Winter 12-15-2015

Data Structures and Algorithms for Scalable NDN Forwarding

Haowei Yuan
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering & Applied Science

Department of Computer Science & Engineering

Dissertation Examination Committee:
Patrick Crowley, Chair
Roger Chamberlain
Roch Guerin
Raj Jain
Neil Richards
Jonathan Turner

Data Structures and Algorithms for Scalable NDN Forwarding

by

Haowei Yuan

A dissertation presented to the
Graduate School of Arts & Sciences
of Washington University in
partial fulfillment of the
requirements for the degree
of Doctor of Philosophy

December 2015
Saint Louis, Missouri

Table of Contents

List of Figures . . . . . **v**

List of Tables . . . . . **vii**

Acknowledgments . . . . . **viii**

Abstract . . . . . **xii**

1 Introduction **1**
- 1.1 Named Data Networking 3
- 1.2 Contributions 7
- 1.3 Methodology 8
- 1.4 Organization 9

2 Background and Related Work **10**
- 2.1 Challenges in Scalable NDN Forwarding 10
  - 2.1.1 Forwarding Information Base 11
  - 2.1.2 Pending Interest Table 12
  - 2.1.3 In-network Caching Elements 13
- 2.2 Research Problems and Design Considerations 14
  - 2.2.1 Research Problems 14
  - 2.2.2 Design Considerations 15
- 2.3 High-performance Packet Processing Techniques for IP Networks 16
  - 2.3.1 Memory Technologies 16
  - 2.3.2 Packet Processing Platforms 18
  - 2.3.3 Packet Processing Frameworks 20

3 Establishing a FIB Lookup Performance Baseline **21**
- 3.1 Introduction 22
- 3.2 Related Work 25
  - 3.2.1 Existing Name Prefix Lookup Solutions 26
  - 3.2.2 Binary Search of Prefix Lengths 28

3.3 Binary Search of Hash Tables . . . 28
3.3.1 Binary Search for Name Prefix Lookup . . . 29
3.3.2 Fingerprint-Based Hash Table . . . 33
3.4 Performance Evaluation . . . 38
3.4.1 Hash Table Performance . . . 38
3.4.2 Performance Comparison with Linear Search . . . 47
3.4.3 Multicore Performance . . . 49
3.4.4 System Evaluation . . . 51
3.5 Discussion . . . 53
3.6 Summary . . . 54

4 FIB Optimizations . . . 56
4.1 Level Pulling . . . 57
4.2 Speculative Forwarding . . . 60
4.3 Fingerprint-based Solutions to Enhance Scalable Name-based Forwarding . . . 62
4.3.1 Information-theoretic Difference Approach . . . 63
4.3.2 The String Differentiation Problem . . . 64
4.3.3 Fingerprint-based Patricia-trie . . . 66
4.3.4 Hash Table-based Methods . . . 69
4.3.5 Distributed Forwarding . . . 74
4.3.6 FIB Updates . . . 77
4.3.7 Experimental Evaluation . . . 81
4.3.8 Related Work . . . 84
4.4 Summary . . . 85

5 Pending Interest Table . . . 86
5.1 Introduction . . . 87
5.2 Background . . . 89
5.2.1 Design Considerations . . . 89
5.2.2 Hash-based Techniques . . . 90
5.3 Pending Interest Table Requirements . . . 90
5.4 Fingerprint-only Pending Interest Table . . . 92
5.4.1 Design Overview . . . 92
5.4.2 Data Structures . . . 95
5.4.3 Segregated Pending Interest Table . . . 97
5.4.4 Popular Content Optimization . . . 98
5.4.5 Analysis . . . 100
5.5 The Case with Multiple Core Routers . . . 109

5.5.1 Supporting Multiple Core Routers . . . 110
5.5.2 Analysis . . . 111
5.6 Performance . . . 112
5.6.1 Experimental Setup . . . 113
5.6.2 Simulation . . . 113
5.6.3 Latency Measurement . . . 114
5.7 Discussion . . . 115
5.7.1 Allowing False Positives . . . 115
5.7.2 Pending Interest Table Security . . . 116
5.8 Related Work . . . 117
5.9 Summary . . . 117

6 In-network Caching Elements . . . 119
6.1 Content Store . . . 120
6.1.1 Requirements . . . 121
6.1.2 Content Store Data Structures . . . 123
6.1.3 Supporting Cache Replacement Policies . . . 125
6.2 Repo . . . 128
6.2.1 Performance Evaluation . . . 132
6.3 Summary . . . 139

7 Conclusion . . . 141
7.1 Future Research Directions . . . 143
7.1.1 A Full-fledged Forwarding Engine . . . 143
7.1.2 Forwarding Information Base . . . 145
7.1.3 Pending Interest Table . . . 146
7.1.4 In-network Caching Elements . . . 146

References . . . 148

List of Figures

1.1 NDN Content Delivery Example 5
1.2 NDN Packet Operational Flow 6

3.1 Binary Search for Name Prefix Lookup 29
3.2 Example with Seven Forwarding Rules Stored in the FIB 30
3.3 Hash Bucket Memory Layout 34
3.4 Name Prefix Entry Memory Layout 36
3.5 System Architecture 39
3.6 Impact of Page Sizes 42
3.7 Impact of Prefetching Strategies 43
3.8 Lookup Performance for Datasets with Seven Components 44
3.9 Lookup Performance for Datasets with 15 Components 45
3.10 Lookup Latency with Linear Search 48
3.11 Multicore Performance (Prefetch All) 50
3.12 Simplified Packet Format 51
3.13 Experiment Configuration 52

4.1 Level Pulling Concept 58
4.2 Speculative Forwarding Illustration 61
4.3 Perfect Hashing and Non-perfect Hashing 65
4.4 Impact of Fingerprint Length on the Trie Depth 68
4.5 Patricia Trie-based Solution Lookup Performance 68
4.6 Fingerprint-based Hash Table 70
4.7 Hash Table Lookup Performance 73
4.8 Distributed Name-based Forwarding 74
4.9 Distributed Forwarding Memory Optimization 75
4.10 Memory Overhead of Distributed Forwarding 76
4.11 Patricia-trie Update 79
4.12 RIB Insertion Performance 80
4.13 Fingerprint-based Patricia Trie Lookup Performance 82
4.14 Fingerprint-based Hash Table Lookup Performance 83

5.1 System Design 93
5.2 System Operations 94
5.3 PIT Data Structures 96
5.4 Memory Requirement 104
5.5 Network Traffic Overhead 107
5.6 Supporting Multiple Core Routers 110
5.7 PIT Operation Latency 115

6.1 Content Table Entry 124
6.2 Doubly Linked List for the LRU Policy 126
6.3 Data Structures for the LFU Policy 127
6.4 Single-node Repo-ng Throughput Performance (Single Thread) 133
6.5 Single-node Repo-ng Throughput Performance (Two Threads) 134
6.6 Repo-ng Performance with Multiple Client Nodes 137
6.7 Redis Benchmark 138

List of Tables

2.1 Packet Forwarding in IP and NDN 14

3.1 System Configuration 39
3.2 Throughput with 256-Byte Packets 53

4.1 Existing Name Conversion Schemes 58
4.2 Hash Lookup Reduction Percentages 59
4.3 Real-world Dataset Characteristics 66
4.4 FHT Memory Requirements 72
4.5 Distributed Forwarding for One Billion Names 77

5.1 Pending Interest Table Requirements 91
5.2 Memory Requirement of Zipf-like Fingerprint Distributions 105
5.3 Traffic Overhead of Zipf-like Fingerprint Distributions 108
5.4 Fingerprint Collision Rates and Overflow Sizes 114

6.1 Netflix OpenConnect System Specification 139

Acknowledgments

First of all, I would like to express my deepest gratitude to my advisor Prof. Patrick Crowley for his invaluable support and guidance during my study at Washington University. He introduced me to the NDN project, which provided a unique opportunity for me to explore methods for scalable data structures that handle extremely large datasets. Along the journey, he has always been encouraging and provided insights to help me develop research ideas. I have always been impressed by how he reduces complex issues to simpler matters based on practical considerations.

I would also like to thank my other committee members, Prof. Roger Chamberlain, Roch Guerin, Raj Jain, Neil Richards, and Jonathan Turner, for their valuable feedback on my work. I would like to thank Prof. Jonathan Turner for discussing how the FIB updates, and for the valuable data structure and algorithm knowledge gained from his class.

It has been a rewarding experience to work on a large collaborative project with the faculty and students of the NDN team. It was valuable to learn perspectives from different research areas and work together on the open-source NDN forwarding daemon implementation.

I am grateful to have the opportunity to intern at Cisco, where I learned more about the NDN research work conducted in industry. I would like to express my sincere thanks to David Oran, Won So, Mark Stapp, and Ralph Droms.

I would like to thank the staff members at ARL, John DeHart and Jyoti Parwatikar. They have been very supportive and always helped get the testing environments ready for experiments.

I am grateful to James Ballard for help with editing all my manuscripts and providing valuable suggestions on writing.

I would like to thank my fellow students at ARL, including Charlie Wiseman, Mike Wilson, Ben Wun, Shakir James, Mart Haitjema, Michael Schultz, Jason Barnes, Hila Abraham, and Adam Drescher.

Thanks to all my friends for making the past few years in St. Louis enjoyable and memorable.

Finally, I am grateful to my parents and all the family members, who have always been supportive and patient.

Haowei Yuan

Washington University in Saint Louis

December 2015

This work is supported by National Science Foundation grants CNS-1040643 and CNS-1345282.

Dedicated to my parents, Zhiyong Yuan and Jinmei Tian.

ABSTRACT OF THE DISSERTATION

Data Structures and Algorithms for Scalable NDN Forwarding

by

Haowei Yuan

Doctor of Philosophy in Computer Engineering

Washington University in St. Louis, 2015

Professor Patrick Crowley, Chair

Named Data Networking (NDN) is a recently proposed general-purpose network architecture that aims to address the limitations of the Internet Protocol (IP), while maintaining its strengths. NDN takes an information-centric approach, focusing on named data rather than computer addresses. In NDN, the content is identified by its name, and each NDN packet has a name that specifies the content it is fetching or delivering. Since there are no source and destination addresses in an NDN packet, it is forwarded based on a lookup of its name in the forwarding plane, which consists of the Forwarding Information Base (FIB), Pending Interest Table (PIT), and Content Store (CS). In addition, as an in-network caching element, a scalable Repository (Repo) design is needed to provide large-scale long-term content storage in NDN networks.

Scalable NDN forwarding is a challenge. Compared to the well-understood approaches to IP forwarding, NDN forwarding performs lookups on packet names, which have variable and unbounded lengths, increasing the lookup complexity. The lookup tables are larger than in IP, requiring more memory space. Moreover, NDN forwarding has a read-write data

plane, requiring per-packet updates at line rates. Designing and evaluating a scalable NDN forwarding node architecture is a major effort within the overall NDN research agenda.

The goal of this dissertation is to demonstrate that scalable NDN forwarding is feasible with the proposed data structures and algorithms. First, we propose a FIB lookup design based on the binary search of hash tables that provides a reliable longest name prefix lookup performance baseline for future NDN research. We have demonstrated 10 Gbps forwarding throughput with 256-byte packets and one billion synthetic forwarding rules, each containing up to seven name components. Second, we explore data structures and algorithms to optimize the FIB design based on the specific characteristics of real-world forwarding datasets. Third, we propose a fingerprint-only PIT design that reduces the memory requirements in the core routers. Lastly, we discuss the Content Store design issues and demonstrate that the NDN Repo implementation can leverage many of the existing databases and storage systems to improve performance.

Chapter 1

Introduction

The remarkable success of the Internet in the past few decades rests largely on its layered architectural design. The simplicity of the Internet Protocol (IP) layer, which is the *narrow waist* of the hourglass-shaped Internet protocol stack, has enabled global connectivity. To date, although many new network protocols have been deployed at other layers, the IP layer is largely unchanged: Every network element is assigned an IP address, and each packet carries a source address and a destination address. In the IP layer, network devices need only to support packet forwarding based on the destination addresses to be connected to the global network. The design of the IP layer comfortably supports point-to-point communication, the major use of the Internet in its early days. However, today's Internet supports far more functions than what it was originally designed for, and it has effectively became a platform for information dissemination.

The original IP layer has several important limitations. For instance, when a large number of clients that are geographically close to each other request the same piece of content, such as a movie, this piece of content has to be transferred across the network multiple times so that every client receives it. To address this issue, web caching proxies and Content Delivery Networks (CDNs) are deployed in the current Internet so that content is stored

geographically closer to users. The current Internet relies heavily on middleboxes and layers of redirections, making the network less efficient. What is more, IP does not support secure information transfer efficiently. The point-to-point conversation model requires securing the communication channels, limiting the ability to cache and redistribute content by third-party middleboxes.

Named Data Networking (NDN) [90] is a general-purpose network architecture that aims to address the shortcomings of the Internet in its current usage, while maintaining its strengths. Specifically, NDN maintains a similar hourglass-shaped protocol stack, where the narrow waist supports a richer set of functions than IP, including efficient information dissemination and embedded data security. Compared to IP, which relies on middleboxes to support content distribution, NDN allows packets to be cached and redistributed by the forwarding nodes, supporting efficient content distribution in the network layer. In addition, authentication is also enabled in the network layer. Unlike IP, which secures communication channels, NDN secures content. NDN packets that carry content are signed by publishers, thus each packet can be validated individually. NDN also supports client-side mobility by design. The goals of NDN are to support more secure communications, more efficient use of underlying infrastructure, and simpler applications that enable new things. Despite these benefits, it remains unclear how well the NDN narrow waist can scale up and whether it can be implemented efficiently in practice.

