Reliably Scalable Name Prefix Lookup

Haowei Yuan & Patrick Crowley
Washington University
Department of Computer Science & Engineering
St. Louis, Missouri 63130
{hyuan, pcrowley}@wustl.edu

ABSTRACT

Name prefix lookup is a core building block of information-centric networking (ICN). In ICN hierarchical naming schemes, each packet has a name that consists of multiple variable-length name components, and packets are forwarded based on longest name prefix matching (LNPM). LNPM is challenging because names are longer than IP addresses and the namespace is unbounded. Recently proposed solutions have shown encouraging performance, however, most are optimized for or evaluated with a limited number of URL datasets that may not fully characterize the forwarding information base (FIB). What's more, the worst-case scenarios of several schemes require $O(k)$ string lookups, where k is the number of components in each prefix. Thus, the sustained performance of existing solutions is not guaranteed.

In this paper, we present a LNPM design based on the *binary search of hash tables*, which was originally proposed for IP lookup. With this design, the worst-case number of string lookups is $O(log(k))$ for prefixes with up to k components, regardless of the characteristics of the FIB. We implemented the design in software and demonstrated 10 Gbps throughput with *one billion* synthetic *longest* name prefix matching rules, each containing up to seven components.

We also propose *level pulling* to optimize the average LNPM performance based on the observation that some prefixes have large numbers of next-level suffixes in the available URL datasets.

Categories and Subject Descriptors

C.2.1 [**Network Architecture and Design**]: Network communications

General Terms

Algorithm, Design, Performance.

Keywords

Information-centric networking, longest name prefix lookup, binary search of hash tables.

1. INTRODUCTION

Information-centric networking [1], such as the Named Data Networking (NDN) [2] project, has been proposed to address the shortcomings of the Internet in its current usage while maintaining its strengths. The IP thin waist in the Internet has been extremely successful for supporting large-scale communication between endpoints. ICN focuses on what the content is rather than where it is from, therefore names packets rather than endpoints. To achieve an efficient ICN thin waist design, scalable name prefix lookup solutions are required. In this paper, we focus on two questions. First, is it feasible to perform line-rate *longest* name prefix matching with a large FIB table *regardless of the specific characteristics of the forwarding rules*? Second, are there new generic characteristics in real-world URL datasets that can be leveraged to accelerate existing solutions?

Various naming schemes have been proposed in ICN [3], such as flat self-certifying names and hierarchical human-readable names. We use NDN, which takes the latter approach, as the targeting architecture for our name prefix lookup design, although the algorithms and data structures can be applied to other ICN designs. Each NDN name consists of multiple variable-length *name components*. For instance, the name `/a/b/c/` has three name components delimited by `'/'`, `a/`, `b/`, and `c/`; and its name prefixes are `/a/`, `/a/b/`, and `/a/b/c/`. Note that the NDN forwarding plane includes not only the FIB, but also the pending interest table (PIT) and the content store (CS) [2], which are equally important topics. In this paper, we focus on the longest name prefix matching problem for the FIB.

Several recently proposed name prefix lookup solutions [4, 5, 6, 7] have demonstrated encouraging performance results on CPU- or GPU-based multicore platforms. However, most of these schemes are optimized for or evaluated with a limited number of URL datasets, such as the Alexa [8], Dmoz [9] domain names, the URL blacklist [10], the IRCache traces [11], or crawled URLs [4]. These URL datasets may not fully characterize the NDN forwarding rules in the future. In particular, as namespace design and name assignment principles are still being studied, the characteristics of the NDN FIB tables have not been determined, and it is even possible that multiple namespaces with distinct characteristics may co-exist. Besides, the experience with IP shows that the characteristics also evolve, like the transition from classful forwarding to classless inter-domain routing (CIDR) in the early 1990s. Hence, it is unclear if future FIBs will have characteristics similar to the URL datasets. What's more, several schemes achieve good average-case performance, but the worst-case scenarios require $O(k)$ string lookups, where k is the number of components in each prefix. As a result, the performance of the existing solutions is not guaranteed to be sustainable. Although we believe efficient solutions can always be designed when real-world NDN FIBs become

In this paper, we use terms "name component" and "component", as well as "name prefix" and "prefix", interchangeably.

978-1-4673-6633-5/15/$31.00 © 2015 IEEE

available, we hope to provide a performance baseline that can comfortably support any namespace, and thereby allow the naming schemes to be designed without concern for the forwarding performance.

In this paper, we present a scalable name prefix lookup design based on the binary search of hash tables organized by prefix lengths, which was originally proposed for accelerating IP prefix lookup 18 years ago [12]. With this idea, the number of hash lookups is reduced to $log(32) = 5$ and $log(128) = 7$ for IPv4 and IPv6, respectively. NDN names are much longer than IP addresses in terms of bits, but these names contain explicit delimiters that separate the name components. As a result, the hash tables can be organized by the numbers of components in the prefixes. Applying binary search of hash tables, only $log(k)$ hash table lookups are required for rules that have up to k name components in each prefix, regardless of their other characteristics.

The fact that the presented design is oblivious of FIB characteristics allows synthetic forwarding rules to be easily constructed, so that the *longest* name prefix matching design can be evaluated with large FIB tables, such as the one with *one billion* rules. To the best of our knowledge, the FIB that contains one billion rules is the largest dataset that has been studied for the longest name prefix matching problem.

We have implemented the design in software on a general purpose multicore platform. IP lookup has successfully powered the ever-growing Internet because of efficient algorithms and compact data structure designs. In addition, the IP FIB size is small so that purpose-built hardware that employs high-speed memory devices, such as TCAM or SRAM, can be used. In NDN, such hardware is not large enough to store the entire FIB when the number of rules is large. Several recent works have demonstrated the effectiveness of hash table-based applications on multicore platforms [5, 13]. As a result, we propose a fingerprint-based hash table to implement the idea of binary search of hash tables in software. Our evaluation shows promising performance results with one billion names that have up to seven name components. For the datasets with 15 name components, the performance degraded due to the specific name parsing and hashing implementation, but the cost of hash table bucket memory accesses was still bounded by $O(log(k))$. To demonstrate the performance with real network traffic, we have developed an NDN name prefix lookup engine with Data Plane Development Kit (DPDK) [14] as the underlying packet I/O and multicore framework. We show that 10 Gbps forwarding throughput can be achieved with one billion synthetic longest prefix matching rules that have up to seven name components.

As in IP forwarding, the lookup performance can be improved by taking advantage of the specific characteristics of real-world FIBs. We notice the existence of prefixes that have large numbers of next-level suffixes in the available URL datasets, and thus propose a generic *level pulling* method, which stores a small number of prefixes in the cache or SRAM so that more prefixes can be promoted to the hash table that is accessed first, thus reducing the average number of hash lookups. The effectiveness of level pulling with the available URL datasets is evaluated via simulation.

