fast memory of a line card. Second, we show that it is possible to design and optimize the data structure to make its size dependent only upon the number of rules in a ruleset, rather than the length of rules.

We reduce our designs to practice and experimentally evaluate memory requirements and performance. We demonstrate that a ruleset with one million rules based on the Alexa dataset only needs 5.58 MiB memory, which can easily fit in fast memory like SRAM, and with one billion synthetic rules it takes 7.32 GiB memory, which is within the range of DRAM in a line card. These are about an order of magnitude improvement over the state-of-the-art solutions. The above efficient memory size produces high performance. Estimated throughput of the SRAM- and DRAM- based solutions are 284 Gbps and 62 Gbps respectively.

ICN'15, September 30–October 2, 2015, San Francisco, CA, USA.
Copyright is held by the owner/author(s). Publication rights licensed to ACM.
ACM 978-1-4503-3855-4/15/09 ...$15.00.
DOI: http://dx.doi.org/10.1145/2810156.2810166 .

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Store and forward networks; C.2 [**COMPUTER-COMMUNICATION NETWORKS**]: Miscellaneous

General Terms

Algorithms

Keywords

Information-Centric Networking, Named Data Networking, Name-based Packet Forwarding, Longest Prefix Matching, Speculative Forwarding

1. INTRODUCTION

Recent information-centric networking (ICN) [1] concepts, such as Named Data Networking (NDN) [4], seek to address weaknesses in today's Internet by shifting the communication abstraction away from machine addresses and toward data names. Rather than forwarding packets based on their destination addresses, ICN forwards packets based on data names, such as URL like name prefixes in NDN. Benefits aside, this shift introduces a new, and some would argue more difficult, problem: forwarding name-based packets with variable-length names based on longest-prefix match against a global data namespace toward high throughput and strict memory requirements.

Longest prefix match (LPM) methods for IP forwarding have been remarkably successful. While innovation continues in this area, the core ideas introduced in the late 1990s, refined with strong ideas [6] [10] [12] based on prefix expansion and bitmap representations, have been sufficient to enable vendors to implement cost-effective LPM solutions across all performance tiers. With fixed-length IPs and fewer than 500 K rules [11], modern LPM implementations enable Internet-scale rulesets to fit comfortably within a few mega-bytes of fast memory, such as TCAM and SRAM, in order to achieve high performance. For IP, most agree that LPM has been a solved problem for some time.

However, those approaches for IP do not work well for name-based forwarding. Traditional LPM methods are optimized for IP prefixes but not designed for general namespaces, such as those used in ICN. Specifically, a scalable and efficient name-based forwarding scheme needs to successfully address the following challenges.

Complex name structure: Names, identifiers to data, have more complex and unrestrained formats than IPs. Different ICNs adopt different naming schemes. NDN uses URL-like hierarchical names [4], while some other ICN architectures [2] [3] use flat self-certifying names. A scalable name-based packet forwarding solution should be agnostic to naming schemes.

Large forwarding table: Compared to the size of forwarding information base (FIB) in core IP routers, the size of FIB for name prefixes can be much larger. Referring to the scale of DNS domain names, So et al. [24] estimated that the name-based FIB is at the scale of $O(10^8)$, two orders of magnitude larger than IP's. As more contents are made available through ICNs over time, the FIB size may eventually reach the level of billions of rules.

High throughput: As 100 Gbps Ethernet is approaching in practice, name-based forwarding needs to be fast enough to keep up with the wire speed. Although ICN's data plane do not forward every request due to in-network caching, a forwarding scheme should be scalable to higher throughput under the worst-case scenario.

Most existing work in name-based forwarding [18] [31] [23] [24] assumes URL-like hierarchical names, i.e., multiple name components delimited by a slash ('/'). The ruleset is often stored in a hash table, and to find the longest prefix match for a key, the key's prefixes of different length are looked up against the hash table. This approach works only for the particular name structure and often results in large hash tables due to the storage of redundant information in the ruleset. Moreover, its efficiency depends on the length of the names, making it difficult to scale to long names.

In this work, we have intentionally avoided making strong assumptions about the precise features of future namespaces. As we will see, flat and heavily hierarchical namespaces each admit opportunities to engineer feasible one-off optimizations of our core ideas. We will discuss these, but our primary aim is to design and evaluate an approach that will work for large future namespace, regardless of their particular characteristics.

In this paper we propose a binary Patricia trie based approach to name-base forwarding. Our methods are driven by two key insights. First, we show that, represented by binary strings, a name-based FIB can be notably compressed by a trie data structure to fit within contemporary fast memory of a line card, even with several million names. Second, we show that it is possible to design and optimize a trie-based data structure whose size is dependent upon the number of rules in a ruleset, rather than the length of the rules.

Our discussion considers several cross-cutting ideas, from algorithm design to namespace characteristic to forwarding methodology. To clarify the overall presentation, the paper is organized to discuss contributions in sequence, as follows.

- We first present the tokenized binary Patricia trie as an efficient data structure to store name-based forwarding tables (Section 3).
- To support FIBs at the scale of billions, we introduce a novel forwarding method, *speculative forwarding*, and its specific data plane (Section 4). Speculative forwarding trades transmission bandwidth for reduced memory size by doing longest prefix classification (LPC) instead of LPM.
- A data structure, *dual binary Patricia*, is described to realize speculative forwarding. Its size is dependent only upon the number of rules rather than the length of the rules. Further analysis and optimizations are also presented in Section 5.

To evaluate our designs, we experimentally validate the memory requirements and performance over namespaces ranging from 1 million to 1 billion rules. The smaller rulesets are real-world ones drawn from today's Internet; the larger ones are synthetically generated with similar characteristics to the real-world datasets.

The results show that only 5.58 MiB[1] memory is needed for a 1 M ruleset, and 7.32 GiB for a one-billion ruleset, about an order of magnitude improvement over the state-of-the-art solutions. Our SRAM- and DRAM-based solutions are estimated to be able to achieve throughput of 284 Gbps and 62 Gbps respectively.

2. DESIGN RATIONALE

Scalable packet forwarding has two fundamental requirements: fast FIB lookup and small memory footprint, and these two requirements are highly related. The main contributor to FIB lookup time is memory access time, which depends on the type of memory used to store the FIB. TCAM has a search time of 2.7 ns [26] for one lookup, while SRAM can achieve each access time of 0.47 ns [27] but one lookup requires multiple memory accesses. These two types of fast memories are commonly used in a line card for FIB lookup, but their sizes are limited. TCAM is available in no more than 10 MiB and SRAM can be available up to 135 MiB. DRAM, on the other hand, has a read latency of 50 ns and access time of 3 ns, but can be available in tens of GiB. It is usually used for packet buffer in line cards.