This dissertation addresses these questions by exploring data structures and algorithms to support scalable NDN packet forwarding. We have implemented the proposed designs in software and demonstrated that 10 Gbps packet forwarding rates can be reached with longest name prefix lookups on general-purpose multicore platforms, using 256-byte packets.

Purpose-built hardware, more sophisticated processor architectures, and advanced memory technologies are expected to further improve the forwarding performance.

1.1 Named Data Networking

The Internet was originally designed to enable resource sharing between endhosts. Since its inception, the Internet has expanded rapidly, and the services provided by the Internet have grown explosively. We have observed the emergence of many popular Internet applications: simple resource sharing is joined by Email, HTML webpages, voice over IP, content delivery, and video streaming. Today, users are mostly interested in retrieving information from the network, and the Internet has effectively became an information dissemination platform.

The Internet infrastructure has also evolved towards an information-centric network with a layered approach. The IP layer enables network connectivity, and its design resembles conversations between endhosts. To forward an IP packet, the destination IP address is used as the key to perform a longest prefix match (LPM) in the *Forwarding Information Base* (FIB) of the network device. Today, the most commonly used method to retrieve information is by requesting its Universal Resource Locater (URL) [57]. URLs cannot be handled in the IP layer directly, but instead rely on redirections of the Domain Name System (DNS), which maps domain names to IP addresses. Essentially, the current Internet relies heavily on middleboxes and layers of redirections, making the network hard to configure and less efficient.

A number of clean-slate Internet architectures have been proposed to address the limitations of the current Internet by focusing on what the content is rather than where it is

from [32, 33, 36], and these proposed network architectures can be categorized as *information-centric networking* (ICN) [82]. NDN is a general-purpose network architecture that takes the information-centric approach. Instead of naming endhosts in IP, in NDN names are assigned to content. Content names are hierarchically-structured human-readable names, just like URLs we use today. Each name consists of multiple *name components*. For instance, an NDN packet name may look like `ndn:/wustl.edu/ndn/research`, where `wustl.edu`, `ndn`, and `research` are the name components. Compared to the forwarding plane functions in IP, the enriched NDN forwarding plane requires more operations, some of which may be challenging. Packets in NDN are forwarded based on name-lookup results. Each NDN forwarding node contains three components: the *Forwarding Information Base* (FIB), the *Pending Interest Table* (PIT), and the *Content Store* (CS). Two types of packets exist in NDN networks, namely, *Interest* packets and *Data* packets. To fetch content, a content consumer sends an Interest packet, which contains the name of the requested content. A Data packet, which contains both the content and its name, will be returned if the requested content is available in the network. Large files can be chunked and thus fetched, cached, and redistributed at granularities smaller than the sizes of files and data objects. NDN Data packets can be cached and redistributed by in-network caching elements, including both the Content Store and *Repository* (Repo). The CS is a short-term content storage residing on NDN forwarding plane, while the Repo is a long-term persistent content storage. A Repo can serve as a database for a specific application and run on the same machine as the application; it can also serve as a content distribution node and run on a dedicated machine or even a cluster.

Figure 1.1 shows an example of retrieving content in NDN. Alice and Bob are connected to the Internet via the same gateway router. The content they are interested in, with the name `ndn:/wustl.edu/ndn/1`, resides in the server or the Repo that belongs to the content

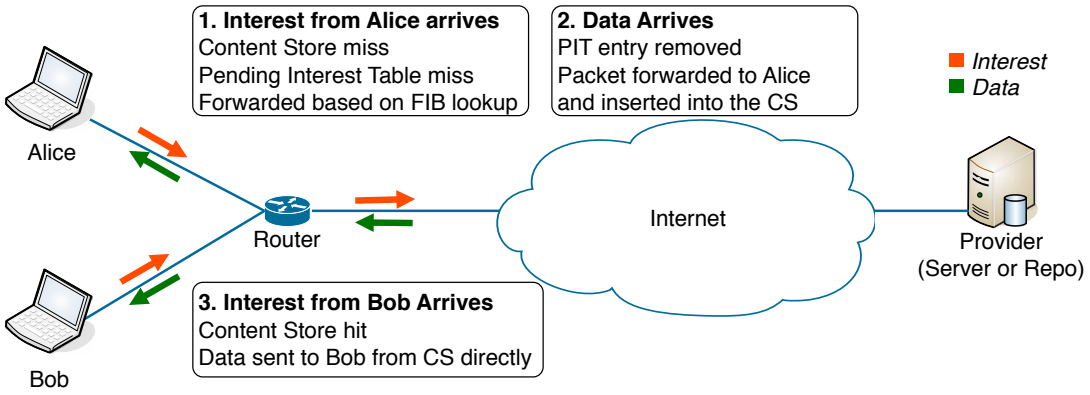


Figure 1.1: NDN Content Delivery Example

provider. In this example, Alice sends out the Interest packet first, and Bob waits until Alice has received the Data packet to send his Interest packet. When the Interest packet sent from Alice arrives at the gateway router, the router first queries the Content Store, which essentially is a cache for Data packets, and sees if the requested content is already cached. In this example, there is no such content in the CS. Then the router looks up the Pending Interest Table to see if there is already a request on the fly for this content. In this example, there is no such request either. As a result, the Interest packet is forwarded based on the FIB lookup results, and the incoming interface and the name of the requested content are recorded in the PIT. As there is no cached content for this request in the network, the Interest arrives at the content provider, and the Data packet is sent back. When the Data packet arrives at the gateway router, the router first checks whether a pending PIT entry is waiting for this Data packet. If so, the Data packet is forwarded to the incoming interface(s) and the PIT entry is removed. Then the Data packet is cached in the Content Store. When Bob sends the Interest packet, there is a CS hit at the gateway router. As a result, the Data packet is delivered from the gateway router to Bob directly. Obviously, in this simple example, the requested content is sent from the content provider only once.

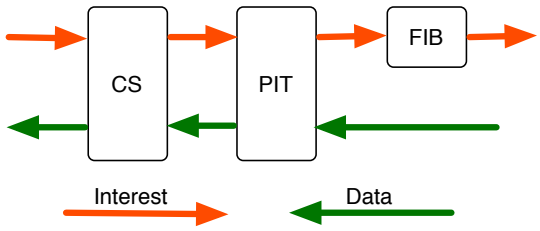


Figure 1.2: NDN Packet Operational Flow

The operational flows of the Interest and Data packet processing in the NDN forwarding plane are shown in Figure 1.2. On the arrival of an Interest packet, the CS is looked up first to see if this Interest request can be satisfied by a cached Data packet. If yes, the Data packet is sent in reply immediately. Otherwise, the Interest packet is forwarded based on the FIB lookup result if the content name is not found in the PIT, then the packet name and the incoming interface of this Interest packet are recorded in the PIT. On the arrival of a Data packet, the PIT is looked up to retrieve the outgoing *faces*, where the notion of face in NDN defines a network interface. Then this Data packet is delivered, and also inserted into the CS if it has not been cached.

A scalable NDN forwarding plane is a key to enabling large-scale NDN deployment and demonstrating its feasibility. This dissertation focuses on issues in scalable NDN forwarding, and we propose data structures and algorithms to support efficient name prefix lookup mechanisms. We also implement the design in software and experimentally demonstrate that scalable NDN forwarding is feasible.

It is important to note that NDN can be used today without requiring that all of today's IP infrastructure be discarded. When discussing any future Internet architecture, it is easy to dismiss any new idea on the basis that there is no way to displace today's Internet and its protocols. IPv6 is a good example of how hard it is to replace IP. So, it is important to

recognize that NDN does not need to completely replace IP in order to be helpful. In the same way that IP operated initially as an overlay network atop the telephone system, NDN operates well as an overlay atop IP. In today's NDN deployments, including the global NDN testbed [46], NDN runs over IP links because IP provides universal connectivity.

1.2 Contributions

Specifically, we make the following contributions in this dissertation.

- We propose a scalable FIB longest name prefix lookup design based on the binary search of hash tables. We have implemented the proposed design in software and demonstrated 10 Gbps forwarding throughput with 256-byte packets and one billion synthetic longest name prefix matching rules, each containing up to seven name components. At the time of writing, this is still the largest dataset that has been studied for longest name prefix lookup.

- As in IP FIB lookup, the NDN FIB lookup performance can be optimized based on the characteristics of the specific forwarding rules. For real-world datasets, such as the Alexa [2] and Dmoz [21] domain names, we first propose a general level-pulling method to reduce the average number of lookups, and then propose memory-efficient data structures to reduce the memory requirements of the FIB.

- We propose a fingerprint-only PIT design that reduces the memory requirements by relaxing the Interest aggregation feature in the core networks.

- As for in-network caching elements, we discuss the Content Store design issues and evaluate the performance of a modified NDN Repo based on the Redis [59] key-value store. We show that existing storage systems and databases can be employed in NDN.

1.3 Methodology

The research methodology employed in this dissertation involves a combination of analysis, simulation, and empirical evaluation of software implementations.

To evaluate the proposed designs, we use the memory requirements of the data structures and the forwarding throughput as the performance metrics. For single-threaded implementations, we also use lookup latency as a metric because it is inversely proportional to the forwarding throughput.

Unlike in IP forwarding, where real-world datasets are already available, large scale NDN forwarding rules are not available yet. As a result, we first provide a reliable performance baseline using large-scale synthetic datasets as the workload, and then propose optimization methods based on characteristics of real-world datasets that have been reported in literature.

The proposed data structure and algorithm designs can be implemented in both software and hardware. In this dissertation, we use software-based implementation to demonstrate their performance.

1.4 Organization

The rest of the dissertation is organized as follows. Chapter 2 provides more background on the NDN forwarding plane and reviews existing technologies for scalable packet processing. Chapter 3 presents a scalable longest name prefix lookup design based on the binary search of hash tables. Chapter 4 focuses on FIB optimizations based on the specific characteristics of real-world forwarding datasets. Chapter 5 focuses on the PIT design and presents the proposed fingerprint-only PIT. Chapter 6 focuses on the design of in-network caching elements. Finally, Chapter 7 concludes the dissertation and discusses remaining research issues in scalable NDN forwarding.

Chapter 2

Background and Related Work

In this chapter, we discuss the challenges in scalable NDN forwarding, outline the research problems and design considerations, and then review state-of-the-art high-performance packet processing techniques for IP networks.

2.1 Challenges in Scalable NDN Forwarding

Each NDN forwarding node consists of three major components, the Forwarding Information Base, Pending Interest Table, and Content Store. These three logically separated structures have distinct performance requirements. In addition, the NDN Repo is also a key component in NDN networks and requires scalable designs and implementations. In this section, we discuss the challenges for the FIB, PIT, and in-network caching elements, which include both the Content Store and NDN Repo.

2.1.1 Forwarding Information Base

As in IP packet forwarding, the FIB in NDN stores the forwarding information for Interest packets. In IP, the FIB table is keyed by IP prefixes, and each FIB entry stores only one outgoing port. Each IP packet is forwarded based on the FIB *longest prefix matching* (LPM) result, with the destination IP address as the lookup key. Because the content names in NDN are hierarchically structured, the Interest packets are forwarded based on *longest name prefix matching* (LNPM) results, with the requested content names as the lookup keys. The FIB in NDN is keyed by name prefixes. In addition, because NDN intrinsically supports multi-path forwarding for efficient content fetching, each FIB entry can store information about multiple outgoing ports. As in IP, the FIB is populated by the Routing Information Base (RIB), which is constructed based on the content availability information exchanged via routing protocols. In this dissertation, we focus on efficient FIB lookup mechanisms and assume the forwarding table has already been populated.

The longest name prefix matching problem is complex because name prefixes are longer than IP addresses and the namespace is unbounded. First, NDN names are of variable length and can be much longer than fixed-length IP addresses. The complexity of many IP forwarding solutions is proportional to the length of prefixes. As a result, directly applying those schemes yields lower performance. Second, 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 [14]. For NDN, because real-world FIB datasets are not available yet, we use the DNS, which is the largest namespace in the Internet, as an example to show the potential size of the NDN FIB. For instance, there were already 271 million registered domain names in the Internet in 2013 [73]. What is more, there were 968 million host names in the network, and 181 million of them were active [28]. Although the number

of rules in the FIB table is determined by the namespace design and the effectiveness of name prefix aggregation, millions of domain names are expected to be in the FIB in order to handle a network on the scale of the current Internet.

In this dissertation, our goal is to demonstrate 10 Gbps forwarding throughput with one billion (10^9) forwarding rules.

2.1.2 Pending Interest Table

The PIT keeps track of the currently unsatisfied Interest packets. Arriving Interest packets are forwarded to the next hop based on a FIB lookup only if the PIT finds no pending Interest packet with the same name. The PIT also records the unique incoming interfaces of each requested content name so that it keeps the destination information for Data packets. For each Data packet, the PIT is queried to find the incoming face(s) that requested the content, and then the Data packet is delivered and its content name is deleted from the PIT. Hence, the PIT requires per-packet updates, including memory writes. In the worst case, every arriving packet, whether an Interest or Data packet, requires an update operation.