In this paper, we make the following contributions:

- **Binary search of hash tables for name prefix lookup.** We present a longest name prefix lookup design based on the binary search of hash tables, which provides a reliable forwarding performance guarantee for future ICN research and development. We implemented the design with fingerprint-based hash tables and evaluated its performance.
- **Name lookup engine.** We developed a name lookup engine on a general purpose multicore platform, and we demonstrated that 10 Gbps throughput could be achieved with one billion synthetic names that have up to seven name components. We have released the source code of the name lookup engine on Github[1].
- **Level pulling.** We propose a generic longest name prefix matching optimization method to further reduce the number of hash lookups with real-world URL datasets used in the NDN research literature.

2. BACKGROUND AND RELATED WORK

In this section, we first introduce the challenges in longest name prefix lookup and then present related name lookup solutions.

2.1 Name Prefix Lookup Challenges

NDN names are much longer than IP addresses in bits. The complexity of many IP forwarding solutions is proportional to the length of prefixes, and as a result, directly applying those schemes yields lower performance. On the other hand, NDN names contain explicit delimiters that separate the components. The cardinality of name components at each component level is infinite. This suggests that the number of *name components* in the NDN forwarding rules may not be as large as the number of *bits* in IP addresses.

The number of rules in the NDN FIB table is expected to be much larger than seen with IP. The size of the IP FIB table was only about 530 thousand as of December 2014 [15]. For names, we use the DNS system, which is the largest namespace in the Internet, as an example. For instance, there were already 271 million registered domain names in the Internet in 2013 [16]. What's more, there are 968 million host names in the network, and 181 million of them are active [17]. Although the number of rules in the FIB table is determined by the namespace design and the effectiveness of name prefix aggregation, millions of domains names are expected to be in the FIB in order to handle a network on the scale of the current Internet. As a result, the NDN FIB data structures can hardly fit into SRAM or TCAM [18], thereby DRAM has to be employed, which has longer access latency.

2.2 Existing Name Prefix Lookup Solutions

Name prefix lookup methods can be classified into hash table-, bloom filter-, and trie-based solutions, and we present the most relevant works in each.

Hash table-based solutions. Most hash table-based methods [19, 5, 6] choose fingerprint-based design, because string comparison is required only if the fingerprints match. These solutions differ mostly in the prefix-seeking strategy. The CCNx prototype [19] starts with the full name and then eliminates one component each time if there is no match in the FIB. Cisco's solution [5] begins with querying a prefix with M components, where M is generally the most populated component level. If no match is found, just as in

[1]https://github.com/WU-ARL/ndnfwd-binary-search

CCNx, a shorter prefix is queried. If there is a match, the deepest component level of that specific prefix, denoted as MD, is queried. Then the same strategy as CCNx is used if there is no match at level MD. Wang et al. [6] proposed a greedy prefix-seeking strategy, so that more populated levels are looked up first. The solutions proposed in [5, 6] take advantage of the component number distribution in the rules and achieve better average-case performance. The worst-case scenarios require $O(k)$ hash lookups, except the solution in [5] has a relatively better guarantee, which requires either M or $MD - M$ lookups. In addition, the hash table in [6] stores only 32-bit long signatures, reducing the memory requirements at the cost of forwarding packets incorrectly in case of false positives. Our design differs from previous hash table-based solutions in that the worst-case $log(k)$ hash lookups is always guaranteed regardless of the FIB characteristics.

Bloom filter-based solutions. Bloom filters are typically used to reduce hash table accesses in this context. In NDN, bloom filters have to be stored in DRAM because the number of rules is expected to be large. A naive bloom filter design requires multiple memory accesses for each lookup. The prefix bloom filter proposed by Alcatel-Lucent [7], aims at storing prefixes that share the same first-level component in the same cache-line sized bloom filter. The bloom filter is expanded if the number of suffixes exceeds the bloom filter capacity. As noted in [7], with URL datasets, there were cases where multiple bloom filter expansions were required to store all the prefixes. Although unlikely to happen in practice, a dataset that requires bloom filter expansion at every name component level can be generated, in which a certain number of prefixes have large numbers of suffixes.

Trie-based solutions. Linear search is typically performed in trie-based solutions, thus the lookup complexity is at least $O(k)$, where each step processes a name component. An encoding method [20] has been explored to reduce the FIB memory requirement. Although being relatively more memory-efficient, additional lookups are required to generate the encoded name for each component, increasing the total number of string lookups in the system. The trie-based lookup scheme has been implemented on GPUs to take advantage of their massive parallel processing power [4]. The solution proposed in [4] employs a multi-striding trie, where each step processes multiple characters rather than a complete component. As a result, the number of string lookups is expected to be increased. Previous work on URL-based forwarding [21] has also employed trie-based solutions.

Besides the recently proposed name prefix lookup solutions, CuckooSwitch [13] is also closely related to our work. CuckooSwitch exploits several software optimization techniques, such as large page size and batched software prefetching, to improve the throughput of the Cuckoo hash table. Although a FIB table that contains one billion entries was evaluated in [13], only exact match was performed, and no string matching was involved because the lookup keys were MAC addresses.

2.3 Binary Search of Prefix Lengths

The original paper [12] presented binary search of hash tables organized by prefix lengths to accelerate IP lookup. It also presented mutated binary search which takes advantage of the prefix length distribution for each specific prefix. In particular, the algorithm makes branching decisions based on the characteristics of the suffixes of the visited prefix entry, reducing the average number of memory references significantly. Binary search of prefix lengths has also been used with distributed hash tables [22].

Existing solutions provide many insights on building efficient name prefix lookup systems, although most of these systems have been evaluated with the URL datasets or synthetic datasets that have characteristics similar to those of the original URLs. Our design benefits from these works, while aims at providing a worst-case performance baseline regardless of the characteristics of the forwarding rules. In addition, the largest dataset used in previous works contains 64 million rules [5], while the largest dataset evaluated in this paper has one billion rules.

3. BINARY SEARCH OF HASH TABLES

In this section, we describe how binary search of hash tables works for name prefix lookup, and then present the proposed fingerprint-based hash table design. To evaluate the design, we present the performance micro-benchmarking of the hash table-based implementation, report the name lookup performance on a general purpose multicore platform, and then present the measured forwarding throughput with real network traffic in the end.

3.1 Binary Search for Name Prefix Lookup

Binary search of hash tables organized by prefix lengths was originally proposed for accelerating IP lookup. Although we provide sufficient details about how to apply this idea to name prefix lookup, more discussion about this method can be found in the original paper [12].