While IP-based FIBs are stored in data structures that can fit in TCAM or SRAM for fast lookup, it is a challenge to name-based FIB. Existing solutions, such as [23] [24] [31], would cost hundreds of MiB to store even a few million name prefixes. Therefore they use DRAM to store FIB and rely on massive parallel processing to speed up lookup operations.

Our design goal is to have a compact data structure so that FIBs with a few million entries can still fit in SRAM for fast lookup, and when the FIB grows to hundreds of million or even a billion entries in the future, its memory footprint is still scalable in that it only depends on the number of entries rather than the length of names, which would allow effective and efficient parallel lookups to provide high and scalable performance.

To make the FIB data structure compact, we need to avoid storing redundant information. Existing hash-table based work incurs large memory footprint because they store all the rules and there're significant redundant information across different rules. We propose to use **binary Patricia**

[1]Mi is *mebi*, multiplied by a power of 1024, while M is *mega* with a power of 1000. We also use k (*kilo*), Ki (*kibi*), G (*giga*), and Gi (*gibi*) in this paper.

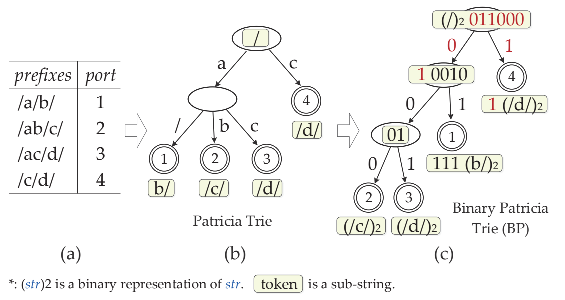

prefixes | port
/a/b/ 1
/ab/c/ 2
/ac/d/ 3
/c/d/ 4
Patricia Trie
Binary Patricia Trie (BP)
(a) (b) (c)
*: (str)2 is a binary representation of str. token is a sub-string.

Figure 1: A Patricia and Binary Patricia trie: (a) A small set of prefixes; (b) Patricia representation; (c) binary Patricia representation.

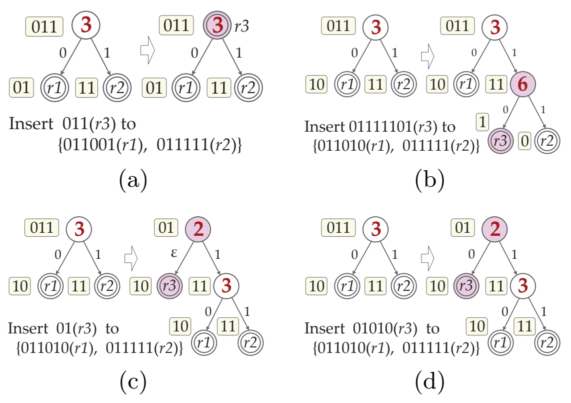

Insert 011(r3) to {011001(r1), 011111(r2)}
Insert 01111101(r3) to {011010(r1), 011111(r2)}
Insert 01(r3) to {011010(r1), 011111(r2)}
Insert 01010(r3) to {011010(r1), 011111(r2)}
(a) (b) (c) (d)

Figure 2: Four possible cases in key insertion.

trie data structure to minimize the redundant information stored. Evaluations showed that this is enough to compress FIBs of a few million entries to fit in tens of MiB of SRAM, the first part of our goal.

To achieve the second part of our goal, which is to scale to billions of FIB entries, we further reduce the data structure to store only the differentiating bits between different rules, not the rules themselves. The result is a minimal data structure that can scale ten times better than existing work, and a corresponding forwarding behavior that we termed "speculative forwarding". While speculative forwarding may cost extra bandwidth when the packet carries a name which doesn't match any prefix in the FIB, we think this is a favorable tradeoff between bandwidth and memory used to support billions of name prefixes.

3. BINARY PATRICIA FOR NAME-BASED LPM

3.1 Binary Patricia for Small Tables

Patricia trie [13], also known as radix tree, is a space-optimized general trie structure. It is generated from a character-based prefix trie by merging every single-child node with its child. The internal nodes in Patricia trie are used to branch by discrimination characters while leaf nodes contain the match information, i.e., the prefix. *Binary Patricia* is a commonly used variant, which treats prefixes as binary strings in building the trie.

In a binary Patricia, a key is represented as $k = b_0, ..., b_{l-1}$ where $k[i] = b_i \in \{0, 1\}, 0 \leq i < |k|$, $|k| = l$ representing the width of k. A key k' is said to be a proper (strict) prefix of key k, denoted by $k' \prec k$, iff $|k'| \prec |k|$ and $k'[i] = k[i], 0 \leq i < |k'|$. If two keys are identical, they are considered as one key. A binary Patricia is a full binary tree in which every non-leaf node has exact two children.

Fig.1 illustrates a character-based Patricia and a binary Patricia built from the same FIB. In a character-based Patricia, each internal node may have at most 256 children, branched by octet characters. From the root to a leaf, all tokens stored in the nodes and characters along the arrows together form a complete prefix. The delimiter ('/') is treated as a character as well. To build a binary Patricia, we use the ASCII code to represent prefixes. Other coding schemes can be used too. Each node has at most two children, branched by a bit of 0 and 1. All bits in the nodes and along the arrows from the root to a leaf together form a complete prefix. This binary Patricia is also called as *tokenized binary Patricia* in this paper, since a prefix is divided to a sequence of tokens. Especially, the branched 0 and 1 in an internal node are part of the sequence.

While the use of binary Patricia in IP lookup dates back to BSD kernel implementation [22], using it for name-based forwarding is novel; most existing work store prefixes in hash tables. We choose binary Patricia for three reasons. First, the binary representation provides the most opportunity to compress shared parts between different prefixes. In the widely used component-based approach, only common components can be compressed; in binary Patricia, arbitrary common bit sequences can be compressed. Second, treating names as bit strings means that we do not assume any property of names, therefore our design can be applied to any name scheme. Third, since names are encoded as binary strings on the wire during transmission, we can directly use the name's wire format in table lookup and forwarding without parsing the name. The component-based approach, however, would need to parse the wire format to figure out the component boundaries before table lookup, which can incur significant processing overhead. Therefore, although component-based approach is more intuitive, binary Patricia is more generic and costs less in memory and processing.