As the link speed keeps increasing, the processing time available for each packet at the PIT is reduced. Moreover, the PIT memory size becomes larger as the link speed increases. The content name needs to be stored in each PIT entry. The names are similar to URLs, and today's URLs typically require tens of bytes of storage. For example, the URLs for the pictures and videos on popular social networking websites, which include long hash numbers, are more than 80 bytes long. Moreover, there are websites that include the entire article names in the URLs, making the URLs longer. Larger memory requirements limit the ability

to employ high speed memory devices. As a result, a fast and scalable PIT design is also demanded.

2.1.3 In-network Caching Elements

The In-network caching elements include both the Content Store, a temporary storage like packet buffers in IP networks, and the Repo, a long-term storage for the content.

The Content Store essentially stores packet buffers, and it is equipped with a lookup structure and supports cache replacement policies. Because cache replacement policies, such as least recently used (LRU), can be implemented efficiently, the requirements of the CS are similar to those of the PIT.

Unlike the CS, which resides in the data plane, the Repo is much larger in size and runs as a service. The Repo registers name prefixes to neighboring NDN routers so that Interest packets that match those name prefixes are forwarded to the Repo. The announced prefixes also include control prefixes, and applications can send control commands to request the specified content to be fetched and stored in the Repo. The Repo can serve as a database for an application. Alternatively, it can serve as a content distribution node in the current content delivery networks (CDNs). Because CDNs and large-scale key-value stores have been shown to be scalable, in this dissertation, we focus on demonstrating that these techniques can be used to implement the backend storage for the Repo.

2.2 Research Problems and Design Considerations

2.2.1 Research Problems

Compared with IP forwarding, NDN forwarding is complicated and more difficult. Table 2.1 lists the differences between IP packet forwarding and NDN packet forwarding. For these reasons a scalable NDN forwarding node architecture is a considerable challenge.

Table 2.1: Packet Forwarding in IP and NDN

	IP	**NDN**
Forwarding Key	IP address	Content name
Key length	32 bits	Variable
Forwarding rules	LPM	LNPM
Per-packet READ	Yes	Yes
Per-packet WRITE	No	Yes

Considering the above, we have identified two key research problems in scalable NDN forwarding.

- *Longest prefix matching for variable-length and unbounded names.* In longest prefix matching (LPM), there is a lookup key k and a set of strings *Set*. The problem is to find a string s from *Set* such that s is the longest prefix of k among all the strings in *Set*. Longest prefix match is performed in FIB lookups. Because generally the forwarding rules are not updated very frequently, the implementation of LPM should mainly focus on fast lookup of variable-length and unbounded names. Existing IP LPM solutions cannot be readily applied since their complexity is generally proportional to the length of the rules. In NDN, packet names are much longer than 32-bit long IP addresses.

- *Exact string matching with fast updates.* In exact string matching, there is also a lookup key k and a set of strings stored in *Set*. The problem is to verify whether k is in *Set*, and then to perform operations such as insert, delete, or update values in *Set*. In the NDN forwarding plane, exact string matching with fast updates is performed in PIT lookups and CS lookups. A PIT lookup inserts a new Interest packet, updates an existing PIT entry, or deletes a satisfied Interest packet. For a Data packet, a Content Store lookup results in inserting this Data packet into the CS or updating the expiration time of a stored Data packet. In the worst case, every packet requires an update operation.

2.2.2 Design Considerations

Three design considerations guide our design.

- *The characteristics of NDN name structures can be leveraged.* NDN names contain explicit delimiters that separate their components. The cardinality of name components at each component level is infinite, which 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. As a result, name prefix lookup algorithms can focus on processing one name component in each step.

- *Simple data structures allow fast updates.* Simple data structures include hash tables, d-left hash tables, and counting Bloom filters [34]. Tree- or Trie-like data structures may need readjusting in order to balance the data structure following an update, and thus may not be as efficient as hash-based solutions.

- *Edge routers and core routers have different requirements.* Since edge routers are closer to the content consumers, sophisticated PIT and CS lookups can be performed, such as the *all prefix match* in the PIT and CS lookup, which performs lookups with all the proper prefixes of the content name carried in the incoming Data packet and attempts to consume as many pending Interests as possible [89]. However, in the core routers, the PIT and CS have higher link rate requirements, and thus simple exact string matching can be employed instead. The Content Store and other in-network caching elements can be very large for edge routers, while the size of the CS in the core routers may be small because intuitively the more mixed traffic in the core routers degrades the effectiveness of caching. A recent study also shows that performing only edge caching could achieve the considerable benefit of information-centric network design [24].

2.3 High-performance Packet Processing Techniques for IP Networks

Significant research has been devoted to scalable packet processing in IP networks. In this section, we review state-of-the-art techniques for scalable packet processing, including memory technologies, packet processing platforms, and packet processing frameworks. The related work for each NDN forwarding structure is presented in the later chapters that describe specific designs.

2.3.1 Memory Technologies

Memory technologies have considerable impact on packet forwarding performance because the lookup data structures and packet buffers are stored in memory and they

are frequently visited. In high-performance packet processing research, typically three types of memory devices have been used, namely *Dynamic Random-Access Memories* (DRAMs), *Static Random-Access Memories* (SRAMs), and *Ternary Content Addressable Memories* (TCAMs).

DRAMs are the most commonly used memory devices. DRAMs cost significantly less than TCAMs and SRAMs, but DRAMs have relatively higher memory access latencies. Servers nowadays can be equipped with hundreds of gigabytes of DRAM. For packet processing, DRAMs have been used as packet buffers and to store lookup structures when the structure cannot fit into high speed memory devices, such as SRAMs and TCAMs.

SRAMs have shorter memory access latency, however, SRAM resources are limited: The largest single-chip SRAM device has 9 MB [53]. In packet processing applications, SRAMs have been typically used to store entire lookup structures or to store filters to reduce the number of accesses to DRAMs, which store the complete lookup structure. SRAMs are also used to implement L1 caches in general-purpose processors. Recent works [5, 83] have focused on reducing the sizes of IP lookup data structures so that they fit into CPU caches.

TCAMs support fast longest prefix match because the lookup key, i.e., the destination IP address, is matched against all of the stored entries at once. TCAMs are used in high-performance router implementations to support high-speed longest prefix match. Similar to SRAMs, TCAMs have limited total storage space. Also, the width of each TCAM entry is limited, thus multiple cycles are required to return the final lookup result if the key is long. What is more, because overlapped prefixes have to be kept in sorted order in TCAMs to support longest prefix matching, updates are relatively slow [62].

2.3.2 Packet Processing Platforms

Packet processing platforms can be categorized as either hardware-based or software-based. Generally, hardware-based platforms support higher performance but are difficult to program, while software-based platforms are easier to program but sacrifice the performance.

Hardware-based Platforms

Purpose-built hardware employs application-specific integrated circuits (ASICs), embedded high-speed memory devices, and pipelined stages to improve packet processing throughputs. In addition, the I/O can be configured to support higher link rates, which generally is not possible in software-based platforms. Despite the performance gains, the packet processing logic cannot be easily changed after the hardware is manufactured, thus limiting its ability to support new applications. Moreover, the design cycles of purpose-built hardware are long.

To address this issue, reconfigurable hardware devices, such as *Field-programmable Gate Arrays* (FPGAs), have been employed for high-performance packet processing. The ability to reconfigure the packet processing logic enables fast application prototyping.

In hardware-based designs, packet processing applications can be pipelined to improve the throughput. As a result, the most time-consuming pipeline stage determines the system throughput. In IP networking, memory accesses are typically the bottleneck because the memory size, memory bandwidth, and the number of memory controllers are limited on these platforms.

Software-based Platforms

Compared to hardware-based platforms, *network processors* provide a more developer-friendly programming environment to support configurable packet processing with advanced processor architectures. For example, the IXP network processor provides a C-like programming environment and contains 16 micro engines for high-speed packet processing. Each micro engine supports up to eight hardware threads, which take only four cycles to perform a context switch. The highly parallel design makes it possible to hide memory access latency. In addition, specialized hardware units, such as high-speed memory devices, including SRAM and TCAM, and dedicated hashing units, are available on network processors.

General-purpose multicore processors have recently regained popularity for high performance packet processing research. The performance of general-purpose processors has increased considerably: more cores are integrated within each processor, hyper-threading is supported on each physical core, memory controllers are embedded in the processors, and the traditional font-side bus has been replaced by more advanced networks, such as the Intel QuickPath Interconnect [4]. In other words, the current general-purpose multicore processor architectures have characteristics that once were available only from network processors.

Graphics processing units (GPUs) have also been employed for packet processing [26]. Their massively parallel processing power and ample memory bandwidth permit considerable performance improvement for memory-intensive applications.

In software-based designs, a hybrid of pipelining and thread-level parallelism is typically applied to improve the system performance.

2.3.3 Packet Processing Frameworks

Developing high-performance packet processing applications requires a concrete understanding of performance optimizations at multiple abstraction levels in the entire system. Various platforms and frameworks have been proposed to reduce the complexity of the development procedure and support rapid application prototyping.

NetFPGA is an FPGA-based packet processing platform. Multiple reference applications, such as an IPv4 router, are available to speed up the learning curve for developers. The NetFPGA family includes multiple versions, supporting line rates at 1 Gbps, 10 Gbps, and 100 Gbps. The most advanced version, NetFPGA SUSE [92], is equipped with a Xilinx Virtex-7 690T FPGA device, 27 MB of SRAM, and up to 32 GB of DRAM.

With the advances in general-purpose multicore processors, several software-based packet processing frameworks have been proposed, such as the netmap [61], the Intel *Data Plane Development Kit* (DPDK) [29], and the PF_RING ZC developed by ntop [50]. These frameworks support user-space packet processing and bypass the generic but inefficient kernel stack. Hardware features provided by the network interface card (NIC), such as zero copy and multi-queue (a.k.a. receive side scaling) techniques, are also leveraged by these frameworks. Among these frameworks, DPDK provides additional support for building multicore packet processing applications.

In this dissertation, DPDK is used as the packet processing framework because the organization of the provided sample load balance applications suits our needs and the platform is easy to program.

Chapter 3

Establishing a FIB Lookup Performance Baseline

FIB lookup is a core building block of the NDN forwarding plane. In IP, FIB lookup is a well-studied problem, and several IP lookup solutions take advantage of FIB characteristics, such as the prefix length distribution. Similarly, FIB characteristics should also be taken into consideration when designing scalable NDN FIB lookup methods.

FIB characteristics are determined by namespace designs. In NDN, each application has its own namespace, thus NDN FIBs will have characteristics based on applications. For example, we can understand what a WWW-like namespace would look like because the web exists, and therefore we can talk about a corresponding FIB with confidence. Other namespaces are possible, however, like the one used in the authenticated lighting control application [11], that may or may not be wide-area and may or may not look like the web. Most importantly, we cannot anticipate what namespaces might be invented in the future; the goal for NDN is to provide an architecture that will support them all, regardless of what they turn out to be. As for scalable NDN forwarding, establishing a reliable FIB performance baseline that supports any kind of namespace is desired at the current research stage.

In this chapter, we present a longest name prefix matching (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 specific characteristics of the FIB. We implemented the design in software and demonstrated 10 Gbps throughput with 256-byte packets and one billion synthetic longest name prefix matching rules, each containing up to seven components.

3.1 Introduction

To achieve an efficient NDN narrow waist design, scalable name prefix lookup solutions are required. In this chapter, we focus on the following question: 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? When considering large-scale namespaces, for example at the scale of the world-wide web, we make the assumption that the routing protocols that configure NDN FIBs are similar to those that configure IP FIBs, so the rates of updates are expected to be similar. As a result, we focus on the FIB lookup performance in this chapter. FIB update issues, as well as a linear search method that supports FIB updates better, are also discussed.

Various naming schemes have been proposed in ICN, such as flat self-certifying names and hierarchical human-readable names [25]. We focus on 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. Recall that each NDN name consists of multiple variable-length *name components*. For instance, the name `/a/b/c/` has three name

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

components delimited by '/', a/, b/, and c/; and its name prefixes are /a/, /a/b/, and /a/b/c/.

Several recently proposed name prefix lookup solutions [54, 65, 79, 80] 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 [2], Dmoz [21] domain names, the URL blacklist [70], the IRCache traces [30], or crawled URLs [79]. 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 is 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 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 chapter, 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 [74]. 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 as of this writing.

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 [65, 91]. 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 DPDK 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.

Specifically, we make the following contributions in this chapter:

- **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 NDN forwarding 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 256-byte packets and 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].

3.2 Related Work

In this section, we review recently proposed name prefix lookup solutions and provide more background on binary search of prefix lengths.

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

3.2.1 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 [12, 65, 80] choose fingerprint-based designs, because string comparison is required only if the fingerprints match. These solutions differ mostly in the prefix-seeking strategy. The CCNx prototype [12] starts with the full name and then eliminates one component each time if there is no match in the FIB. Cisco's solution [65] 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 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. [80] proposed a greedy prefix-seeking strategy, so that more populated levels are looked up first. The solutions proposed in [65, 80] 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 [65] has a relatively better guarantee, which requires either M or $MD - M$ lookups. In addition, the hash table in [80] 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 [54], aims to store 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 [54], with the tested 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 [77] 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 [79]. The solution proposed in [79] 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 [44] has also employed trie-based solutions.