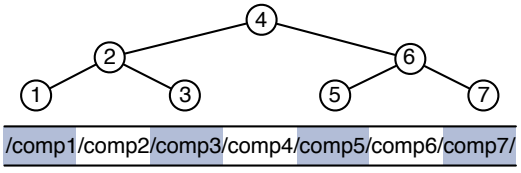

Figure 1: Binary Search for Name Prefix Lookup

Hash tables for name prefix lookup are organized by the numbers of their name components. That is prefixes with the same number of name components are stored in the same table. For names with up to k name components, k hash tables are created, and these hash tables form a balanced binary search tree. To locate the longest matching prefix for each querying name, binary search is performed on these k hash table nodes. At each node, if there is a matching prefix in the corresponding hash table, then the algorithm proceeds to the right subtree to search for a potential longer matching prefix; if there is no match, then the longest matching prefix has to be in the left subtree. The lookup procedure terminates only when the bottom binary search level is reached, or when a leaf FIB entry is reached. The total number of hash tables visited is bounded by $log(k)$. For each lookup, generally $log(k)$ hash lookups are required, because the lookup procedure has to access the hash table that stores prefixes with one more name component, which is likely to be at the bottom level of the search tree, to confirm that there is no

longer matching prefix. For names that match leaf entires, the lookup procedure can be terminated immediately if a matching leaf entry is encountered. Figure 1 shows an example with rules that have up to seven name components, where up to three hash lookups are required for each query. Each lookup starts with the hash table node 4, and then if there is a match, it proceeds to node 6, otherwise, it backs off to node 2. The procedure continues until the termination condition is satisfied.

As shown in [12], additional *marker* entries are required to ensure that prefixes can find shorter matching prefixes. For example, assume that `/a/b/c/d/` and `/a/b/c/` are both in the FIB, but `/a/b/` is not. With the binary search shown in Figure 1, a name `/a/b/c/random` first visits hash table 4, and finds no match. In this case, it proceeds to the left subtree, and then finds no match for `/a/b/` at hash table 2. Eventually, `/a/` is found to be the longest matching prefix at hash table 1, however, the correct longest prefix should be `/a/b/c/`. To resolve this issue, a marker entry `/a/b/` needs to be added to hash table 2. Adding marker entries increases the memory consumption, but the number of additional marker entries is bounded by $log(k)$ for each prefix [12]. For convenience, marker entries can be added when the forwarding rules are inserted into the hash tables. Essentially, whenever a hash table is visited during an insertion, if the number of name components in the name prefix is no less than the number of components of the prefixes stored in this specific hash table, then either a marker or the actual prefix entry has to exist afterwards. Recent proposals that change the prefix seeking strategies [5, 6] also need to insert additional prefixes into the hash tables.

It has also been noticed that adding markers directly introduces the backtracking problem in [12]. We illustrate the problem with the same example again. Now assume that when the name `/a/b/random/` is looked up, the algorithm checks hash table 2 because the full name has only three components. With the help of the marker entry `/a/b/`, it finds a match at hash table 2 and then proceeds to hash table 3, where eventually no match is found. In this case, backtracking is required: the algorithm needs to visit `/a/` to find the correct longest prefix matching results. As noted in [12], the worst-case backtracking could be k hash lookups. To resolve this problem, each marker stores the longest prefix match information inherited from its own longest matching prefix. In this case, for the same example, when it is determined that `/a/b/c/` is a mismatch, the algorithm can safely return the forwarding information of `/a/b/`, which is inherited from `/a/`.

3.2 Fingerprint-Based Hash Table

In this section, we first present the hash table design, and then illustrate the string matching strategies.

3.2.1 Hash Table Memory Layout

Hash tables have been widely used in network applications [23]. The original binary search of hash tables organized by prefix lengths stores IP prefixes, which have a maximum of either 32 bits or 128 bits. Storing name prefixes requires much more memory space than IP prefixes and potentially has larger memory footprint. Fingerprint-based hash tables have been used to reduce the number of memory accesses, where fingerprints are hash values of the keys in the table. Typically, fingerprint-based hash tables have cache-line sized buckets, where each bucket stores a constant number, denoted as E, of fingerprint entries. Each fingerprint entry contains a fixed-length fingerprint and also stores either the string address or an index that eventually leads to the actual string. This way, each hash lookup requires one hash bucket access, followed by one string comparison if there is a matching fingerprint in the bucket. On the other hand, if a naive hash table is used with the same bucket setup, in the worst case, E memory accesses are required.

The number of memory accesses is also affected by hash table load factors. Higher load factors result in more hash collisions, and therefore require additional memory accesses. Hash tables with multiple choices, such as d-left hash tables [24] or Cuckoo hash tables [25], support high load factors with a constant number of memory accesses for each hash lookup. However, on average, $(1 + d)/2$ or 1.5 hash bucket accesses are required for d-left hash tables and Cuckoo hash tables, respectively. To keep the average number of memory accesses low, we choose to trade memory space for speed. Similar as in [5], the hash table has a relatively low load factor so that most hash lookups require only one hash bucket access, and chaining is used to resolve bucket overflows. In our experiments, to store n items in a hash table, $n/4$ hash buckets are allocated, and $10\% \times n/4$ additional buckets are preallocated in case of bucket overflows.

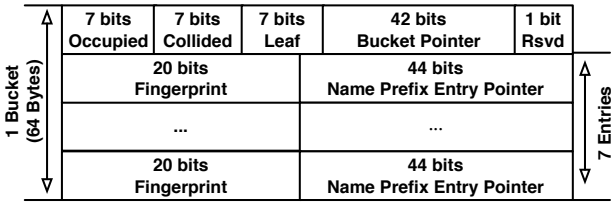

Figure 2: Hash Bucket Memory Layout

Hash bucket design. The organization of the hash bucket is shown in Figure 2. Each hash bucket takes one cache line, i.e., 64 bytes on our system, and contains up to seven fingerprint entries, where each entry stores a 20-bit fingerprint and a 44-bit name prefix entry pointer, which is the address where the forwarding information and the actual prefix string are stored. Note that although each pointer takes 64 bits by default in 64-bit operating systems, current processors support up to 48-bit virtual addresses. As a result, we need to store only the lower 48 bits of the address. In addition, the address length can be reduced by aligning the name prefixes on a 16-byte boundary, thereby saving four more bits. The address size can be reduced further if name prefix entries are preallocated with a fixed size, thus only an offset index needs to be stored, but the name prefix storage could be much larger because the size is fixed and larger than the prefix lengths. To facilitate examining the fingerprint entries, each entry has one Occupied bit indicating whether the entry is taken, one Collided bit indicating whether there is a collision for this entry, and one Leaf bit indicating whether this is a leaf entry. The hash bucket also stores a 42-bit memory pointer that holds the address of the chained hash bucket in case of bucket overflow. We use 42-bit pointers because hash buckets are aligned on a 64-byte boundary. Lastly, one bit in each hash bucket is unused.

Fingerprint collisions during insertion are indicated by the Collided bits in the hash buckets. In other words, there could be duplicate fingerprints in a bucket, while their corresponding name prefixes are different. During a lookup, if a collided fingerprint is matched, then all of the matched fingerprint entries must be visited to find the correct matching prefix. It is possible to delay the string matching until the end with string matching strategies that perform string matching at the end of the lookup as illustrated in the Section 3.2.2, in the hope that a longer and collision-free prefix can be matched. In our implementation, because fingerprint collisions during insertion are rare, we simply perform string matching immediately if a collided fingerprint entry is visited.