3.2 Binary Patricia Details

A binary Patricia is built by inserting FIB entries one by one into an initial empty tree. Alg.1 details the *key-insertion* algorithm. More specifically, a common prefix is first found by comparing the token which is stored in the root with the key, k. There are three terminated cases: if this common prefix a) equals to both k and the *token* (Fig.2(a)), b) does not equal to either of them (Fig.2(d)), and c) equals to k (Fig.2(c)). If the common prefix equals to the *token* but not k, another iteration is needed to search lower level nodes of the trie with the remaining sequence of k (Fig.2(b)). We omit some details, such as labeling bit positions and creating new nodes, which are described in [13] [15]. The structure of a binary Patricia is independent of key insertion order, i.e., there is a unique trie for any given set of keys.

Removing a key from a binary Patricia requires a key query to locate its leaf node first, then deletes the leaf node and recursively merges all single-child nodes with their children along the path back. Updating a key is the combination of deleting the old key and inserting the new one.

Algorithm 1 Key Insertion Algorithm in *BP*

```
Input:
    t: binary Patricia, static, initial is t[0]
    k: binary digits of a key
Output:
    t: binary Patricia with k
 1: cp=find_common(t[0].token, k); i=0;
 2: while TRUE do
 3:   if cp==t[i].token and cp==k then
 4:     mark t[i] with k's next-hop {as Fig.2(a)}
 5:   else if cp==t[i].token then
 6:     i=t[i].next[k[|cp|]]
 7:     cp=find_common(t[i].token,k[|cp|+1:]){Fig.2(b)}
 8:     continue
 9:   else if cp==k then
10:     add two nodes with an $\varepsilon$ label {as Fig.2(c)}
11:   else
12:     add two nodes {as Fig.2(d)}
13:   end if
14:   break
15: end while
```

Suppose l_m is the maximum length in a set with n keys, the construction time and space complexity of a binary Patricia is $O(n \times l_m)$ and $O(n)$ respectively. The time complexity of update and search is $O(l_m)$.

4. SPECULATIVE FORWARDING

Although binary Patricia is more compact, its scalability is still similar to other solutions in that its size depends on the length of names. The memory footprint of a binary Patricia consists of two parts: the memory for the trie structure, which is really the differentiating bits between prefixes, and the memory for all the tokens. Note that the binaries of 0 and 1 on the arrows in Fig.1 are counted as part of the trie, not the tokens. Given it is a full trie, a binary Patricia trie has *n-1* internal nodes and *n* leaf nodes for *n* keys. The memory for this structure is independent to the length of the names. The tokens, however, are determined by the length of names. The longer the names, the more memory needed by storing them. A FIB with a few hundreds of thousands of long prefixes may well exceed the capacity of SRAM, let alone FIBs with billions of entries. This is a unique problem arisen in name-based forwarding but not in IP.

The idea to solve this problem, under the context of binary Patricia, is to remove the tokens from the Patricia. This will leave only the differentiating bits in the data structure and make it independent to the length of name prefixes. If this works, it will fundamentally change the scalability property. It will lead to small memory footprint, faster processing, and resilience to attacks that exploit very long names. However, removing the tokens from binary Patricia also changes the lookup behaviour from longest-prefix match (LPM) to longest-prefix classification (LPC). From the perspective of matching, LPM generally consists of two steps: longest-prefix classification and verification. LPC is a step to filter some candidate rules, and an additional verification is required to affirm the right one.

In this section, we will discuss the data-plane forwarding behavior which removes verification from LPM to LPC and the corresponding memory-efficient data structure.

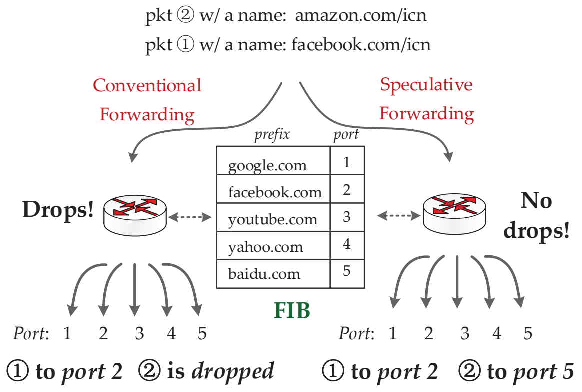


Figure 3: Speculative forwarding vs. conventional forwarding: an example.

4.1 Speculative Data Plane

We present *speculative forwarding*, which is defined as a forwarding policy that relays packets by LPC instead of LPM. In conventional LPM lookup, if there is a match, the packet will be forwarded to the corresponding nexthop, otherwise it will be dropped. LPC lookup guarantees that if there is a match, the packet will be forwarded to the same nexthop that LPM would use, but if there is no match, the packet is still forwarded.

4.1.1 Forwarding Behaviors

For example, in Fig.3, assume the FIB has already been populated by five prefixes. When the first packet with name *facebook.com/icn* arrives, both conventional forwarding and speculative forwarding will find the same matching prefix in the FIB and forward the packet out via port 2. This is because both data structures contain the same trie structure, therefore the lookup will end up with the same leaf node.

When the second packet with a name *amazon.com/icn* arrives, its name will be used to look up against the trie structure and end up with a leaf node. In conventional forwarding, the leaf node contains the prefix tokens, which will be used to verify whether the FIB entry indeed is a prefix to the packet name, and in this case the verification fails and the packet is dropped. However, in speculative forwarding, since the tokens have been removed from storing, there is no verification step, thus the packet will be forwarded out.

Speculative forwarding can guarantee correct known-prefix lookup and forwarding, but will *wrongly* relay (instead of drop) unknown-prefix packets. In return, it can reduce the memory footprint significantly by not storing rules for verification. Two issues should be further resolved: a) how to design a complete data plane which can work well with speculative forwarding, and b) how to design a data structure to support LPC in all cases for speculative forwarding.