Besides the recently proposed name prefix lookup solutions, CuckooSwitch [91] 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 [91], only exact match was performed, and no string matching was involved because the lookup keys were MAC addresses.

3.2.2 Binary Search of Prefix Lengths

The original paper [74] 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 [58].

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 aiming 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 [65], while the largest dataset evaluated in this dissertation has one billion rules.

3.3 Binary Search of Hash Tables

In this section, we first describe how binary search of hash tables works for name prefix lookup, and then present the proposed fingerprint-based hash table design.

3.3.1 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 [74].

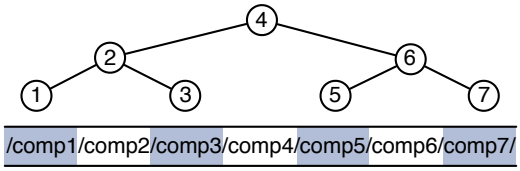


Figure 3.1: Binary Search for Name Prefix Lookup

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 entries, the lookup procedure can be terminated immediately if a matching leaf entry is encountered. Figure 3.1 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. Figure 3.2 shows an example where seven forwarding rules are stored. The highlighted path shows the lookup procedure for the query string `/a/b/c/d/e/ndn`, where the longest matching prefix is `/a/b/c/d/e/` in hash table node 5.

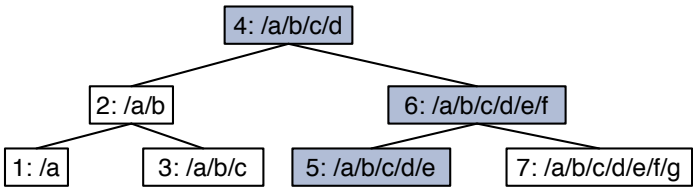


Figure 3.2: Example with Seven Forwarding Rules Stored in the FIB

As shown in [74], 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 3.1, 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 [74]. 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 [65, 80] also need to insert additional prefixes into the hash tables.

It has also been noticed that adding markers directly introduces the backtracking problem in [74]. 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 [74], 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/`.

FIB Updates

The forwarding information base needs to be updated when the routing information changes. For instance, when network links become up or down, or when new publishers become available, the corresponding FIB entries need to be updated. Generally, the FIB update information is determined by the Routing Information Base (RIB), which is maintained by a Router Controller (RC) in modern routers. Here, we discuss issues related to FIB updates.

Updating the FIB for the binary search of hash tables is a known issue [75]. As with the original binary search of hash tables designed for IP addresses, the primary challenge comes from inserting or deleting name prefixes that are proper prefixes of marker entries. For instance, assume that prefixes `/a` and `/a/b/c` are the forwarding rules that need to be stored in the binary search of hash tables. Following the procedure described in Section 3.3, the prefix `/a` is inserted into hash table 1, and the prefix `/a/b/c` is inserted into hash table 3.

In addition, an additional marker entry `/a/b` needs to be inserted into hash table 2, and its forwarding information is inherited from the prefix `/a`. If the forwarding information for the prefix `/a` is updated, then the forwarding information stored for the prefix `/a/b` needs to be updated. Similar requirements hold when a prefix that is a proper prefix of a marker entry needs to be inserted or deleted. The key problem is how to find the related marker entries efficiently.

The original paper first proposed several simple solutions which can be directly applied for name prefix lookup [75]. The first solution requires no additional data structure: When a FIB entry is updated, all of the entries that have longer prefixes can be enumerated to find the matching marker entries, and the forwarding information of these marker entries is updated. Obviously, this solution requires $O(nlog(k))$ complexity, where n is the total number of FIB entries and k is the maximum number of name components, because all of the entries need to be visited in the worst case. The second solution requires maintaining an additional data structure to store the marker entries associated with each prefix. This data structure, for instance, could be a trie maintained by the router controller as part of the RIB. However, the worst-case complexity is still $O(nlog(k))$ because all of the marker entries could be associated with a prefix, although this is unlikely to happen in practice.

The original paper also proposed a marker partition method [75] so that the forwarding information can always be fetched correctly for the marker entries with at most one additional memory access. Essentially, marker entries can store a memory address instead of the actual forwarding information, and this memory address leads to the structure that holds the actual forwarding information. The key optimization method is that, markers can be grouped into partitions, and markers in the same partition can share the same memory address of the forwarding information structure. This way, when the forwarding information of a prefix

is updated, only the shared memory addresses of the partitions need to be updated. We believe similar approaches can be explored for name prefix lookup, but the detailed design is beyond of the scope of this dissertation.

In general, FIB update operations have looser latency requirements than FIB lookup operations. As a result, when the number of marker entries associated with each prefix is small, the simpler solutions discussed above could meet the requirements. When the FIB update operations is extremely frequent, besides further exploring marker partition schemes for name prefix lookup, other update-friendly FIB lookup schemes can be employed, such as the linear search method to be presented in Section 3.4.2.

3.3.2 Fingerprint-Based Hash Table

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

Hash Table Memory Layout

Hash tables have been widely used in network applications [34]. 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 [10] or Cuckoo hash tables [52], 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 [65], 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.

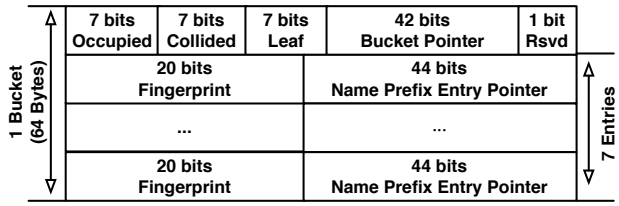


Figure 3.3: Hash Bucket Memory Layout

Hash bucket design. The organization of the hash bucket is shown in Figure 3.3. 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 3.3.2, 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.

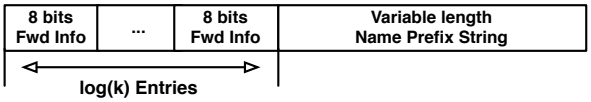


Figure 3.4: Name Prefix Entry Memory Layout

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 3.4, 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.

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 [65]. 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 3.4.1, 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 [15], 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 [54, 65]. 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 [65], SipHash [63] is one such hash function that can be employed.

3.4 Performance Evaluation

The performance study aims to demonstrate that the presented design supports large FIB tables efficiently in software with modest performance optimization. In this section, we present the performance micro-benchmarking of the hash table-based implementation, compare the performance of binary search to that of linear search, report the name lookup performance on a general-purpose multicore platform, and then demonstrate 10 Gbps forwarding throughput with real network traffic in the end.

3.4.1 Hash Table Performance

Experimental Setup

The experiments in this chapter 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 3.5 and Table 3.1. All of the experiments were performed within the DPDK environment, which provided huge page memory allocation supports.

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

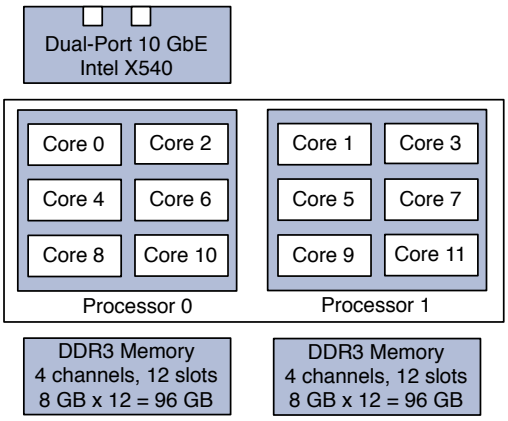


Figure 3.5: System Architecture

Table 3.1: 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

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 [77, 65]. 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 chapter, we always present performance results of datasets with up to seven name components, except in Section 3.4.1, 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.

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 [6]. 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 for large page memory, which was equally distributed between two NUMA nodes. We ran each experiment five times, and the average lookup latencies together with 95% confidence intervals are shown in Figure 3.6. It is worth noting that lookup latency is used as the metric here because it is the time, in terms of CPU cycles, spent for processing each packet, and it is inversely proportional to the forwarding throughput.

In all of the presented cases, the lookup latency was reduced significantly (about 31% to 48%) 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

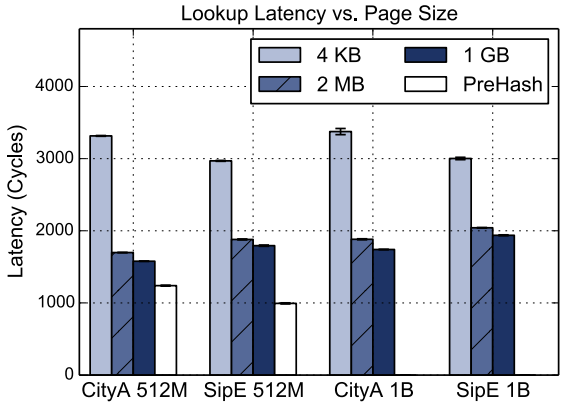


Figure 3.6: Impact of Page Sizes

likely due to the small number of TLB entries when 1 GB pages are 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 this chapter, 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 3.6. 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 [15], and we modified the SipHash reference implementation [63] so that all of the k hash values can be computed in one pass, as suggested

in [65]. 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.

Software Prefetching

Software prefetching has been demonstrated to be effective for accelerating hash lookups. In [65], the hash buckets corresponding to the prefixes of the querying name are all prefetched. In [91], 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.

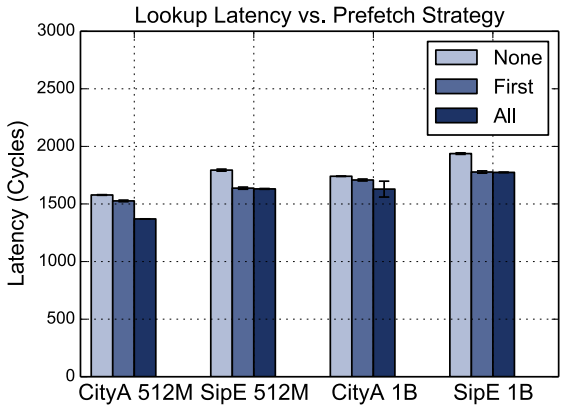


Figure 3.7: Impact of Prefetching Strategies

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. For each configuration, the experiment repeated five times. The average performance results and 95% confidence intervals are shown in Figure 3.7. For CityA, prefetching the first visited bucket achieved about 3% and 2% latency reduction for 512M and 1B, respectively. Prefetching all of the buckets reduced the lookup latency by 13% and 6% 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 9% for both 512M and 1B.

Performance with Various Datasets

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

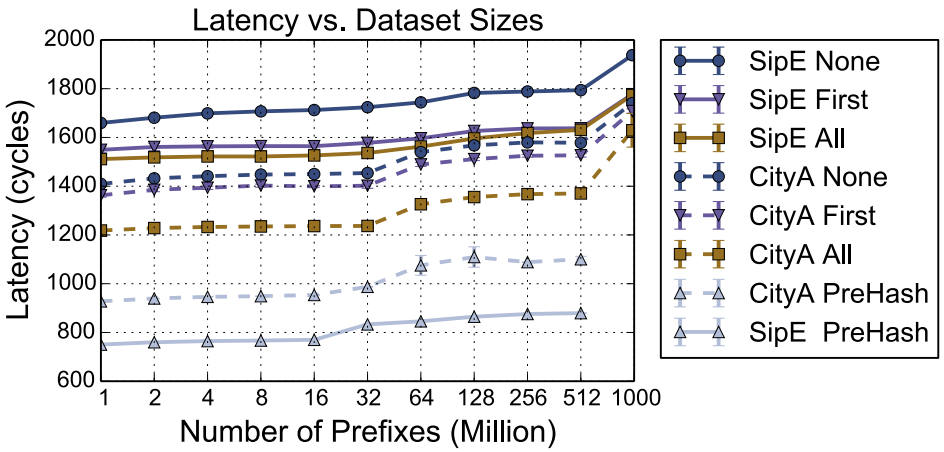


Figure 3.8: Lookup Performance for Datasets with Seven Components

Various dataset sizes We present the performance of both CityA and SipE with different dataset sizes and prefetching strategies in Figure 3.8. As before, we ran each experiment five times, and the average lookup latencies together with 95% confidence intervals are shown. 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.

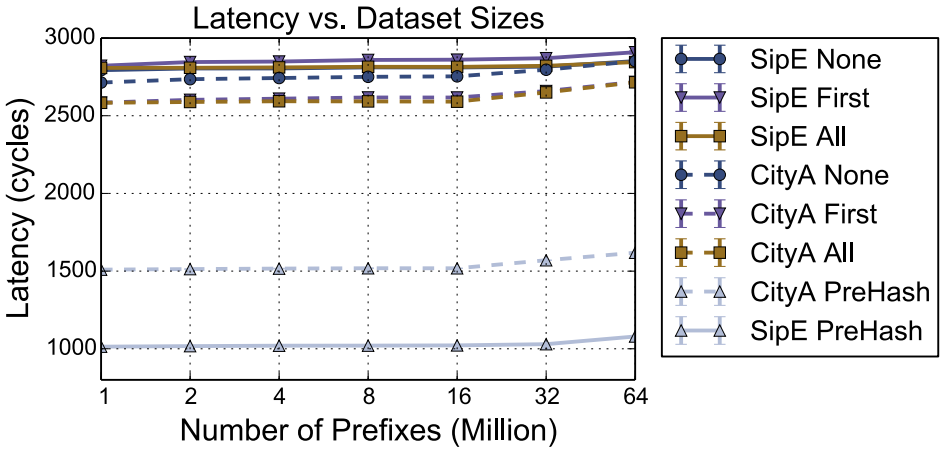


Figure 3.9: Lookup Performance for Datasets with 15 Components

Datasets with 15 **name components.** We present the performance with name prefixes that have 15 name components in Figure 3.9. 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 3.9, 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 related operations, such as deriving the memory addresses from the hash values, 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 3.9, when hash values were

precomputed, the lookup latencies were much closer to the expected values, determined by $log(k)$ number of hash lookups.