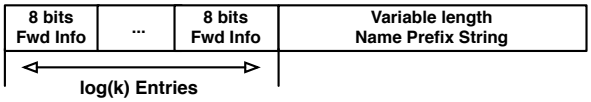

Figure 3: Name Prefix Entry Memory Layout

Name prefix entry. Each name prefix entry stores the forwarding information and the actual name prefix string, which is compared with when a longest matching prefix candidate is found. In the current implementation, the forwarding information stores an eight-bit index that identifies the outgoing port and the destination MAC address. A naive name prefix entry stores forwarding information for only one name prefix, thus the additional marker entries also have their own name prefix entries, increasing the memory requirements. In our design, as shown in Figure 3, each name prefix entry stores the forwarding information for up to $log(k)$ entries, so that when marker entries are inserted, only additional fingerprint entries are added into the hash tables, and the name prefix string of a marker entry is shared with the original name prefix. As mentioned before, the name prefix entries are aligned on a 16-byte boundary in our implementation.

3.2.2 Hash Table String Matching Strategy

String matching is performed when there is a fingerprint match. With 20-bit fingerprints, the chances of getting a false positive are low with normal network traffic. For instance, when E fingerprint entries are visited, the expected false positive rate is $E \times 2^{-20}$. As a result, it appears to be possible to perform string matching only at the end, after a longest matching prefix has been determined by looking up solely fingerprints. Unfortunately, when a non-cryptographic hash function is used to generate fingerprints, names that always cause false positives can possibly be generated, degrading the name prefix lookup performance. Because we are interested in the worst case performance, to address the issue, cryptographic hash functions can be employed [5]. We present the detailed analyses of two string matching strategies below.

Always perform string matching when fingerprints match. In this approach, string matching is performed whenever there is a fingerprint match. Hence each fingerprint-matched hash lookup involves retrieving the hash bucket and fetching the name prefix entry. As we need up to $log(k)$ hash lookups for each name query, in the worst case, $log(k)$ hash buckets and $log(k)$ name prefix entries are accessed. Note that we assume each hash lookup requires one bucket access, because the chances of getting a bucket overflow are rare. Names that always trigger false positives cause the worst-case behavior. In addition, the worst case could also happen with normal traffic. For instance, for rules that have up to seven components, when the longest matching prefix is at hash table 7, string matching is always required at hash tables 4, 6, and 7. As described later in Section 3.3.1, the worst case happens when the visited entries have different name prefix entries. The advantage of this approach is that it allows fast software-based non-cryptographic hash functions, such as CityHash [26], to be employed.

Perform string matching only at the end of the search. The longest name prefix matching can be divided into two stages. The first stage determines the matching prefix length, and the second stage verifies the matching prefix and retrieves the forwarding information [5, 7]. If string matching is performed only at the end, the longest matching prefix length is determined solely by the fingerprint lookups. This way, in the worst case, $log(k)$ hash buckets and one name prefix entry are accessed. In case of false positives, a backtracking must be performed. In our implementation, a binary search that always performs string matching is required when false positive happens. To provide a reliable performance baseline, cryptographic hash functions are required. As pointed out in [5], SipHash [27] is one such hash function that can be employed.

3.3 Hash Table Performance

The performance study aims to demonstrate the presented design supports large FIB tables efficiently in software with modest performance optimization.

3.3.1 Experimental Setup

The experiments in this paper were performed on a Dell PowerEdge R620 rack server equipped with two six-core Intel Xeon E5-2630 processors and 192 GB of DDR3 memory. The detailed architecture and configuration of the system are shown in Figure 4 and Table 1. All of the experiments were performed within the DPDK environment, which provided huge page memory allocation supports.

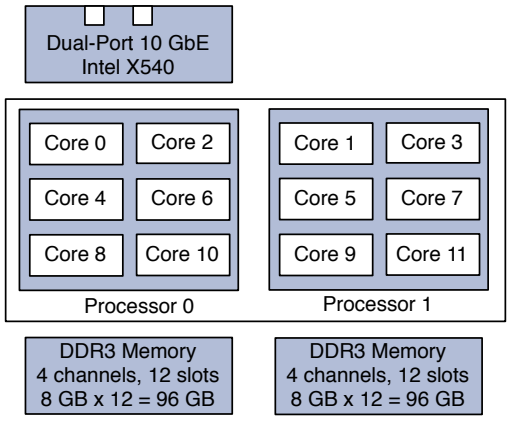

Figure 4: System Architecture

Table 1: System Configuration

CPU	2×Intel Xeon E5-2630, 2.30 GHz
L1d cache	32 KB
L2 cache	256 KB
L3 cache	2×15 MB
DRAM	2×96 GB DDR3, 1333 MHz
OS	Ubuntu 12.04 LTS

We focus on evaluating the worst case performance, and study both the case that always performs string matching in case of fingerprint match using CityHash, denoted as CityA, and the case that performs string matching at the end using SipHash, denoted as SipE. For the CityA case, the worst case scenario may not be obvious. For instance, n forwarding rules that all have k components do not represent the worst case, because although the number of hash table fingerprint entries is increased to $log(k) \times n$ due to the additional marker entries, the name prefix entries are shared by the marker entries and the actual name prefixes. Thus, during a lookup, although $log(k)$ string matching operations are required, the name prefix entry is fetched from memory only in the first time, subsequent string matching operations are expected to get cache hits. The worst case is when string matching is always required and the visited name prefix entries have different memory addresses. To emulate this situation, we populated the hash tables in two phases. We illustrate the procedure using the dataset with seven name components as an example. In the first phase, $n - n/log(k)$, i.e., $2n/3$ names with seven components are inserted, where marker entries are inserted into hash tables 4, 6, and 7, without storing the name prefix entries. In the second phase, $n/log(k)$, i.e.,$n/3$, names with seven components are inserted into the hash tables, where marker entries are inserted into hash tables 4 and 6, and their prefix entries are also stored. This way, the hash tables 4, 6, and 7 store $3n$ fingerprints in total, and n name prefix entries are stored in memory, representing the worst case scenario. It is worth noting that in phase two, name entries with the same number of components are inserted together in batches, so that the memory locations of all the prefix entires of the same name are separated. Modern computer systems employ non-uniform memory access (NUMA), and local memory accesses are faster than remote memory accesses. As our system has two NUMA nodes, in the experiments, hash tables are allocated first, and then name prefix entries are allocated. This way, hash tables are always located at NUMA 0, and name prefix entry storage may include memory from NUMA 1 when no memory is available from NUMA 0.