4.1.2 Speculative Data Plane

We propose to use speculative forwarding in default-free zone (DFZ) and conventional forwarding in places with default routes, for the following reasons. DFZ routers are those need to support all global name prefixes and need to forward at high wire speed. Thus they can benefit the most from using speculative forwarding that gives smaller routing table and faster lookup. For routers in edge networks, they need

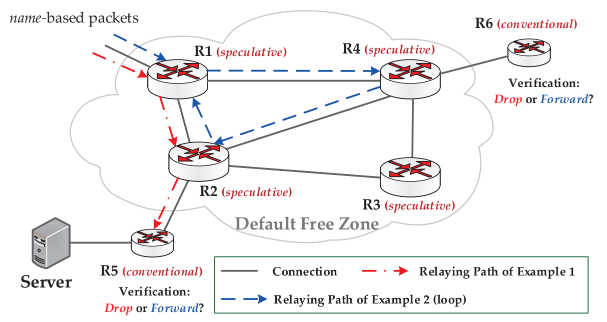


Figure 4: Speculative data plane: an example.

to support much fewer number of prefixes when using the default route. They probably can afford storing all the tokens in the binary Patricia, and they can serve as termination points for those packets wrongly forwarded by speculative forwarding. The combination of edge routers running conventional forwarding and core routers running speculative forwarding forms our global data plane.

We use two examples to explain the forwarding behaviors in this data plane. Fig.4 illustrates a network where there are 6 routers, 4 of which (R1 through R4) are in DFZ running speculative forwarding and 2 (R5, R6) in non-DFZ running conventional forwarding.

4.1.3 *Example 1: Packet Forwarding*

When a packet from edge networks arrives at R1, R1 performs a FIB lookup by LPC, which will always return nexthop such as R2. R2 does the same thing and forward the packet to R5. Now the packet has left DFZ and arrives at R5, which will perform LPM lookup. There are two possible outcomes. If the packet name matches a known prefix, R5 will forward it and we know that R1 and R2 must also have made the correct forwarding decisions to this packet. Otherwise the packet name doesn't have any match in the FIB, or strictly speaking, it matches R5's default route which is where the packet comes from. R5 will then drop the packet. The packet should have been dropped when it arrives at R1 if R1 runs conventional routing, but now it is dropped some time later at R5 for the purpose of reducing memory footprint at all core routers in DFZ.

4.1.4 *Example 2: Loop Handling*

Forwarding loops can happen with speculative forwarding. In Fig.4, when an unknown-prefix packet arrives at R1 from edge networks, it could be forwarded to R4, then R2, and R2 send it back to R1, forming a loop. Whether it happens or not depends on the actual content of the FIB, the network topology, and the packet's name. While IP's data plane doesn't handle loops and only relies on TTL exhaustion as the last resort, ICN architectures, especially NDN, employs stateful data plane, which can actually detect and drop looped packets. For example, if this is an NDN network, R1 would store forwarded Interests in the Pending Interest Table (PIT), whose purpose is to guide Data packets back to the data consumer. If the Interest packet somehow loops back to R1, R1 will be able to tell this is a looped packet by looking it up in PIT and as a result drop the packet. The packet doesn't complete the loop again and again until TTL exhaustion. Therefore the damage of looping packets, if it happens, is much smaller than that in IP networks.

4.1.5 *Feedback from the Edge*

A feedback mechanism can be applied to a speculative data plane to reduce the overhead of wrongly forwarded packets. In a stateful data plane like NDN's, unknown-prefix Interest packets are stored in PIT, and will be removed after a timeout since it will not bring back any data. Optionally the edge router that dropped the unknown-prefix packet can send back a feedback packet (such as a NACK packet [20]), which will explicitly inform all nodes along the path to delete related pending Interest entries, and may be cached for a while to suppress other interests under the same unknown prefix. This mechanism may immediately reduce the memory overhead of unknown-prefix packets.

4.1.6 *Prefixes Over Different Routers*

Speculative data plane uses both traditional forwarding and speculative forwarding, and the combination is flexible and configurable. For core routers in DFZ, they may choose to exploit speculative forwarding to reduce its memory footprint of prefixes, or use traditional forwarding when resources are not in count. For other routers, they are configured as traditional forwarding.

For a unknown-prefix named packet whose name contains not any prefix in FIB, it will be forwarded, matched and eliminated by routers in traditional data plane. It may also be relayed and eliminated by routers in speculative data plane. The only difference is that the routers in DFZ can only relay but not drop those packets, because speculative forwarding can only filter but not verify prefixes of names.

4.2 Data Structure: Dual Binary Patricia

In this section, we describe *dual binary Patricia* to support LPC in a speculative data plane. (*dual Patricia* and *DuBP* for short in the rest of text and in the figures)

Generally speaking, a given FIB table (S) can be classified into two subsets. One (F) consists of all flat name prefixes (i.e., no one is a proper prefix of another one), and the other (P) consists of all entries where their names are covered by some entries in the flat subset. Thus, S=F+P. As its name implies, dual Patricia is a data structure that consists of two different styles of Patricia: a (tokenized) binary Patricia for subset P and a speculative binary Patricia for subset F.

4.2.1 *Speculative Binary Patricia*

Speculative binary Patricia (denoted as *sBP*) is a variant binary Patricia which is proposed by our work. It does not store any token in internal or leaf nodes. Instead, every internal node stores a value representing a bit position, namely a *discrimination bit position*. Every lookup in an internal node is branched by identifying the digit of that position in the lookup name

Fig.5(b) shows an example of speculative binary Patricia for FIB in Fig.5(a). The root node is labeled with *p:15*. It means that the root will check the *15th* bit of the lookup name. If that bit is 0 in the name, go to the left child, otherwise go to the right. The whole trie does not store any token to verify correctness of other bit positions except discrimination bits.

Speculative Patricia has a similar building process to the binary Patricia described in Section.3.

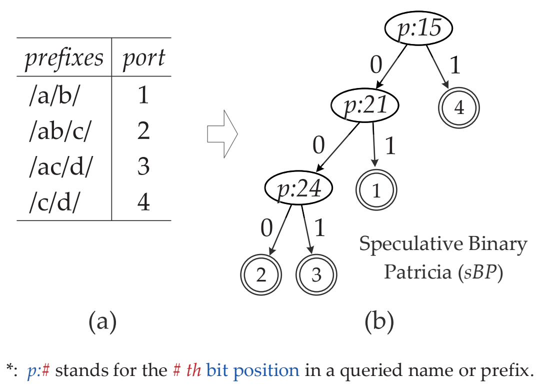

prefixes
port
/a/b/
1
/ab/c/
2
/ac/d/
3
/c/d/
4
p:15
p:21
p:24
0
1
Speculative Binary Patricia (sBP)
(a)
(b)
*: p:# stands for the # th bit position in a queried name or prefix.

Figure 5: Speculative binary Patricia trie: (a) A small set of prefixes; (b) speculative binary Patricia representation.