3.4.2 Performance Comparison with Linear Search

It is not easy to perform a head-to-head comparison between the proposed longest name prefix lookup design and existing solutions proposed by others, because none of the existing solutions have performed experiments on datasets with up to one billion forwarding rules, and they do not focus on worst-case performance. To demonstrate the effectiveness of the proposed design, we have measured the longest name prefix lookup latency with linear search, which requires $O(k)$ string lookups in the worst case.

In our implementation, we start from looking up the first name component at hash table 1, which stores name prefixes with only one name component. Then we increase the number of name component in the lookup key by one and query the hash table that stores prefixes with the same number of name components. We continue this process until there is a mismatch or a leaf entry is visited. Note that it is possible to reduce the number of string comparisons by starting from the last hash table, and then reduce the number of name components by one each time, but it requires generating lookup names that match only the first name component. The performance results are shown in Figure 3.10.

According to Figure 3.10, the performance of linear search is worse than that of binary search, because more memory addresses need to be visited. The performance difference between linear search and binary search is greater when CityHash is used, and less when SipHash is used. This is expected because CityHash requires performing string matching

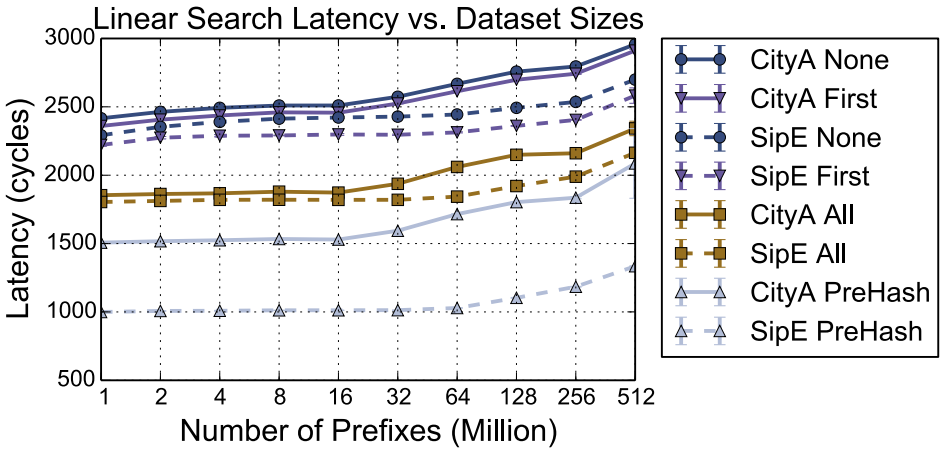


Figure 3.10: Lookup Latency with Linear Search

whenever there is a match, while SipHash requires performing string matching only at the end.

Comparing Figure 3.8 and Figure 3.10, when all of the hash buckets are prefetched, for the case with SipHash, the performance differences between linear search and binary search are small because all of the visited memory locations have already been prefetched. Although all of the hash buckets are prefetched into the cache, the lookup algorithm still needs to go through the cache line to validate if there is a match or not. Similarly, when hash values are precalculated, the performance differences between binary search and linear search are also small for the case with SipHash.

In addition, as mentioned in Section 3.3.1, the linear search method supports FIB updates better. There are two lookup schemes for linear search: First, if the linear search scheme starts from the first hash table, i.e., from the shortest prefix, then additional marker entries are still required to ensure a longer matching prefix can always be found. But the marker entries do not store inherited forwarding information because the backtracking problem no

longer exists. A reference counter can be maintained for each name prefix, so that a prefix is deleted only if it is not a matching prefix or a marker entry. Second, if the linear search scheme starts from the last hash table, i.e., from the longest prefix, then marker entries are no longer required. Name prefixes can be inserted into or deleted from the corresponding hash table directly. As a result, if FIB updates become very frequent, the linear search method assisted by hardware-accelerated hash computation and memory prefetching can be applied.

In summary, the lookup performance of binary search of hash tables is still better than the that of the linear search method, although the performance differences become smaller when aggressive memory prefetching strategies are applied.

3.4.3 Multicore Performance

In this section, we present the performance of the design on the multicore platform. In the experiments, similar as in [65], 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.

Each experiment was repeated five times, the average performance results and the 95% confidence intervals are shown in Figure 3.11. 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 3.11, the throughput increases proportionally to the number of threads.

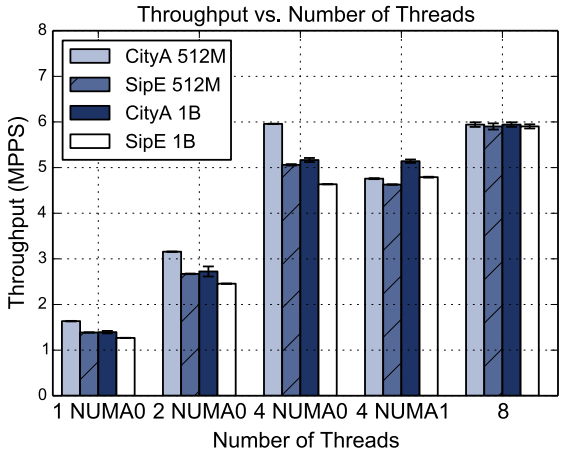


Figure 3.11: Multicore Performance (Prefetch All)

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 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. A plausible explanation is that some resource contention happened between NUMA nodes.

3.4.4 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 [29]. The forwarding engine performed only longest name prefix lookup.

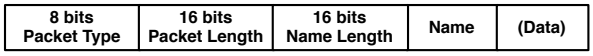


Figure 3.12: Simplified Packet Format

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 3.12. The Packet Type field, which takes eight bits, indicates if this is an Interest packet or a Data packet [90]. 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 packet I/O and a multicore framework for fast packet processing applications. We modified the existing DPDK load balancer sample application [22], 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 [81], which provided isolated performance evaluation environments. We used the DPDK-based Pktgen application [55] 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.

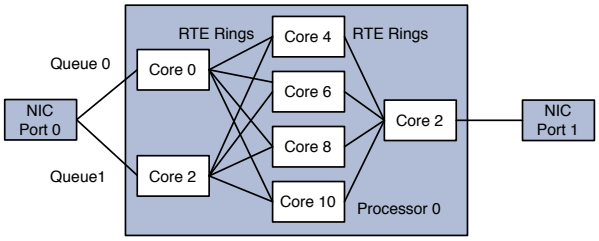


Figure 3.13: Experiment Configuration

Figure 3.13 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 3.13.

Table 3.2: 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

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 3.2. 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 3.2.

3.5 Discussion

Although the presented design has demonstrated 10 Gbps forwarding throughput with 256-byte packets and one billion forwarding rules, each containing up to seven name components, 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 servers are capable of supporting a larger amount of memory now 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 [65], therefore an efficient combination of hash functions is needed.

3.6 Summary

In this chapter, 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. NDN 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.

Chapter 4

FIB Optimizations

In this chapter, we continue to focus on scalable forwarding information base design. As in IP forwarding, the longest name prefix matching performance in NDN can be optimized by leveraging the specific characteristics of the forwarding rules stored in the FIB. In this chapter, we present two approaches to improve FIB lookup performance.

First, we note 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. We evaluate the effectiveness of level pulling with the available URL datasets is via simulation.

Second, we focus on reducing the memory requirements of forwarding rules that contain large numbers of name prefixes that have only one name component. We first briefly review speculative forwarding, which has been proposed to reduce the memory requirements of the FIB in core routers, and then present fingerprint-based methods to further improve name-based forwarding performance. The proposed fingerprint-based hash table design requires

only 3.2 GB of memory to store one billion names, and the measured lookup latency of the software-based single-threaded implementation is 0.29 microseconds.

4.1 Level Pulling

In this section, we still focus on hash table-based longest name prefix lookup and present level pulling, a generic method that reduces the average number of hash lookups for each LNPM query.

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 [65, 80], so that the hash tables that store more prefixes are visited early on. As noted in [80], 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.

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

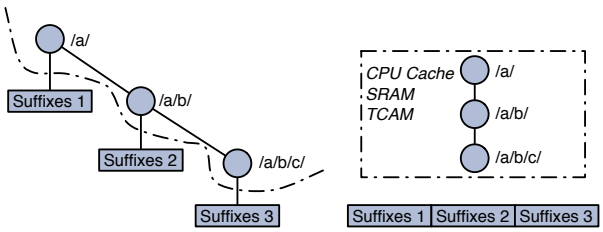


Figure 4.1: Level Pulling Concept

`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 4.1 shows the concept of level pulling with a name-component trie as the cache-like structure. In this dissertation, we evaluate the percentage of hash lookup reduction via simulation. A Python program was developed to measure the reduced number of hash lookups.

Table 4.1: Existing Name Conversion Schemes

URL	http://www.named-data.net/project/
TLD [65, 80]	/net/named-data/www/project/
Site [32]	/named-data.net/www/project/
Host [54]	/www.named-data.net/project/

Various ways of converting URLs to NDN names have been used in research literature, as listed in Table 4.1. 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 4.2: 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 4.2. 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.

4.2 Speculative Forwarding

In this section, we briefly review the speculative forwarding method proposed in our collaborative work [67]. The next section presents our second FIB optimization method that further improves speculative forwarding performance.

The speculative forwarding method reduces the memory requirements of name-based forwarding for datasets that have large numbers of prefixes with only one name component. Speculative forwarding employs a Patricia-trie like data structure, which performs the path compression as the original Patricia-trie [35]. The original Patricia-trie still has large memory

cost because names are stored to support prefix matching. In speculative forwarding, only the information differences among the rules are stored in the speculative Binary Patricia-trie (sBPT). Thus, the name prefixes that have only one component are no longer stored in sBPT. To support longest prefix matching, suffix strings still need to be stored. For example, if both `/a` and `/a/b/c` are in the FIB, the suffix `/b/c` needs to be stored. Hence, the memory requirements of names that have only one component, such as domain names, can be reduced significantly. Although the matching prefixes are not verified in core routers, the packets that truly have matching prefixes in the FIB are guaranteed to be forwarded correctly. Eventually packets are verified in edge routers.

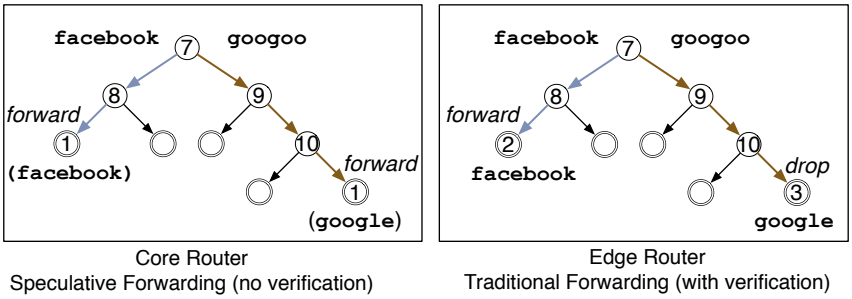


Figure 4.2: Speculative Forwarding Illustration

Figure 4.2 shows speculative forwarding in the core router and traditional forwarding in the edge router. In the sBPT shown on the left, only the bit positions that hold the information differences are stored. The name prefixes, as shown in parenthesis, are not stored. The edge router, shown on the right, stores the complete name prefixes. As shown in the figure, for a forwarding table that contains `facebook` and `google`, packets with name `facebook`, which is in the FIB, are forwarded correctly in the core router and are verified in the edge router. Packets with name `googoo`, which is not in the FIB, are also forwarded by core routers but eventually get dropped in edge routers. For the Alexa top one million domain names [2], the

sBPT requires only 5.58 MB of memory. The memory requirement of one billion synthetic name prefixes that have similar characteristics is 7.32 GB.

4.3 Fingerprint-based Solutions to Enhance Scalable Name-based Forwarding

In this section, we present the second FIB optimization method. Our goal is to further improve the name-based forwarding performance. To achieve this, we formulate the *string differentiation* (SD) problem, which is based on the speculative forwarding behavior in core networks, and identify the advantages that allow us to find efficient solutions to the problem. We focus on *exact string differentiation* (ESD), a special case of the string differentiation problem where no proper prefixes exist in the rule set. Unlike exact string matching, strings in ESD only need to be differentiated rather than matched. Following the information-theoretic difference approach, we propose fingerprint-based methods to improve the forwarding performance. In essence, by transferring the information differences among name prefixes to fixed-length fingerprints, the differences are expressed more concisely, reducing the lookup latency of Patricia trie-based methods. In terms of memory requirements, fingerprints are more compact than name prefixes, giving opportunities for memory-efficient solutions to name-based forwarding. We propose a fingerprint-based hash table (FHT) that stores only the fingerprint of the name, reducing the memory requirements considerably. When the physical resources on a single machine cannot meet the requirements of large datasets, a distributed forwarding scheme is needed. We study the distributed string differentiation problem, and evaluate the memory requirements of applying the proposed data structures to support distributed name-based forwarding.