With the above worst case scenario, synthetic rules can be easily generated. We developed a Python program to generate name prefixes. Each prefix has k name components, where each name component has six to ten random ASCII characters. The range of numbers of characters in each name component is based on the URL characteristics presented in [20, 5]. Each dataset contains n names, where the first $n/log(k)$ names are inserted in the second phase, and the rest $n - n/log(k)$ names are inserted in the first phase. The lookup traces are generated by randomizing the first $n/log(k)$ names using the `shuf` program.

In the rest of this paper, we always present performance results of datasets with up to seven name components, except in Section 3.3.4, which includes performance results with datasets containing 15 components. In most figures, we show the performance with both 512 million names (512M) and one billion (1B) names, because 512M is the largest dataset that can fit into one NUMA node in our experiments, and 1B requires allocating memory from both NUMA nodes. The memory requirements of 512M and 1B are 57.3 GB (27.0 GB of hash tables and 30.2 GB of name prefix entries) and 111.8 GB (52.8 GB of hash tables and 59.0 GB of name prefix entries), respectively.

3.3.2 *Page Sizes*

Computer systems employ the virtual memory system, thus whenever a memory is referenced, a virtual memory address is translated to a physical memory address. The translation lookaside buffer (TLB) is employed to accelerate the address translation procedure. However, for applications require large memory space and expose scarce memory access locality, the page-based virtual memory system consumes a considerable amount of CPU cycles due to TLB misses [28]. To reduce the amount of TLB misses, large pages can be employed.

We evaluated the lookup performance with three different page sizes: the default 4 KB pages, 2 MB pages, and 1 GB pages. Both the CityA and SipE string matching strategies were used with datasets 512M and 1B. In the experiments, we allocated 128 GB large page memory, which was equally distributed between two NUMA nodes. We ran each experiment three times, and the average lookup latencies are shown in Figure 5.

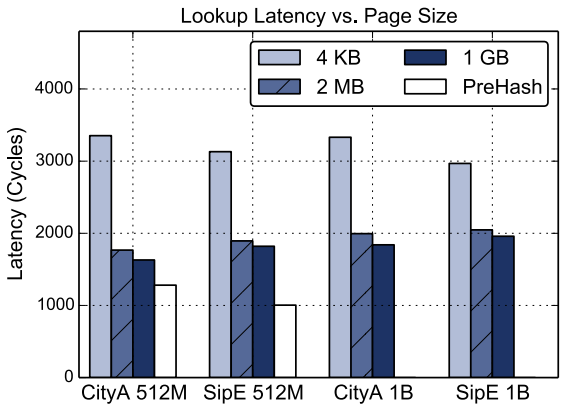

Figure 5: Impact of Page Sizes

In all of the presented cases, the lookup latency was reduced significantly (about 31% to 47%) when the page size was increased from 4 KB to 2 MB. When the page size was increased from 2 MB to 1 GB, only a small latency reduction (about 4% to 8%) was observed, which is likely due to the small number of TLB entries when 1 GB pages are used. When large pages were used, the cases with 1B had longer lookup latencies, because a large portion of the name prefix entries were allocated in NUMA 1, incurring higher-cost remote memory accesses. In the rest of the paper, the experiments were always performed with 1 GB pages.

Precomputed Hash Values. To quantify the impact

of hash computation, we measured the lookup latency with precomputed hash values. In the experiment, hash values were stored in an array following the prefix lookup order, thus minimizing the hash value memory access overhead. Due to the memory size constraint, we measured only the results for the 512M dataset. The average lookup performance is shown in Figure 5. For the CityA and SipE cases, the lookup latency is reduced by 21% and 45%, respectively. In addition, SipE outperforms CityA, which is expected because of less memory accesses.

Although SipHash is expected to be slower than CityHash, the measured numbers of cycles spent on hash computation depend on the specific implementation. In our design, we used the original CityHash reference implementation [26], and we modified the SipHash reference implementation [27] so that all of the k hash values can be computed in one pass, as suggested in [5]. The performance of CityA and SipE can be improved if more efficient hash functions are used. For instance, purpose-built hardware may include hardware-based hash units that compute hash values efficiently. We have released our source code online, so that more efficient hash implementation or optimization can be evaluated by others.

3.3.3 *Software Prefetching*

Software prefetching has been demonstrated to be effective for accelerating hash lookups. In [5], the hash buckets corresponding to the prefixes of the querying name are all prefetched. In [13], prefetching is batched for every 16 packets, therefore higher hash table throughput is achieved because sufficient delay is placed between prefetching and the actual data access.

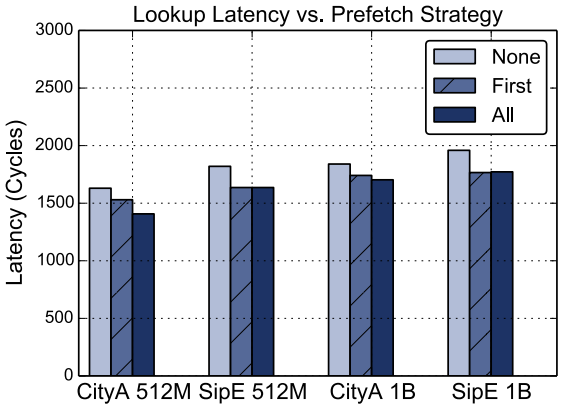

Figure 6: Impact of Prefetching Strategies

We studied the impact of prefetching strategies on the lookup latency. In the experiments, hash values were always computed for all of the prefixes together because of better performance. The hash values were also computed in the case in which no prefetch was performed, thus only the prefetching strategy was varied. In this experiment, we evaluated two prefetching strategies: prefetch only the first visited hash bucket, which is guaranteed to be accessed; and prefetch all of the hash buckets, where $log(k)$ out of k buckets are eventually accessed. The software prefetching instruction was issued once the hash value was computed for each prefix. All of the experiments were performed on a single core with 1 GB pages. The performance results are shown in Figure 6. For CityA, prefetching the first visited bucket achieved about 5% latency reduction for both 512M and 1B, and prefetching all of the buckets reduced the lookup latency by 13% and 7% for 512M and 1B, respectively. For SipE, the performance improvements of prefetching only the first visited bucket and all of the buckets are comparable, which is about 10% for both 512M and 1B.

3.3.4 *Performance with Various Datasets*

In this section, we present the name prefix lookup performance with different dataset sizes and longer names.