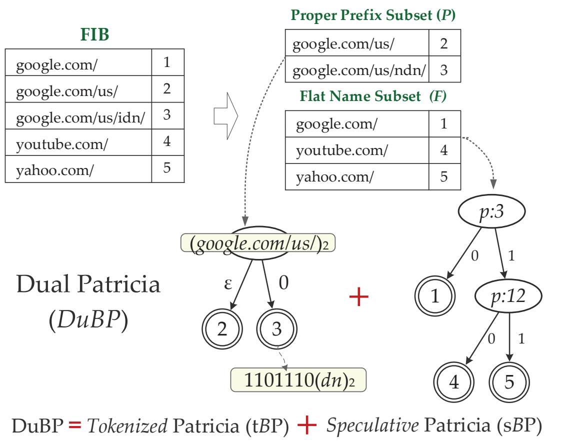


Figure 6: Dual Patricia: an example.

4.2.2 *Dual Binary Patricia*

Fig.6 illustrates a dual Patricia built for a FIB with 5 entries. The prefixes are classified to (only) two subsets. For the subset P with 2 prefixes, we build a tokenized Patricia to perform LPM. For the subset F with 3 flat entries, we build a speculative Patricia to perform LPC.

For an incoming packet, name lookup has two steps. First, the name is given to the tokenized Patricia to preform LPM. If there is no match found, the name is then given to the speculative Patricia to perform LPC among flat names. S-ince speculative Patricia always returns to a port for a name, the packet will be relayed to that port. Thus dual Patricia overall provides longest prefix classification for speculative forwarding. If a name is related to multiple ports, it is the duty of forwarding strategy to choose one of them.

4.2.3 *Discussions about Dual Patricia*

The reason that two different Patricia are required is because in the case of a prefix covered by another, such as *google.com/* and *google.com/us/*, a speculative binary Patricia cannot distinguish them apart. Since there are no tokens available in terms of LPC and *google.com/* is along the path to *google.com/us/* in Patricia, a given name cannot be verified whether it matches the internal node which represents *google.com/* or it should go further to match the leaf node which represents *google.com/us/*. Thus, we distinguish proper prefixes from flat names and build two types of Patricia for them separately.

This design of data structure actually fits well with the design of data plane, where edge routers perform LPM and core routers perform LPC. Edge routers have default route, which covers all other prefixes. Therefore by our trie construction procedure, the entire dual Patricia becomes a binary tokenized Patricia, and the other speculative trie is empty. This naturally leads to LPM for any router that has a default route. Actually, dual Patricia is a general framework to consist of both traditional forwarding with LPM and speculative forwarding with LPC. The configuration is on user's choice.

Binary Patricia, speculative Patricia and dual Patricia (built from the first two) are slightly different variants of Patricia. Since they all have identical trie structure to the traditional Patricia, the analysis and optimization in next Section 5 are applicable to all of them.

5. ANALYSIS AND OPTIMIZATIONS

Two quantities of a Patricia are of special interests: the *depth of a leaf* (search time) and the *length of all tokens* (token memory). This section gives mathematical analysis of the average complexity and optimizations for Patricia. Besides, optimal and practical approaches to place Patricia in memory is also discussed.

5.1 Performance Analysis and Optimization

Different from general key searching in a tree, almost all of the searches in name-based lookup are *successful* search that can lead to leaf nodes. For a Patricia with n keys, the average depth of all leaf nodes, ED_n, is [14]

$$ED_n = \frac{1}{h_1}[lnn + \rho] \tag{1}$$

where $ln(\cdot)$ denotes the natural logarithm. Here,

$h_k = (-1)^k \sum_{i=0}^{1} p_i lnp_i$ and $\bar{h_k} = (-1)^k \sum_{i=0}^{1} p_i(1 - lnp_i)$

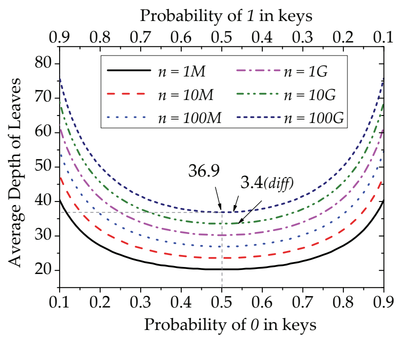
while p_0, p_1 are the probability of digits 0 and 1 in the set. $\rho \overset{\text{def}}{=} \gamma - \bar{h}_1 + h_2/(2h_1)$, where $\gamma = 0.5772$[2].

If the probabilities of digits 0 and 1 are equal in keys, the average depth of Patricia is $logn + 0.3327$. Considering the broad range of probabilities and the scale of the key set, the average depths for different cases are shown in Fig.7(a). On average, there are only about **three** additional steps deeper for a key set that is 10x larger. We can conclude that Patricia scales well and maintains a nearly constant search time. The results in practice are similar.

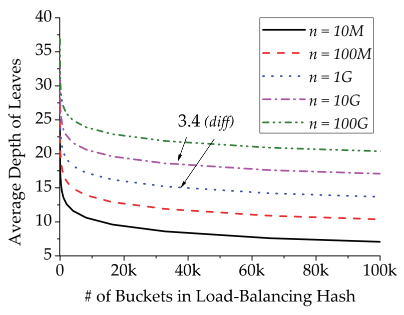
In terms of data structure, reducing the average depth of leaves is always an important step to improve searching performance. It is a classic issue that has already been heavily studied [15]. We utilize a practical scheme, namely load-balancing hash [30], by importing a hash as a load balancer to shorten the depth and therefore improve the search performance in Patricia.

A FIB rulesets can be split into many subsets by hashing, and Patricia can be built individually in each hash bucket. The hybrid scheme of using Patricia and load balancing hash

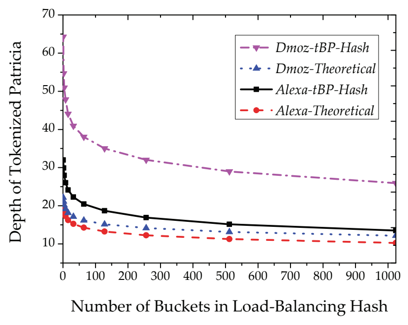
[2] γ is a *Euler* constant



(a)



(b)



(c)

Figure 7: (a) Theoretically average depth of leaves in various possibility of 0 and 1; (b) Theoretical (c) empirical results about average depth using hybrid method with load-balancing hash.

is proven to be efficient to reduce the average depth. Fig.7(b) shows the theoretical average depth of a dual Patricia after importing a load-balancing hash on millions to billions keys. It is important to note that the average depth is a kind of long tail relationship to the number of hash buckets. Smaller hash function is good enough to reduce the depth, even for super-large or super-unbalancing sets.