4.3.1 Information-theoretic Difference Approach

Given a set of strings, what is the minimum information required to differentiate them? A simpler question would be to give you two names, `facebook` and `google`, and ask what is the minimum information that can differentiate them. The answer would be 1 bit, since at bit position 7 of their ASCII binary bit strings, `facebook` has the value of 0 and `google` has the value of 1. Hence, bit position 7 can be used to differentiate these two names. In addition, if given that the query is either `facebook` or `google`, we could identify which name it is by examining bit position 7. Similarly, in order to differentiate n names, at least $log(n)$ bits are needed. In practice, the number of bits required depends on how the information differences are distributed among these names.

There are two practical methods, Patricia-trie and fingerprint, that follow the information-theoretic difference approach. Patricia-trie examines the information differences among the input strings. The original Patricia-trie does not maintain the minimum amount of information differences because it greedily splits at a node whenever there is difference among the rules. A better splitting approach would always select the bit that has the maximum entropy, which is similar to a decision tree. However, generating a minimum-height decision tree has been proven to be NP-complete [60]. Fingerprints follow the information-theoretic difference approach by encoding the information differences in a more compact form. When there are no fingerprint collisions, the minimum fingerprint length is $log(n)$. Although fingerprints can be generated efficiently using hash functions, fingerprint collision is a primary challenge in practice.

4.3.2 The String Differentiation Problem

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

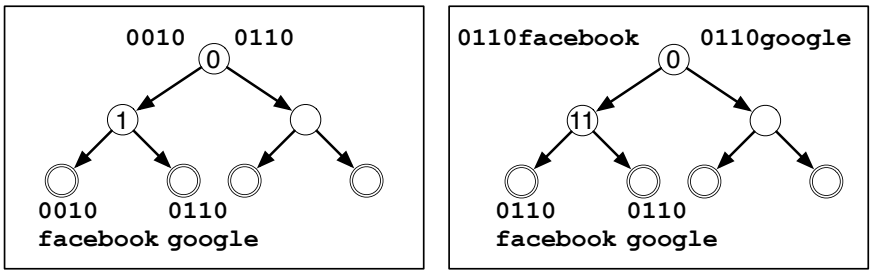
Problem Statement

The string differentiation problem can be defined as: Given a set of strings R, for any query string $r_q \in R$, how to differentiate r_q from the strings in $R - r_q$? The string differentiation problem is similar to the restricted candidate string problem [19]. Compared to the traditional string matching, string differentiation is in a more relaxed form because all the querying strings are assumed to be from the set R. In this chapter, we focus on the *exact string differentiation* (ESD) problem, which is a special case of the SD problem, where there are no proper prefixes in R. The proper prefix is defined as follows, when a string r_a is a prefix of string r_b, and $r_a \neq r_b$, then r_a is a proper prefix of r_b. Traditional longest prefix matching methods can be used to support proper prefixes, as shown in Section 4.2. Thus, combining ESD and longest prefix matching effectively solves the SD problem.

Fingerprint-based Solutions

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

Perfect Hashing In perfect hashing, each string r_i is mapped to a unique fingerprint f_i. Figure 4.3(a) shows how perfect hashing-based fingerprints can be used in the sBPT. It can be seen that using only the fingerprints is sufficient to differentiate the querying names. Although perfect hash functions can be found [80, 45], it is challenging to support fast updates efficiently and requires additional storage to generate perfect hash values.



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

Figure 4.3: Perfect Hashing and Non-perfect Hashing

Non-perfect Hashing Non-perfect hashing can be computed more efficiently, but fingerprint collisions occur. We present two collision resolution methods to address this issue. The first method employs a separate collision table (CT) to store the fingerprint-colliding strings. If inserting a string causes a fingerprint collision, this string is inserted into the collision table. As a result, each lookup requires querying the collision table. The size of the collision table is determined by the fingerprint collision rate. The second approach stores additional information for the fingerprint-colliding strings in the original data structure. Each colliding string is assigned a local fingerprint generated by a different hash function, so that colliding strings can be differentiated by examining the local fingerprints. In a special case, the local fingerprint can be the original strings, as shown in Figure 4.3(b). As can be seen, `facebook`

and `google` share the same 4-bit fingerprint, and they are differentiated by examining the 11th bit position in the input string, i.e., bit position 7 in the original strings.

4.3.3 Fingerprint-based Patricia-trie

In this subsection, we present the fingerprint-based Patricia-trie (FPT), which leverages fingerprints to enhance the Patricia trie-based name prefix lookup design.

The FPT Design

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

Table 4.3: Real-world Dataset Characteristics

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

We measured the leaf-node depths in the sBPT with the real-world Alexa [2] and Dmoz [21] domain name lists. Because we focus on the exact string differentiation problem, only the unique domain names were extracted from the datasets. The major characteristics of these two datasets are listed in Table 4.3. The average depths of the Patricia-trie structures with the Alexa and Dmoz datasets are 30 and 52 levels, respectively. Both are much larger than the optimal value $log(n)$, where n is the number of unique domain names.

Fingerprints are expected to have more balanced information difference distribution. To resolve fingerprint collisions, we could either use a collision table or prepend fingerprints to the names. For Patricia-trie, prepending fingerprints to the names is the simplest approach. It is worth noting that prepending a hash value to a fixed-length flow ID has been explored to reduce the average depth of level-compression tries [51], while we take the same approach on variable-length names to improve name-based forwarding performance. The height of the Patricia trie can also be reduced by dividing the complete datasets into smaller groups via hashing and then construct a subtrie for each group [67], however, prepending fingerprints is an orthogonal approach and can be applied to reduce the depth of each individual subtrie.

Experiments with Real-World Datasets

The original speculative Patricia-trie data structure can be used directly as a fingerprint-based Patricia-trie (FPT). The difference is that, in each insertion, deletion, and lookup operation, a fingerprint is prepended to the original name. Fingerprints are computed using the CityHash function [15].

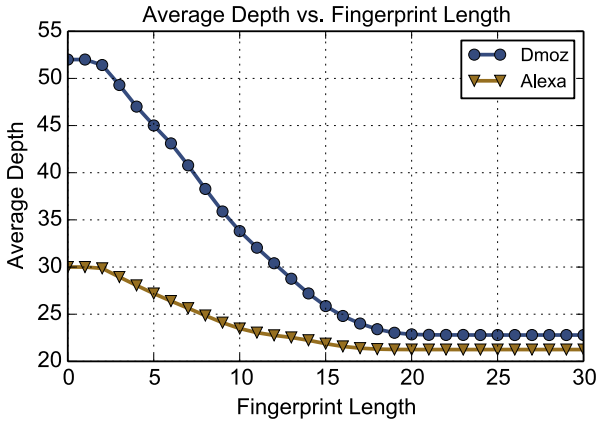


Figure 4.4: Impact of Fingerprint Length on the Trie Depth

We measured the impact of the fingerprint length on the leaf-node depth distribution. The entire Alexa or Dmoz domain names were inserted into the FPT, and the average Patricia-trie depth are shown in Figure 4.4. As the fingerprint length increases, the average leaf-node depth decreases. When the fingerprint length is greater than $log(n)$, the average depth becomes stable. Take the Dmoz dataset for example, after prepending 20-bit long fingerprints, the average depth is reduced from 52 to 24.

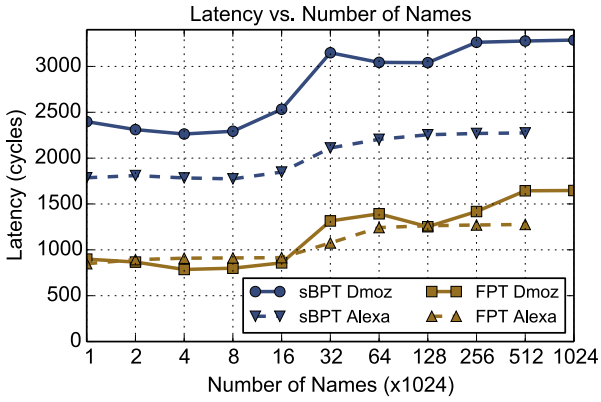


Figure 4.5: Patricia Trie-based Solution Lookup Performance

We have also evaluated the lookup performance of fingerprint-based Patricia-trie in software. The experiment was performed on the same machine used in Chapter 3, which was equipped with 12 Intel Xeon E5-2630 cores operating at 2.3 GHz, 15 MB of L3 cache, and 192 GB of DDR3 memory. To measure the lookup latency, the entire Alexa or Dmoz domain names were inserted into the FPT, and then the names in the same dataset were looked up. The lookup latency was measured using the `RDTSC` instruction, which provides high precision and low processing overhead. We started with looking up $1,024$ names, then doubled the number of names each time, and eventually half a million names were looked up for Alexa and one million names were looked up for Dmoz. Each experiment run 100 times, and the average latency per lookup is shown in Figure 4.5. As can be seen, the lookup latency of the FPT outperforms the sBPT, which is expected because the number of memory references is reduced. Note that only the small-scale experimental results with real-world datasets are presented here, and the large scale experiments, which require further software optimizations, are presented in Section 4.3.7.

4.3.4 Hash Table-based Methods

The Patricia trie-based solutions can possibly be implemented in purpose-built hardware with pipelining, but for processor-based platforms, trie-based schemes require considerable numbers of memory references. In this subsection, we present hash table-based designs for the exact string differentiation problem. The presented collision free fingerprint-based hash table (FHT) can be used in both software-based and hardware-based implementations.

The FHT Design

Hash tables have been used widely in network applications [34], but traditional hash tables store the entire key strings, consuming a large portion of the memory. Fingerprint-only hash tables [7, 23], which store only fingerprints of the keys, have been proposed for approximate membership querying and tolerate a small number of fingerprint collisions, i.e., false positives.

Unlike fingerprint-only hash table-based approximate membership querying designs, fingerprint collisions need to be resolved for the exact string differentiation problem. We propose a *collision free fingerprint-based hash table* (FHT), which has small memory requirements and there are no false positives. Because the query strings in the ESD problem are from a known string set R, thus employing a collision table to store the colliding strings resolves fingerprint collisions. In addition, the collision table also stores overflowed keys from the hash table, increasing the hash table load factors. Note that when the query string is not in R, i.e., packets with names that do not have any matching prefixes in the FIB, false positives could occur. This is a common issue faced by packet forwarding solutions that do not store the complete forwarding rules [86]. What is more, in name-based forwarding, the false positives can be eliminated by the edge routers, which store the full name prefix strings [67].

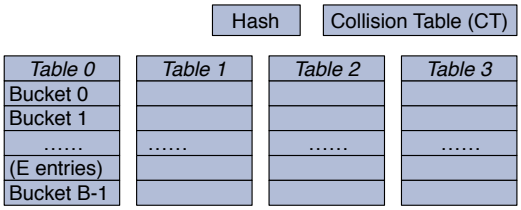


Figure 4.6: Fingerprint-based Hash Table

The collision-free fingerprint-based hash table is a simple modification of the original fingerprint-based hash tables [7]. The only difference is that a collision table is introduced. During an

insertion, if there is a fingerprint collision, then the complete string of the inserting key is stored in the CT. During a lookup, the CT is queried first, and then the hash tables are looked up only if there is no match in the CT. Figure 4.6 shows the collision-free fingerprint-based hash table design. The FHT is based on a d-left hash table [34], where there are d subtables. Each subtable has B buckets, and each bucket has E entries. In our experiments, $d = 4$ subtables are used, and each hash bucket holds $E = 8$ entries. The load factor ld of the hash table is set to 93% to accommodate our largest experiment configuration where one billion (10^9) names are stored in a hash table with 2^{30} entries. Each entry stores a fingerprint, an eight-bit outgoing port, and one more bit to indicate if it is occupied or not. The size of the CT, denoted as S_{CT}, is determined by the fingerprint collision rates and bucket overflow rates.

Because large-scale NDN forwarding rules are not available yet, the workload used in this section is randomly generated. The workload contains one billion names, and each name has only one name component as we focus on the exact string differentiation problem. The average length of the name is about 30 bytes. We calculated the memory requirements of storing one billion names with different fingerprint lengths w. The measured S_{CT}, and the calculated collision table memory requirements (M_{CT}), hash table memory size (M_{HT}), and total memory requirements of the FHT (M_T) are listed in Table 4.4. The fingerprint-based hash table requires less memory storage than the fingerprint-base Patricia trie because it also stores only the information difference among the rules, and what's more, there is no pointer storage.

From Table 4.4, even in the case where 16-bit fingerprints are used, there are only 324K items stored in the collision table. The CT is also implemented as a 4-left hash table, and it is configured with 75% load so that no CT overflow occurs. Each CT entry stores a 64-bit

Table 4.4: FHT Memory Requirements

w	**12**	**16**	**20**	**24**
S_{CT}	3.69E+6	3.24E+5	1.13E+5	1.01E+5
M_{CT}(MB)	1.81E+2	1.59E+1	5.55E+0	4.94E+0
M_{HT}(GB)	2.63	3.13	3.63	4.13
M_T(GB)	2.80	3.14	3.63	4.13

fingerprint, a 48-bit name prefix pointer (only the lower 48 bits of 64-bit memory addresses are used as virtual addresses in current processors [88]), and an eight-bit outgoing port. Because each name is about 30 bytes long, the estimated CT memory requirements is about 16 MB, thus it can be stored in SRAM. Using 16-bit fingerprints, the total memory size of the FHT is 3.14 GB, which reduces the memory requirements of the original speculative Binary Patricia-trie (7.32 GB) by 57%.