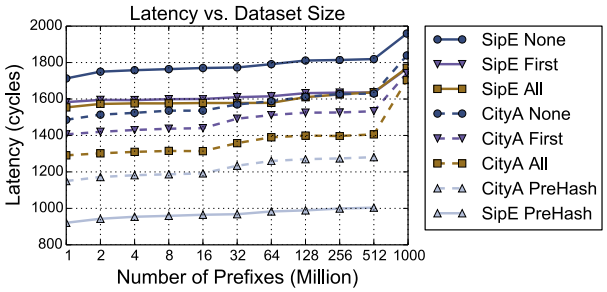

Figure 7: Datasets with Seven Components

Various dataset sizes We present the performance of both CityA and SipE with different dataset sizes and prefetching strategies in Figure 7. For the cases where hash values were precomputed, no prefetching was performed. The average number of cycles increases as the dataset becomes larger. When all the memory are allocated from NUMA 0, i.e., no more than 512 million forwarding rules are used, CityA performs better than SipA, and the performance with different prefetching strategies for various datasets is consistent with our previous observation with the dataset 512M. For the case with 1B, the lookup latencies increase considerably in both CityA and SipE. The performance gap between these two approaches becomes smaller, this is likely due the fact that CityA requires two more memory accesses for each lookup, and that name prefix entires are mostly allocated in the remote NUMA node.

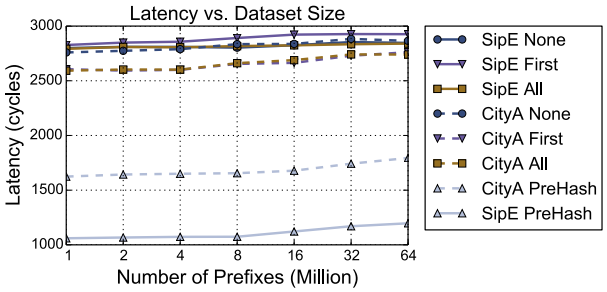

Figure 8: Datasets with 15 Components

Datasets with 15 name components. We present the performance with name prefixes that have 15 name components in Figure 8. Ideally, since at most $log(k)$ hash

lookups are required for datasets with up to k components, the lookup latencies with 15 name components would be about 25% and 33% longer than the ones with seven components for SipE and CityA, respectively. However, the measured lookup latencies are considerably higher, which is largely due to the increased hash computation cost, where although only $log(k)$ buckets are eventually accessed, k hash values are always computed together in the current implementation. Computing hash values together is not only required for prefetching all the buckets, but also has better performance with the seven-component datasets. As a result, the number of hash operations is proportional to k. In addition, the cost of performing parsing, hashing, and string comparison also increased as names became longer. To improve the performance, purpose-built hardware hash units can be employed.

The benefits of prefetching also diminished with 15 components. As shown in Figure 8, performing prefetching for SipE did not improve the performance. When only the first visited bucket was prefetched right after its hash value was computed, the performance was even worse than the case without prefetching. We measured the number of cycles spent on prefetching and binary search using the `RDTSC` instruction. Our preliminary results indicate that the hash bucket might be prefetched too early, where seven more hash values still need to be computed, thus reducing the effectiveness of prefetching. A more efficient prefetching strategy could issue the prefetching instruction at a later time in the process. When all the buckets were prefetched, the increased cost of prefetching offset the reduced number of cycles in binary search. The prefetching performance with CityA was better than SipE, but the benefits of prefetching all buckets still diminished due to excessive prefetching.

Thus, more efficient name parsing and hash implementation as well as prefetching strategies are needed. Nevertheless, as shown at the bottom of Figure 8, when hash values were precomputed, the lookup latencies were much closer to the expected values, determined by $log(k)$ number of hash lookups.

3.4 Multicore Performance

We present the performance of the design on the multicore platform in this section. In the experiments, similar as in [5], a dedicated core loaded names from a local file, and then each name was copied into a packet buffer, which was then distributed to worker threads via software rings provided by DPDK. The worker threads performed name lookup and then released the buffers.

The performance results are shown in Figure 9. In the experiments, each worker thread ran on a dedicated core, and hyper-threading was disabled. We ran up to four worker threads on one NUMA node, because each node had six cores. As shown in Figure 9, the throughput increases proportionally to the number of threads.

To evaluate the effect of NUMA architecture, we also ran four worker threads on NUMA 1, but all the data structures and the core that generated names, were allocated on NUMA 0. For 512M, all the data structures were in NUMA 0, as a result, the throughput with four worker threads on NUMA 1 was degraded by 20% and 8% for CityA and SipE, respectively. For 1B, the hash tables were allocated on NUMA 0, but a majority of name prefix entries were on NUMA 1, therefore the throughputs of running four threads

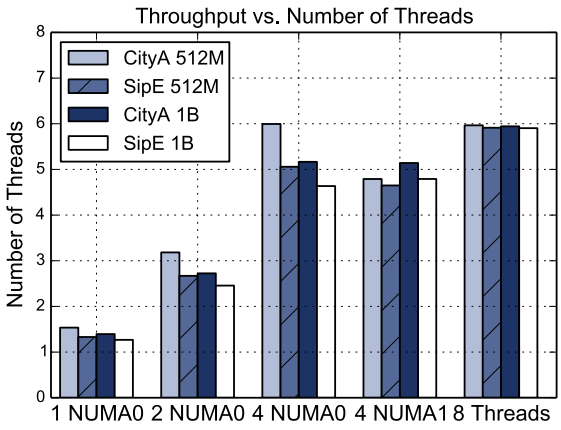

Figure 9: Multicore Performance (Prefetch All)

on NUMA 0 and NUMA 1 were comparable.

In the end, eight threads were ran on NUMA 0 and 1, and we expected to see further throughput increase. However, the performance improved only to about 6 MPPS, which was also achieved by running four threads on NUMA 0 with the 512M dataset using CityA. We are still seeking to confirm why, a plausible explanation is that some resource contention happened between NUMA nodes. We aim to explore it further in future work.

3.5 System Evaluation

To evaluate the name prefix lookup performance with real network traffic, we developed an NDN forwarding engine on top of the DPDK packet I/O and multicore framework [14]. The forwarding engine performed only longest name prefix lookup, and we plan to integrate other components, such as the pending interest table (PIT) and the content store (CS), in future work.

8 bits Packet Type	16 bits Packet Length	16 bits Name Length	Name	(Data)

Figure 10: Simplified Packet Format

In the experiments, we measured the forwarding throughput of one billion synthetic forwarding rules that have up to seven components. To evaluate the worst-case performance, the same experimental setup in the previous two subsections was used. For fast prototyping, we employed a simplified NDN packet format, as shown in Figure 10. The Packet Type field, which takes eight bits, indicates if this is an Interest packet or a Data packet [2]. In our experiments, only Interest packets are generated. The Total Length and Name Length fields store the length of the packet and the name field, respectively. The Name field stores the packet name and has variable length. The Data field, existing only in Data packets, holds the carried content. In our implementation, NDN packets are transmitted on top of UDP. When a packet arrives at the forwarding engine, its name is looked up, and the MAC addresses of this packet are updated according to the lookup results before the packet is

delivered.

DPDK supports zero-copy fast packet I/O and a multi-core framework for fast packet processing applications. We modified the existing DPDK load balancer sample application [29], whose original structure was suitable for our needs. In this design, packets arrived at NIC are fetched by the I/O threads, and then packets are distributed to worker threads via the RTE rings provided by DPDK. The specific worker thread is determined based on hash values of the full packet name, which is required for designs that support dedicated PIT for each worker thread because packets with the same name need to be processed by the same thread. If a centralized PIT is employed, the I/O thread can simply distribute packets to worker threads in a round-robin fashion.