Besides theoretical analysis, Fig.7(c) presents some empirical results of two real-world data sets with 1 million and 3.7 million rules in Table 1. The one-million set came from Alexa [16], which contains the top 1 M most popular sites. The 3.7 million set was collected from Dmoz [17], which is the largest and most comprehensive directory of the Web. For Alexa set, the depth decreases from 32 to 14; for Dmoz, from 65 to 26 with only hundreds of buckets. We conclude that 1 K buckets in a hash table are good enough to get practical minimum depth.

In practice, the depth is not the only parameter of performance because of memory hierarchical system, especially several levels of cache. Searching a trie could be extremely fast if paths and nodes are already in cache. To provide a nearly constant lookup time, our major design goal is to make the data structure small enough or flexible enough to reside in fast memory (SRAM) or cache in DRAM, then to reduce the depth of leaf nodes.

5.2 Memory Layout and Optimization

The memory layout which places a data structure in memory is as important and critical as the data structure in practice. It compacts memory footprint and impacts the lookup speed. An efficient memory layout should balance the flexibility between random access for high speed and the effectiveness of memory usage.

Basically, binary Patricia is a trie data structure. The memory layout issue is to efficiently *link* all nodes and *store* tokens. Tokenized and speculative Patricia are full binary trie, and every internal node has two children. An obvious memory layout in software implementation is to maintain two pointers from an internal node to its children. For example, suppose we have 1 billion keys, i.e., 1 billion internal nodes, a Patricia should maintain 2 billion pointers. On a 64-bit platform, only maintaining those pointers would consume 16 GiB memory.

Our work introduces a concise memory layout, namely *single-address memory layout*, which can use only one pointer to link an internal nodes with its two children in Patricia. The motivation comes from the fact that Patricia is a full, but not a complete, binary tree. So we have to store pointers (or addresses) from internal nodes to their children, but we may need only one, not two, pointer since there are always two children and they can be put together in memory.

In an internal node, a single address is used to locate the memory of its left child as a base address, and an offset is used to calculate the right child by adding to the base. Tokens can be optionally stored within the trie structures. For 1 billion FIB entries, the single-address layout can easily save about 8 GiB memory size. Similarly, a leaf node stores token related fields and its next-hop information of the corresponding FIB entry. An additional bit is also needed to distinguish leaf nodes and internal nodes for different data structures [15].

Although this single address memory layout may not be the most memory efficient solution for Patricia, it remains a simple way to point to children of nodes and a flexible placement of nodes in memory. Each pair of children can be stored individually, and they are small enough for a cache system to fetch and keep as a fragment of the whole tree. These merits improve the performance by using cache system in software implementation, and enable the hardware implementation to gain high performance by applying on-chip memory.

Table 1: The data set configurations

data set	# of keys	% of p.p.	avg.len	alphabet
Alexa	999,973	1.03	15.5 B	readable
Dmoz	3,711,540	3.85	27.3 B	readable
Gen100	100 M	5-50/+5	37.6 B	readable
Gen300	300 M	5-50/+5	37.6 B	random
Gen1000	1 G	5-50/+5	37.6 B	random

6. EXPERIMENTAL RESULTS

6.1 Methodology and Setup

To evaluate the above approaches, we collected five data sets, shown in Table 1, from one million to one billion keys to model realistic routing tables in different scales. Two of them are real-world sets, as mentioned in Section 5. We gen-

Table 3: The memory devices in a line card

device	single-chip	#	total size	lookup
TCAM [26]	2.5 MiB	4	10 MiB	2.7 ns
SRAM [27]	33.75 MiB	4	135 MiB	0.47 ns
DRAM	4 GiB	4-8	16-32 GiB	50 ns

erated another three sets. Gen100 set was built on a human readable character set by combining three random English words in the dictionary and one of 271 TLDs from INNA. The other two sets were generated by random bytes and TLDs. To evaluate the cases of highly hierarchical namespace, all generated sets has 5% to 50% of proper prefixes (p.p.) with a step of 5%, where 5% is set as default. We suppose that every router can support the maximum of 256 next hops. If experimental results come from any random related circumstance, they are the average of 10 tries.

6.2 Memory Footprint and Comparison

To normalize results from different scale data sets, we use *memory per million keys* as a unit. In some cases, we also use absolute values directly. For fairness, the single-address memory layout is used as default.

Fig.8(a) presents the memory size of a binary (tokenized) Patricia in conventional forwarding with detailed trie and token memory for five data sets. The memory cost per million keys remains stable in different scales from millions to billions. The slight differences come from the token memory size, which is related to the data sets. For Alexa and Dmoz, both token memory account for about 60% of all memory, which gives us an idea of real-world namespaces.

Fig.8(b) gives the memory cost of a dual Patricia for speculative forwarding. Compared to Fig.8(a), token memory is reduced heavily because no tokens in speculative Patricia is stored for flat names. Because of proper prefixes in the subset P, dual Patricia also consumes some memory for tokens.

All results with detailed values are shown in Table.2. It shows that dual Patricia in speculative forwarding consumes only about 40% of the memory of binary Patricia, achieving 2.5 times memory efficiency. Even for tokenized Patricia, 64.80 MiB for 3.7 million DMOZ set also gains more than 5 times better than other approaches in the literatures, as shown in Table 4. In this table, all memory and performance results of other approaches are extracted from those papers. For memory footprint, the comparison is fair enough, however, for performance, different approaches may build on different platforms, so the results are just for reference.

In the real-world sets in Table 1, there is a very limited proportion of proper prefix keys. To evaluate the cases of highly hierarchical namespace, we extended three generated sets to have from 5% to 50% of proper prefixes, with a step size of 5%. Fig.8(c) gives the memory footprint for dual Patricia. The results indicate a nearly linear scalable trend.

Table 3 lists different configurations of memory devices provided in a line card. Compared to the results in Table.2 (the bottom two lines), it can be calculated that a 135 MiB SRAM-based solution which consists four channels of 33.75 MiB single-chip SRAMs can host 7.71 M FIB entries in conventional forwarding and 16.11 M FIB entries in speculative forwarding. Because 10 M FIB entries are expected as a practical target [23], our approach successful builds the target within fast memory of a contemporary line card.