For one billion names, the d-left hash table requires approximately 3 to 4 GB of storage, which needs to be stored in DRAM. In a hardware-based implementation, each query first visits the on-chip SRAM-based collision table, and then looks up the DRAM-based main d-left hash table. In the worst case, each lookup requires d number of DRAM memory accesses; and the average case requires $(1 + d)/2$ accesses. Because the d-left hash table is already a compact data structure that stores only fingerprints, previous works [38] on employing on-chip filters to reduce memory accesses cannot be applied. In a hardware-based design, with sufficient resources, it is possible to store each subtable into a separate DRAM module, so that these d memory accesses can be pipelined or parallelized. In software-based designs, general purpose processors can maintain multiple memory requests to hide the access latency, and we present the impact of two known software optimization techniques in Section 4.3.7.

Experiments with Real-World Datasets

To evaluate the performance of software-based FHT, we used the same experimental setup as what we did with FPT to measure the lookup latency. Since the FHT load factor is kept as 93%, we started with looking up $1024 \times 93\% \approx 954$ names, then the number of names were doubled each time, and eventually $976,562$ names were queried. The performance results are shown in Figure 4.7. Because the collision table lookup can be offloaded in a hardware-assisted design, we present the FHT lookup latency with both querying the CT (denoted as w/ct) and skipping the CT (denoted as w/oct). We also include the previous FPT results for comparison.

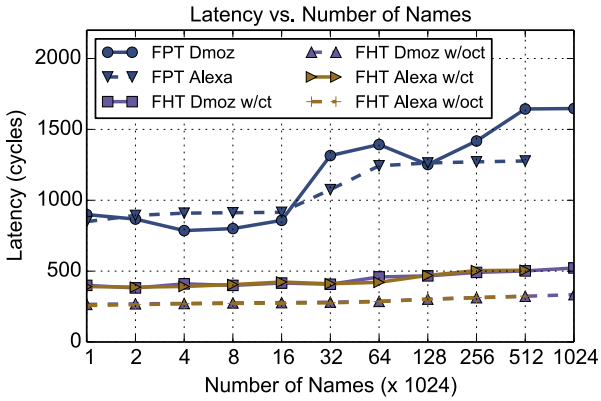


Figure 4.7: Hash Table Lookup Performance

From Figure 4.7, the hash table-based design outperforms the fingerprint-based Patricia-trie considerably. Although the CT size is small, there is still overhead associated with querying the CT before looking up the main d-left hash table.

4.3.5 Distributed Forwarding

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

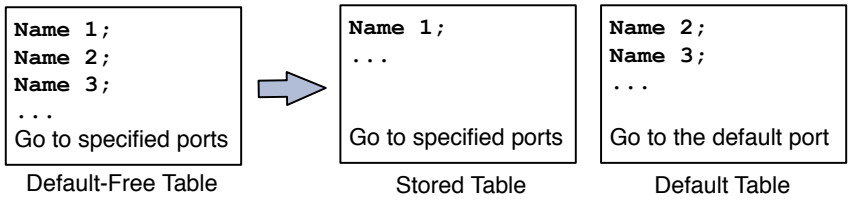


Figure 4.8: Distributed Name-based Forwarding

Figure 4.8 illustrates the distributed name-based forwarding application. In this example, the complete FIB table, i.e., the default-free table, is divided into two tables, a *stored* table and a *default* table. The stored table contains the forwarding rules stored in the router, and each rule has a specific outgoing port. The default table contains the rules that are not stored, and packets with these name prefixes need to be recognized and forwarded to a default port. The distributed string differentiation problem is about differentiating any string $r_i \in R_S$, and recognizing strings in $R - R_S$, i.e., membership testing. To reduce the memory requirements, the data structure is typically built with only strings in R_S. However, when only the information differences among R_S are stored, strings in $R - R_S$ is not guaranteed be recognized. Our approach is to store a small amount of additional information to support membership testing. We use the Patricia-trie to illustrate the approach.

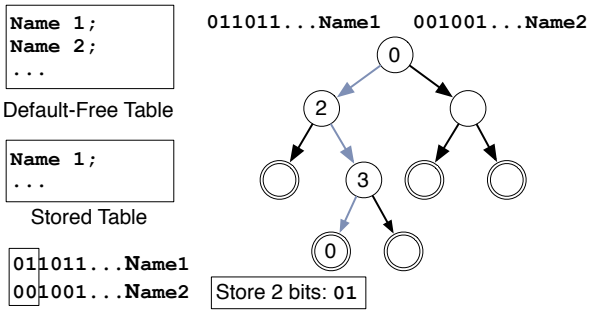


Figure 4.9: Distributed Forwarding Memory Optimization

In Figure 4.9, a Patricia-trie is built with the strings in the stored table, therefore, `Name1` is stored and `Name2` is not. When a packet with prefix `Name2` arrives, it is possible for `Name2` to reach the matching node of `Name1` if the bit values of the examined positions happen to match. For a FIB of size m, and a stored table of size n, the average number of collisions is expected to be m/n. Thus, $m/n-1$ strings need to be recognized from the *owner* string of a matching node. In Figure 4.9, the least number of bits required to recognize the *owner* name are stored in the matching node. In this example, the first two bits of `Name1`'s fingerprint are stored.

Patricia Trie-based Approach

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

Figure 4.10 shows the memory overhead for distributed forwarding with the Alexa dataset. In our design, local fingerprints were generated by up to two hash functions. In the One

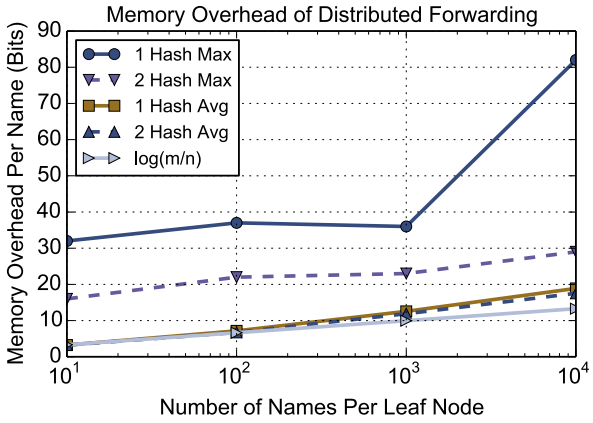


Figure 4.10: Memory Overhead of Distributed Forwarding

Hash case, only one fingerprint was generated, and the number of bits to be stored at each node was noted. In the Two Hash case, two fingerprints were generated for each name, and then the fingerprint that required less number of additional bits is chosen. In addition, one more bit is required to indicate which hash function to use. We present both the average and the maximum number of additional bits. From Figure 4.10, as the number of colliding names per leaf node increases, the number of additional bits increases. Although the average number is close to $log(m/n)$, the maximum number is much larger and it determines the memory overhead if all of the leaf nodes have the same memory layout. To reduce the memory requirements in practice, as most of the leaf nodes require close to $log(m/n)$ of bits, a threshold can be set, such as $log(m/n) + k$, where all the leaf nodes can store up to $log(m/n) + k$ bits, and then the names that require more bits would be inserted into the FPT so that they are differentiated by the trie structure.

Hash Table-based Approach

For distributed forwarding, the fingerprint-based hash table can be more memory-efficient because a fingerprint is already stored in each entry. We use the dataset with one billion randomly generated names to evaluate the memory requirements of the distributed FHT. The FHT is configured with 4 subtables, 8 entries per bucket, 16-bit long fingerprints, and the load factor is kept at 93%. The FHT stores S_p percent of the one billion dataset, and the hash table size N was varied from 64M to 512M. In the first phase, $S_p \times N \times 93\%$ names were randomly chosen and inserted. In the second phase, the rest of the names were queried, and names with fingerprints matched in the FHT are stored in the collision table. The memory requirements of this distributed table (M) and the percentage of total memory (M_p) are listed in Table 4.5.

Table 4.5: Distributed Forwarding for One Billion Names

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

From Table 4.5, the memory requirement of the subtable is always proportional to the number of rules stored in the table. As a result, the FHT can be employed to support distributed name-based forwarding with large datasets.

4.3.6 FIB Updates

The FIB table needs to be updated when routes change. When lossy data structures such as sBPT, FPT, and FHT are used, the FIB does not have sufficient information to process

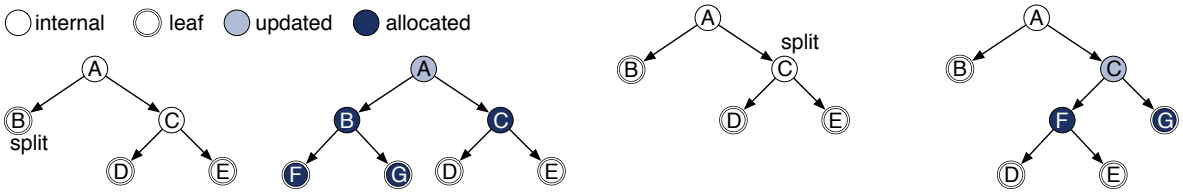
route updates directly. This problem also occurs in today's IP network when FIB compression schemes are used [71]. In practice, the FIB table can be incrementally updated with the assistance of a route controller (RC), which is employed in modern router architectures. The RC maintains the Routing Information Base (RIB), handles route updates, and generates FIB tables. The forwarding engines are distributed to multiple line cards, where they download the FIB table from the RC and then perform fast packet forwarding [13]. On the arrival of route update messages, the RC updates the RIB following the routing protocol and generates the new FIB tables. In general, the RC generates a sequence of *update instructions* to incrementally update the FIB structures in each forwarding engine. In this section, we describe the required instructions for both trie- and hash table-based FIB structures, and present the measured RIB update performance.

Patricia-trie Updates

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

The instruction generation for the lossy Patricia-trie (sBPT, FPT) needs to consider the node memory layouts. Each Patricia-trie node has two children, and maintaining both child pointers has a high memory cost. Thus optimized methods have been proposed [67], such as the single-memory address. In addition, internal nodes and leaf nodes have different memory layouts, so if the node type is changed, both this node and its sibling need to be reallocated,

and their parent must be updated with the child's new address. Here, we use the single-memory address scheme because it reduces the memory cost. Each node stores the left child pointer, and the right child always stays next to the left child in memory.



(a) Splitting a leaf node (b) Splitting an internal node

Figure 4.11: Patricia-trie Update

Figure 4.11 shows the two cases of inserting an entry into the Patricia-trie. In Figure 4.11(a), Node B needs to be split because the inserted entry differs from the entry stored in Node B. As a result, leaf nodes F and G are allocated to store these two entries. Node B now becomes an internal node, thus it needs to be reallocated. Because siblings always stay next to each other, Node C is also reallocated. In the end, Node A is updated with the Node B's new address. In Figure 4.11(b), Node C needs to be split, although its node type is unchanged. In this case, Node F and G are created first: Node F copies the information from Node C, and Node G stores the inserted entry. Node C is then updated with the new bit position and child's address.

Hash Table Updates

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

table. Deletions and updates have similar effects: either a hash bucket or a collision table entry is updated.

Update Performance

We use the insertion latency as the metric to evaluate the update performance. To see the trend of the latency as datasets become larger, we started with approximately one million ($2^{20} \times 93\% \approx 9.77 \times 10^5$) names, then doubled the dataset size at each step, and eventually reached one billion (10^9) names. The experiment was repeated three times, Figure 4.12 shows the measured average insertion latency with 95% confidence intervals.

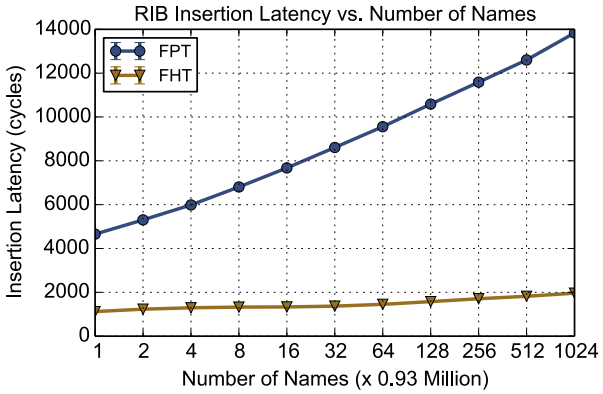


Figure 4.12: RIB Insertion Performance

For one billion names, the FHT requires $1,991$ cycles to insert a name. As the frequency of the processor is 2.3GHz, thus FHT supports processing approximately 1.2 million packets per second (MPPS). Our Patricia-trie implementation is not yet heavily optimized. The FPT has a much higher latency due to more memory accesses, and with the largest dataset, on average $13,838$ cycles are required for each insertion. Obviously, FHT supports faster

updates, while a single FPT structure has limited scalability in terms of updates. In practice, large datasets can be split into smaller ones to be constructed independently for FPT.