The experiments were performed in the Open Network Laboratory [30], which provided isolated performance evaluation environments. We used the DPDK-based Pktgen application [31] to generate NDN traffic. The Pktgen application can both transmit and receive packets. In our experiments, we configured an eight-core machine as the receiver, and a 12-core machine as the sender, because the 12-core machine had larger memory space that could hold the entire lookup traces. The sender was directly connected to the name lookup engine, and the receiver was connected with the lookup engine via a 10 Gbps switch.

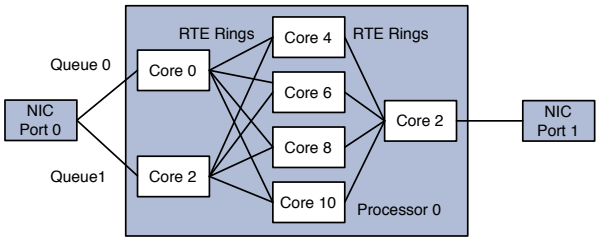

Figure 11: Experiment Configuration

Figure 11 shows the experiment configuration with four worker threads (Core 4, 6, 8, and 10) allocated on the same NUMA node to perform name prefix lookup. Due to the limited performance of the I/O thread, which fetches the packets and computes hash values to distribute the packets, two I/O threads (Core 0, 2) were employed in the experiments. Packets received at NIC Port 0 were distributed to these two I/O threads using the multi-queue feature provided by the NIC. Current multi-queue support typically distributes packets based on the IP five tuples, as our NDN traffic were on top of UDP, the source IP addresses of the generated packets were randomized. When the outgoing port and destination MAC address were determined, packets were sent to the I/O thread (Core 2) and then delivered at NIC port 1. For simplicity, the NIC and Core 2 are shown twice in Figure 11.

The forwarding throughput was reported by the Pktgen program on the receiver side. The observed forwarding throughputs with 256-byte packets are listed in Table 2. When all of the hash buckets were prefetched, 9.7 Gbps and 9.1 Gbps throughputs were achieved for CityA and SipE, respectively.

When eight worker threads were employed using both processors, 10 Gbps throughput was achieved for all of the cases listed in Table 2.

Table 2: Throughput with 256-Byte Packets

	CityA		SipE	
	MPPS	Gbps	MPPS	Gbps
None	4.1	9.1	3.8	8.4
First	4.2	9.4	4.0	8.9
All	4.4	9.7	4.1	9.1

4. LNMP OPTIMIZATION

In this section, we present level pulling, a generic method that reduces the average number of hash lookups for longest name prefix matching.

Like in IP, prefix lookup can be optimized according to the specific characteristics of the forwarding rules. Such optimization normally focuses on improving the average-case performance because routers have buffers that can tolerate temporal long latency. Ideally, optimization should be based on usage, i.e., rules that are looked up more frequently have better performance. However, NDN traffic patterns are not currently available because it has not yet been widely deployed. Recent prefix lookup optimizations are generally based on name component number distribution [5, 6], so that the hash tables that store more prefixes are visited early on. As noted in [6], most entries in the URL datasets are leaf entries. Hence, optimizations based on name component number distribution can be approximately solved by minimizing the weight of a binary search tree, where the weight of each node is the number of leaf entries in the corresponding hash table. Our goal is to further reduce the average number of hash lookups by exploring new characteristics in the URL datasets.

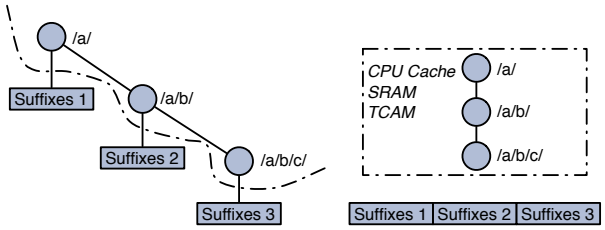

Figure 12: Level Pulling Concept

We propose the level pulling idea based on the observation that some prefixes have large numbers of next-level suffixes in the URL datasets. For instance, in the Alexa dataset, which contains the top one million visited domain names, 3,363 URLs share the prefix `http://youtube.com/user/` and 208 URLs share the prefix `http://sites.google.com/`. If these URLs could be stored in the first-visited hash table, the average name prefix lookup performance could be improved. This characteristic does not exist in IP because each IP prefix length level is only one bit, but in NDN, the cardinality of each name component level is infinite.

We propose to store prefixes with large numbers of next-level suffixes in a cache-like structure. Hence, in addition to reordering the hash table lookup sequence, we modify the starting point of each lookup. The key challenge is to keep the data structure compact so that it can be stored in the

CPU cache, or in SRAM or TCAM with hardware-based solutions. The size of the data structure is largely determined by the number of prefixes being stored. Figure 12 shows the concept of level pulling with a name-component trie as the cache-like structure. In this paper, we evaluate the percentage of hash lookup reduction via simulation. A Python program was developed to measure the reduced number of hash lookups.

Table 3: Existing Name Conversion Schemes

URL	http://www.named-data.net/project/
TLD [5, 6]	/net/named-data/www/project/
Site [1]	/named-data.net/www/project/
Host [7]	/www.named-data.net/project/

Various ways of converting URLs to NDN names have been used in research literature, as listed in Table 3. The first scheme reverses the domain name, i.e., it starts with the top level domain (TLD) name as the first component, followed by all the subsequent components; the second scheme starts with the site name as the first component; and the third uses the entire host name as the first component. Using all of these three schemes, we generated nine NDN name datasets for the Alexa, Dmoz, and Blacklist URL datasets, which has one million, 3.69 million, and 1.39 million URLs, respectively.

Table 4: Hash Lookup Reduction Percentages

	TLD		Site		Host	
	α (%)	Size	α (%)	Size	α (%)	Size
Alexa	12.08	214	4.43	122	0.49	19
BlackL.	9.94	461	6.52	437	4.77	271
Dmoz	11.58	1,781	8.13	1,742	8.60	1,639