6.3 Performance and Comparison

6.3.1 Average Depth

We have addressed the average depth of Patricia in Section 5 from theoretical analysis down to optimizations in Fig.7(a), Fig.7(b) and Fig.7(c). It shows that, for names from millions to billions, an average depth is 15 to 20 in both theoretical and practical cases.

Patricia searches a key step by step from the root to leaf, so we assume that there is one memory access for each step deeper. Therefore, we use 15 memory accesses for millions names and 20 for billions to estimate the real performance in the worse scenarios. Actually, cache and other techniques can greatly reduce the requirement of one memory access per step. Although dual Patricia has two separate Patricia, it can be regarded as one Patricia with 15 to 20 steps because the binary Patricia for proper prefixes, tokenized Patricia, hosts very small number of FIB names according to the characteristics of real-world data sets.

6.3.2 Performance Estimation and Comparison

We estimate SRAM-based performance by using parameters in Table 3, in which each memory access to QDR SRAM consumes 0.47 *ns*. For millions of names, a complete key lookup requires 7.05 *ns*, thus 142 million searches per second (MSPS).

For billions of names, DRAM-based solution with massive parallel computing is the only choice. Different from general computer systems, a line card already supports parallel and low-level hardware specific programming for fast speed. Optimizing a tree search in DRAM has been studied before [29], which exploits pipeline and multiple DRAM channels to achieve near one DRAM access performance for a complete key query. We desire that approach and evaluate our performance based on it. According to Table 3, DRAM-based performance may achieve 20 MSPS.

It is complicated to estimate ICN throughput that an approach can support, because only Interest packets are delivered to FIB lookup in NDN architecture and in-network caching can reduce the lookup requirements. We use a similar estimation approach used by So et al. [23] that each lookup handles 256-byte packet in average. Accordingly, our SRAM-based solution can provide a throughput of 284 Gbps, while DRAM-based solution can achieve 62 Gbps. Finally, our results are compared with other approaches in Table 4.

We have memory and performance results in Table 4 for real and generated data sets. Only memory results are evaluated in real systems. The performance values are all estimated, just for reference, according to hardware behaviour and our experiences on them. The future work is to extend our implementation to various hardware systems to further evaluate our approach.

7. CONCLUSIONS

Named-based forwarding poses unique challenges due to complex name structures, unbounded namespace, and high-speed requirement. We propose to use binary Patricia as the basis for compact name-based FIBs, and further improves its scalability by introducing speculative forwarding as a concept to build a novel forwarding plane and dual Patricia as a data structure to carry longest prefix classification rather than longest prefix match. Evaluations have shown more than an order of magnitude reduction in memory footprint

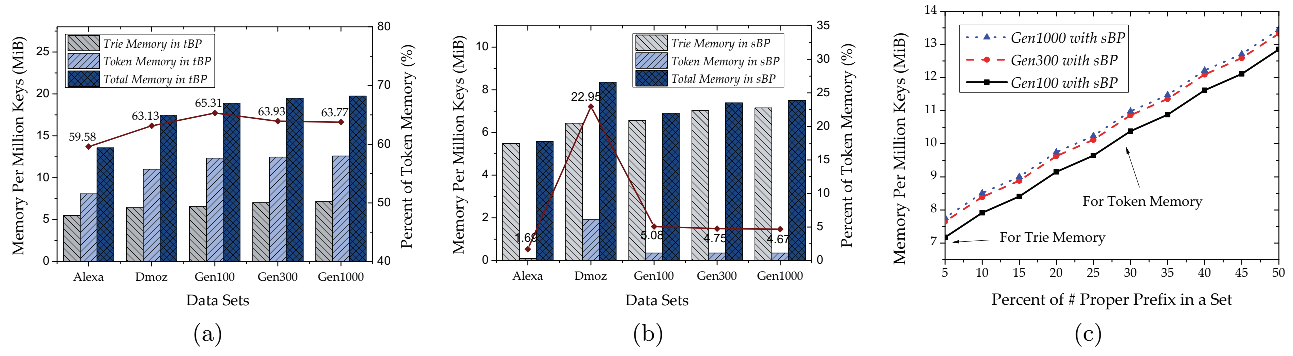


Figure 8: Memory size in a (a) tokenized Patricia, (b) dual Patricia for speculative forwarding,(c) different proper prefix sets.

Table 2: The memory cost of different data structures with one-address memory layout

Data Set	Tokenized Patricia(***tBP***)				Dual Patricia(***DuBP***)				*DuBP/tBP*	*Memory* Efficiency
	Trie	Token	Total	Devices	Trie	Token	Total	Devices		
Alexa (**MiB**)	5.48	8.08	**13.56**	SRAM	5.48	0.10	**5.58**	SRAM	41.1%	2.43
Dmoz (**MiB**)	23.89	40.91	**64.80**	SRAM	23.89	7.12	**31.01**	SRAM	47.8%	2.09
Gen100 (**GiB**)	0.64	1.21	1.85	DRAM	0.64	0.034	0.67	DRAM	36.2%	2.89
Gen300 (**GiB**)	2.06	3.62	5.68	DRAM	2.06	0.103	2.16	DRAM	38.0%	2.63
Gen1000 (**GiB**)	6.98	12.05	**19.03**	DRAM	6.98	0.342	**7.32**	DRAM	38.5%	2.60
7.71 M Rules		← 135 MiB		SRAM						
16.11 M Rules				SRAM				← 135 MiB		

than state-of-the-art solutions. Thus several million of FIB rules can fit into contemporary SRAM in commercial line cards to achieve the wire speed. Future work includes further investigation on speculative data plane, and extending the binary Patricia to other name-based functionality such as content stores and pending interest tables.

8. ACKNOWLEDGMENTS

This work is supported by the National Natural Science Foundation of China under grant No.61272510, the National Science Foundation under grants CNS-1040643, CNS-1345282, CNS-1455794 and CNS-1513505, and Jiangsu Future Networks Innovation Institute of China under the Prospective Research Project on Future Networks No. BY2013095-1-09. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect those of the funding sources.

9. REFERENCES

[1] Ahlgren, Bengt, Christian Dannewitz, Claudio Imbrenda, Dirk Kutscher, and Börje Ohlman. "A survey of information-centric networking." Communications Magazine, IEEE 50, no. 7 (2012): 26-36.

[2] Ahlgren, B., M. D?ambrosio, C. Dannewitz, A. Eriksson, J. Golic, B. Grönvall, D. Horne et al. "Second netinf architecture description." 4WARD EU FP7 Project, Deliverable D-6.2 v2.0 (2010).

[3] Koponen, Teemu, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. "A data-oriented (and beyond) network architecture." In ACM SIGCOMM Computer Communication Review, vol. 37, no. 4, 2007.

[4] Zhang, Lixia, Deborah Estrin, Jeffrey Burke, Van Jacobson, James D. Thornton, Diana K. Smetters, Beichuan Zhang et al. "Named data networking (ndn) project." Technical Report NDN-0001 (2010).

[5] Yuan, Haowei, Tian Song, and Patrick Crowley. "Scalable NDN forwarding: Concepts, issues and principles." In Computer Communications and Networks (ICCCN), 2012 21st International Conference on, pp. 1-9. IEEE, 2012.

[6] Degermark, Mikael, Andrej Brodnik, Svante Carlsson, and Stephen Pink. "Small forwarding tables for fast routing lookups." Vol. 27, no. 4. ACM, 1997.

[7] Eatherton, William N. "Hardware-based internet protocol prefix lookups." Master's thesis, Washington University, 1999.

[8] Jiang, Weirong, and Viktor K. Prasanna. "A memory-balanced linear pipeline architecture for trie-based IP lookup." In High-Performance Interconnects, 2007. HOTI 2007. 15th Annual IEEE Symposium on, pp. 83-90. IEEE, 2007.

[9] Eatherton, Will, George Varghese, and Zubin Dittia. "Tree bitmap: hardware/software IP lookups with incremental updates." ACM SIGCOMM Computer Communication Review 34, no. 2 (2004): 97-122.

[10] Srinivasan, Venkatachary, and George Varghese. "Faster IP lookups using controlled prefix expansion." In ACM SIGMETRICS Performance Evaluation Review, vol. 26, no. 1, pp. 1-10. ACM, 1998.

[11] Smith, P.. "Weekly routing table report." http://thyme.rand.apnic.net.

[12] Doeringer, Willibald, Günter Karjoth, and Mehdi Nassehi. "Routing on longest-matching prefixes."

Table 4: Comparison with other approaches

Method Names	Alexa Mem	Dmoz Mem	Performance	Throughput	Scalability	Comments
Compressed Trie [25]	N/A	400.8 MiB	0.15 MSPS	0.3 Gbps	N/A	
NCE [18]	79.64MiB	272.27 MiB	0.91 MSPS	1.82 Gbps	N/A	3M Dmoz rules
MATA [31](GPU)	N/A	197.6 MiB	63.52 MSPS	127 Gbps	millions	*latency* 100 us
Hash [24](GPU)	153 MiB	587 MiB	8.8 MSPS	17.6 Gbps	millions	massive parallel
Binary Patrica (SRAM)	13.56 MiB	64.80 MiB	142 MSPS	284 Gbps	$\leq$ 7.71 M	3.7M Dmoz rules
Dual Patrica (SRAM)	5.58 MiB	31.01 MiB	**142 MSPS**	**284 Gbps**	$\leq$ **16.1 M**	3.7M Dmoz rules
Dual Patrica (DRAM)	5.58 MiB	31.01 MiB	20 MSPS	62.02 Gbps	billions	w/ pipeline

IEEE/ACM Transactions on Networking (TON) 4, no. 1 (1996): 86-97.

[13] Morrison, Donald R. "PATRICIA?practical algorithm to retrieve information coded in alphanumeric." Journal of the ACM (JACM) 15, no. 4 (1968): 514-534.

[14] Szpankowski, Wojciech. "Patricia tries again revisited." Journal of the ACM (JACM) 37, no. 4 (1990): 691-711.

[15] Knuth, Donald Ervin. "The art of computer programming: sorting and searching." Vol. 3. Pearson Education, 1998.

[16] Alexa: http://www.alexa.com/.

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

[18] Wang, Yi, Keqiang He, Huichen Dai, Wei Meng, Junchen Jiang, Bin Liu, and Yan Chen. "Scalable name lookup in NDN using effective name component encoding." In Distributed Computing Systems (ICDCS), 2012 IEEE 32nd International Conference on, pp. 688-697. IEEE, 2012.

[19] Wang, Yi, Yuan Zu, Ting Zhang, Kunyang Peng, Qunfeng Dong, Bin Liu, Wei Meng et al. "Wire Speed Name Lookup: A GPU-based Approach." In NSDI, pp. 199-212. 2013.

[20] Yi, Cheng, Alexander Afanasyev, Lan Wang, Beichuan Zhang, and Lixia Zhang. "Adaptive forwarding in named data networking." ACM SIGCOMM computer communication review 42, no. 3 (2012): 62-67.

[21] Yi, Cheng, Alexander Afanasyev, Ilya Moiseenko, Lan Wang, Beichuan Zhang, and Lixia Zhang. "A case for stateful forwarding plane." Computer Communications 36, no. 7 (2013): 779-791.

[22] Sklower, Keith. "A tree-based packet routing table for Berkeley unix." In USENIX Winter, vol. 1991, pp. 93-99. 1991.

[23] So, Won, Ashok Narayanan, Dave Oran, and Yaogong Wang. "Toward fast NDN software forwarding lookup engine based on hash tables." In Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems, 2012.

[24] So, Won, Ashok Narayanan, and David Oran. "Named data networking on a router: Fast and DoS-resistant forwarding with hash tables." In Proceedings of the ninth ACM/IEEE symposium on Architectures for networking and communications systems, 2013.

[25] Shue, Craig A., and Minaxi Gupta. "Packet forwarding: Name-based vs. prefix-based." In IEEE Global Internet Symposium, pp. 73-78. 2007.

[26] Renesas TCAM: http://www.am.renesas.com/

[27] Cypress QDR SRAM: http://www.cypress.com/

[28] Sahasra Processor: http://www.broadcom.com

[29] Kumar, Sailesh, Michela Becchi, Patrick Crowley, and Jonathan Turner. "CAMP: fast and efficient IP lookup architecture." In Proceedings of the 2006 ACM/IEEE symposium on Architecture for networking and communications systems, pp. 51-60. ACM, 2006.

[30] Fagin, Ronald, Jurg Nievergelt, Nicholas Pippenger, and H. Raymond Strong. "Extendible hashing?a fast access method for dynamic files." ACM Transactions on Database Systems (TODS) 4, no. 3 (1979): 315-344.

[31] Wang, Yi, Tian Pan, Zhian Mi, Huichen Dai, Xiaoyu Guo, Ting Zhang, Bin Liu, and Qunfeng Dong. "NameFilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters." In INFOCOM, 2013 Proceedings IEEE, pp. 95-99., 2013.