We then evaluate hash lookup reduction percentage with level pulling. For each dataset, all of the prefixes are inserted into a name-component trie, and then the prefixes whose name-component trie nodes have more than a threshold, denoted as T, number of child nodes are stored in the cache-like structure, namely C-Trie. The corresponding next-level suffixes of the stored prefixes are promoted to the first-visited hash table. Smaller T values improve the hash lookup reduction percentage, but also increase the C-Trie size. Just as an example, we set T as 64 to show the effectiveness of level pulling. During a lookup, the C-Trie is visited first. If there is no match in the C-Trie, the name is then looked up in hash tables; if there is a match, then the next-level suffix of the matching prefix is looked up in hash tables. To measure the required number of hash lookups with level pulling, the same datasets were looked up, where each name in the datasets was appended with three additional random components. Because we focus on the average-case performance, each name was looked up only once. We collected the number of hash lookups that were performed on the hash tables. The measured percentages of reduction, denoted as α, and the number of prefixes stored in C-Trie for these nine datasets are listed in Table 4. For both the Alexa and Blacklist datasets, the hash lookup reduction is higher with the TLD scheme, followed by the Site and Host schemes, because the TLD approach has more aggregated prefixes for these two datasets. For the Alexa dataset with the Host first scheme, the hash lookup reduction percentage is only 0.49%, this is because 98.94% of the prefixes have only one name component in that dataset. For the Dmoz dataset, the numbers of prefixes stored in the C-Trie for the presented three name conversion schemes are close to each other. This is because a significant portion of the prefixes stored in the C-Trie contain the components corresponding to the host names of the URLs. Although the number of reduced hash lookups in the Site first scheme is greater than the one in the Host first scheme for the Dmoz dataset, the Host first scheme has slightly higher hash lookup reduction percentage because it requires less number of hash lookups originally. In all of these cases, the largest C-Trie contains only $1,781$ prefixes, requiring a small amount of storage.

The simulation shows encouraging results, although the best way to demonstrate the performance is to implement such a system. Our next step in level pulling is to implement the design and evaluate the performance.

5. DISCUSSION

Although the presented design has demonstrated reliable performance, the following questions remain in longest name prefix matching.

First, the presented name prefix lookup design consumes much memory. For large datasets, the prefix strings already occupy considerable memory, and our hash table design uses more memory to achieve speed with relatively low load factor. Moreover, the nature of binary search requires additional marker entries, which further increases the hash table size. Although modern servers are capable of supporting more than 200 GB of memory and the cost of memory keeps dropping, it is always desirable to have a more compact FIB representation. Smaller memory space reduces the cost of power, and also enables data structure replication among NUMA nodes to improve performance.

Second, the FIB lookup is just one component of the NDN forwarding plane. When other components, such as the pending interest table (PIT) and content store (CS) are integrated, the overall system performance can be further optimized. For example, cryptographic hash functions are required for the PIT [5], therefore an efficient combination of hash functions is needed.

6. CONCLUDING REMARKS

In this paper, we present a longest name prefix lookup design based on binary search of hash tables organized by the numbers of name components in the prefixes. For forwarding rules that have up to k name components in each prefix, regardless of their specific characteristics, this design always guarantees at most $log(k)$ hash lookups. Taking advantage of the recent advances in multicore packet processing platforms, we implemented the design with fingerprint-based hash tables in software and demonstrated that 6 MPPS can be supported with one billion synthetic forwarding rules that have up to seven name components. We prototyped the forwarding engine design, and demonstrated that 10 Gbps forwarding throughput could be achieved with 256-byte packets. ICN might not reach billions of names in the next few years, but we hope that by demonstrating the feasibility of

line-rate name-based forwarding for large FIBs, researchers and application developers can comfortably choose the most efficient namespace design, without concern for the packet forwarding performance.

We have also identified the existence of prefixes with large number of next-level suffixes in the URL datasets, so that when practical NDN forwarding rules are available, there will be an additional tool to optimize the lookup procedure.

Acknowledgment

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

7. REFERENCES

[1] Van Jacobson, Diana K. Smetters, James D. Thornton, Michael F. Plass, Nicholas H. Briggs, and Rebecca L. Braynard. Networking Named Content. In *Proc. of the Fifth ACM CoNext*, 2009.

[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*, 2011.

[4] Yi Wang et al. Wire Speed Name Lookup: A GPU-based Approach. In *Proc. of Tenth USENIX NSDI*, 2013.

[5] Won So, Ashok Narayanan, and David Oran. Named Data Networking on a Router: Fast and Dos-resistant Forwarding with Hash Tables. In *Proc. of the Ninth ACM/IEEE ANCS*, 2013.

[6] Yi Wang et al. Fast name lookup for Named Data Networking. In *IEEE 22nd International Symposium of Quality of Service (IWQoS)*, May 2014.

[7] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue. Caesar: A Content Router for High-speed Forwarding on Content Names. In *Proc. of the Tenth ACM/IEEE ANCS*, 2014.

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

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

[10] URL Blacklist. http://urlblacklist.com/.

[11] IRCache. http://www.ircache.net/.

[12] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. In *Proc. of the ACM SIGCOMM '97*.

[13] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proc. of the Ninth ACM CoNext*, 2013.

[14] Intel. Intel Data Plane Development Kit (Intel DPDK). http://www.dpdk.org/.

[15] CIDR Report. http://www.cidr-report.org/as2.0/.

[16] Domain Name Industry Brief. http://www.verisigninc.com/.

[17] June 2014 Web Server Survey. http://news.netcraft.com/archives/category/web-server-survey/.

[18] Diego Perino and Matteo Varvello. A Reality Check for Content Centric Networking. In *ICN '11*.

[19] The CCNx Project. http://www.ccnx.org/.

[20] Yi Wang et al. Scalable Name Lookup in NDN Using Effective Name Component Encoding. In *Proc. of the 32nd IEEE ICDCS*, 2012.

[21] B. Scott Michel, Konstantinos Nikoloudakis, Peter Reiher, and Lixia Zhang. URL Forwarding and Compression in Adaptive Web Caching. In *Proc. of the INFOCOM 2000*.

[22] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief Announcement: Prefix Hash Tree. In *Proc. of the Twenty-third ACM PODC*, 2004.

[23] Adam Kirsch, Michael Mitzenmacher, and George Varghese. Hash-Based Techniques for High-Speed Packet Processing. In *Algorithms for Next Generation Networks*, pages 181–218. Springer London, 2010.

[24] Andrei Broder and Michael Mitzenmacher. Using Multiple Hash Functions to Improve IP Lookups. In *Prof. of Twentieth IEEE INFOCOM*, 2001.

[25] Rasmus Pagh and Flemming Friche Rodler. Cuckoo Hashing. *J. Algorithms*, 51(2), May 2004.

[26] CityHash. https://code.google.com/p/cityhash/.

[27] SipHash. https://131002.net/siphash/.

[28] Arkaprava Basu, Jayneel Gandhi, Jichuan Chang, Mark D. Hill, and Michael M. Swift. Efficient virtual memory for big memory servers. *SIGARCH Comput. Archit. News*, 41(3):237–248, June 2013.

[29] Intel DPDK Sample Application User Guide. http://www.intel.com/content/dam/www/public/us/en/documents/guides/intel-dpdk-sample-applications-user-guide.pdf.

[30] Charlie Wiseman et al. A Remotely Accessible Network Processor-based Router for Network Experimentation. In *Proc. of the Fourth ACM/IEEE ANCS*, 2008.

[31] Pktgen-DPDK. https://github.com/Pktgen/Pktgen-DPDK/.