4.3.7 Experimental Evaluation

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

Software Optimization for Large Datasets

The characteristics of general-purpose multicore processors allow further optimizations for applications that require frequent memory accesses and large amount of memory storage. Specifically, software memory prefech instructions and large page sizes have been exploited to improve packet forwarding performance [23, 65, 88, 91]. We apply both software prefetching and large page sizes for the experiments with large datasets. Software memory prefetch instructions can be employed when the locations of future memory references can be known. Thus, it is straightforward to improve the performance of hash tables but not trie-based designs. With memory prefetching, all of the d subtables in the FHT are fetched at the same time, and the processor is able to handle multiple memory requests efficiently. Larger page sizes reduce translation lookaside buffer (TLB) misses, and we use both the 4KB pages and 2MB pages for FPT and FHT.

Lookup Latency

We measured the lookup latency of the proposed data structures with one billion synthetic names. Similar as the FIB update experiments, we started with approximately 1 million names, then the number of names in the dataset was doubled at each step. At each step, the forwarding data structure was built, and then all the names were looked up. We ran each experiment three times and recorded the average lookup latency with 95% confidence intervals. The measured results for the FPT are shown in Figure 4.13 and the results for FHT are shown in Figure 4.14.

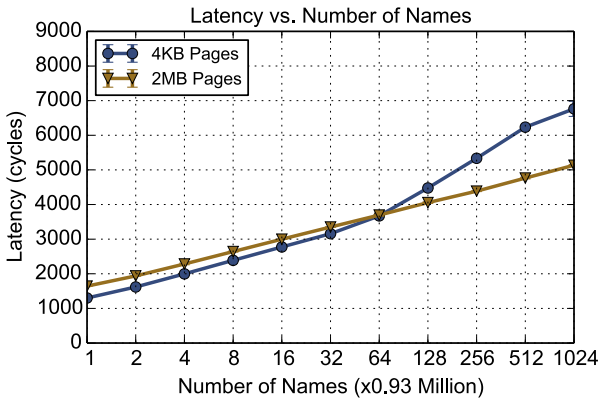


Figure 4.13: Fingerprint-based Patricia Trie Lookup Performance

According to Figure 4.13, the lookup latency of FPT increases linearly as the number of names doubles each time. When the datasets are small, i.e., less than 64 million names, the performance with 4KB pages performs slightly better than the one with 2MB pages. As the datasets become larger, the lookup latency with 2MB pages outperforms the one with 4KB pages. With one billion names, $5,112$ cycles are required for each lookup. The processor's frequency is 2.3 GHz, thus roughly 0.44 million packets can be processed in each second. It is obvious that when the datasets are large, the software-based implementation of the FPT

does not yield impressive results, and therefore, FPT would likely rely on hardware-based implementations.

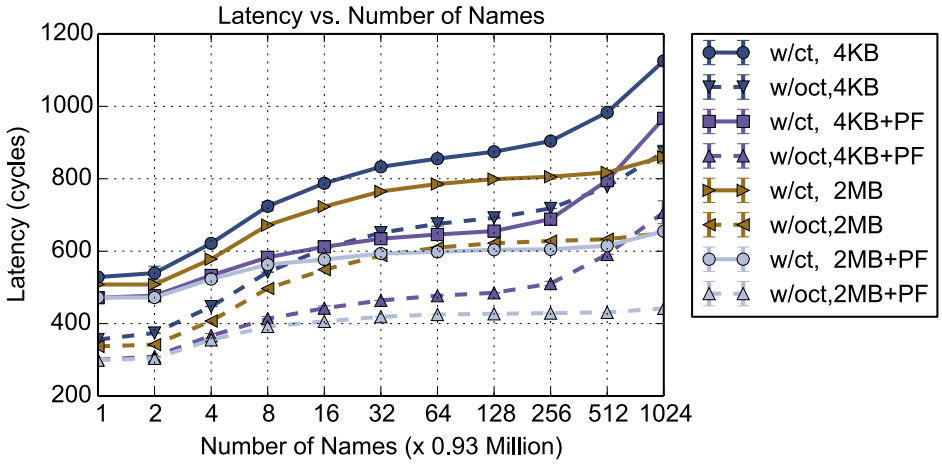


Figure 4.14: Fingerprint-based Hash Table Lookup Performance

As expected, the FHT performs much better than FPT in software. As shown in Figure 4.14, each lookup requires only $1,126$ cycles without any additional optimization. With 4KB pages, the lookup latency increases sharply when the datasets become large, e.g., 256 million names or more, due to more TLB misses. With 2MB pages, the lookup latency increases moderately. Regardless of the configuration, querying the collision table always introduce additional latency. An interesting observation is that, when either only software prefetching or only 2MB pages is employed, the performance with software prefetching outperforms the one with 2MB pages when the datasets are relatively small, i.e., less than 512 million names. Eventually, the performance with 2MB pages outperforms the one of software prefetching with one billion names. This is what we expected as more TLB misses are likely to occur when the memory size becomes large.

The best performance is achieved when both software prefetching and large pages are used. When the CT is queried, each lookup requires 654 cycles (0.29μs); when the CT lookup is

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

4.3.8 Related Work

Previous studies have explored both trie-based and hash-based data structures to reduce the memory requirements of name-based forwarding. Trie-based approaches reduce memory requirements as prefixes are shared, but the memory requirements of storing the shared name prefix are still considerable. Encoding methods [77] have been explored to reduce the memory requirements, with the cost of additional lookups to generate the encoded names. As for the designs that employ compact but lossy data structures, the bloom filter-based design proposed in [78] may introduce false positives even when there is only one name component in the name. The design proposed in [80] stores only fingerprints in the hash table using perfect hashing, however, the structure that used to generate perfect hash values requires additional storage. In addition [78, 80] do not guarantee to differentiate name prefixes with different numbers of name components, thus they may introduce false positives as hash collisions could occur between name prefixes with different numbers of components. In the recently proposed speculative forwarding method [67], the string verification requirement in the core routers is relaxed, and thus the memory requirement is reduced considerably. In our work, we focus on exact string differentiation and only names with one name component are

considered. Names with multiple name components can be looked up in a separate structure that supports traditional longest name prefix match with no false positives.

4.4 Summary

Name-based forwarding is a core component in information-centric networking. The forwarding information base design can be optimized based on the specific characteristics of the forwarding rules.

First, we have 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, the proposed level-pulling method can be applied to optimize the lookup procedure.

Second, the speculative forwarding method reduces the name-based forwarding memory size significantly by relaxing the string verification requirement in core networks. We define the string differentiation problem and propose fingerprint-based solutions to enhance the name-based forwarding performance. The fingerprint-based Patricia-trie effectively reduces the lookup latency of sBPT, and its performance can be improved further with pipelining techniques in practice. The fingerprint-based hash table reduces the memory size and lookup latency further. The FHT requires only 3.2 GB to store 1 billion names, with a lookup latency of 0.29 μs. Hence, the single-threaded software implementation of the FHT design supports packet processing rate at 3.5 MPPS, and its throughputs can be improved further via parallelism on multicore or hardware-accelerated platforms.

Chapter 5

Pending Interest Table

In the previous two chapters, we have focused on the Forwarding Information Base design. In this chapter, we focus on the design of the Pending Interest Table.

A Pending Interest Table (PIT) is a core component in Named Data Networking. Scalable PIT design is challenging because it requires per-packet updates, and the names stored in the PIT are long, requiring more memory. As the line speed keeps increasing, e.g., 100 Gbps, traditional hash table-based methods cannot meet these requirements. In this chapter, we propose a novel Pending Interest Table design that guarantees packet delivery with a compact and approximate storage representation [87]. To achieve this, the PIT stores fixed-length fingerprints instead of name strings. To overcome the classical fingerprint collision problem, the Interest aggregation feature in the core routers is relaxed. The memory requirement and network traffic overhead are analyzed, and the performance of a software implementation of the proposed design is measured.

Our results show that 37 MB to 245 MB are required at 100 Gbps for a single router case, so that the PIT can fit into SRAM or RLDRAM chips. When multiple core routers are used, the memory requirements are twice of that of a single core router. As a result, 74 MB to

490 MB are required in the worst case, which is still slightly more memory-efficient than a standard hash table-based design.

5.1 Introduction

The PIT keeps track of the currently unsatisfied Interest packets. Arriving Interest packets are forwarded to the next hop based on a FIB lookup only if the PIT finds no pending Interest packet with the same name. The PIT also stores the destination information for Data packets. For each Data packet, the PIT is queried to find the incoming face(s) that requested the content, and then the Data packet will be delivered and its content name is deleted from the PIT. Hence, the PIT requires per-packet updates, including memory writes. At 100 Gbps, a 100-byte packet could have an arrival rate of eight nanoseconds. In the meantime, the PIT memory size becomes larger as the link speed increases, which makes space-limited high speed memory devices infeasible to use. In each PIT entry, the content name needs to be stored. The names are similar to URLs, and today's URLs typically require tens of bytes of storage. For example, the URLs for the pictures and videos on popular social networking websites, which include long hash numbers, are more than 80 bytes long. Moreover, there are websites that include article names in the URLs, making the URLs longer. As a result, designing a fast and scalable Pending Interest Table is challenging. Based on the PIT memory requirements analysis in Section 5.4.5, we focus on the PIT design for 100 Gbps links because it is more challenging, although the proposed data structures can be applied with other link rates.

Our approach is to reduce the PIT memory size for core routers, so that fast memory chips, such as SRAM or RLDRAM, can be employed to support per-packet updates. Uniquely, our

design guarantees packet delivery with a compact and approximate storage representation. We propose a network-wide solution to the scalable Pending Interest Table problem. In our design, we classify network routers as either *core* routers or *edge* routers, just as in today's Internet. The edge routers function as usual, but the core routers store a fingerprint instead of the full name string for each Interest packet. As a result, the core router PIT memory requirement is greatly reduced. However, fingerprint collision, a classical problem, arises. To guarantee packet delivery, we relax the Interest aggregation requirement in core routers, i.e., every Interest packet received by the core router PITs will be forwarded. To provide enough time for potential multiple Data packets to arrive, a colliding fingerprint entry will not be deleted until it has expired. Another problem is that the proposed system cannot differentiate fingerprint collisions from duplicate Interest requests. Thus, for duplicate fingerprints, we leverage the idea that most Interest aggregation happens at the edge routers, and also use the Content Store in the core routers to help prevent receiving duplicate Interest requests. The proposed design introduces additional network traffic, but it will be dropped by the edge routers. Hence, the entire packet processing procedure is transparent to the users and content providers.

Specifically, we make the following contributions in this chapter.

- We propose a PIT design that takes advantage of a compact storage representation and the edge router filtering effect, so that the memory requirement of the Pending Interest Table in core routers is reduced.
- We use analytical modeling to analyze the memory requirement and demonstrate the network traffic overhead is acceptable. Our results show that 36.77 MB to 244.44 MB are required at 100 Gbps for the case with a single router. When multiple core routers are used, the memory requirements are doubled.

- We have implemented the PIT design in software, and measured the fingerprint collision rate and update latency. At 1 Gbps, the measured latency is 1.2μs. The update latency can be improved further using software optimization techniques such as memory prefetching.

5.2 Background

In this section, we review the functionality and design considerations of the Pending Interest Table, and then consider general hash-based techniques.

5.2.1 Design Considerations

The Pending Interest Table provides two major functions in the NDN architecture, namely Interest packet aggregation and Data packet multicast. Each incoming Interest name is looked up in the PIT, and duplicate Interest requests are aggregated, i.e., the Interest packet will be forwarded only if its name is not found in the PIT. The PIT keeps track of which face has requested what content, and each PIT entry stores a list of its Interest incoming faces. When a Data packet arrives, the PIT is queried to fetch all the outgoing faces, and then the Data packet is delivered. The PIT design has been recognized as a flow table management problem [89]. In IP networks, generally a five-tuple rule is used to define a flow. Likewise, we could define each NDN flow using a content name. Each flow has its expiration time and a list of incoming faces. NDN packets have hierarchically structured names, while in this chapter we use exact string matching for PIT name lookup. While our approach provides the essential functionality, it is worth noting that the PIT lookup

in the NDN reference implementation has additional features [12]. It is also suggested in [89] that the core and edge routers should have different features. Indeed, our proposed design leverages the differences between these two types of routers. The detailed differences between the edge and core routers are presented in Section 5.3.

5.2.2 Hash-based Techniques

Hash tables have been studied extensively in the past decade for high-speed packet processing [34]. Hash table designs generally aim at a constant number of off-chip DRAM references. Previous studies [38, 40, 66] show that storing filters on a small SRAM chip greatly reduces the number of DRAM accesses. The filters stored in SRAM are typically Bloom filters, or counting Bloom filters. It has also been proven that storing fingerprints in a hash table provides the same filtering function [7]. However, applying these designs directly for PIT still requires at least one DRAM access to write the content name and other information for each insertion. Since a long content name exceeds the DRAM bus capacity, each PIT entry access requires multiple cycles, or can be optimized as one or two burst accesses. Our approach seeks to eliminate the DRAM accesses. Hash tables are preferred to Bloom filters because the expiration time and the face list can be easily stored for each entry.

5.3 Pending Interest Table Requirements

In this section, we discuss the differences between edge and core routers, and then highlight their requirements.

Table 5.1: Pending Interest Table Requirements

Line Rates	**Edge(1 G)**	**Core(10 G)**	**Core(100 G)**
Interfaces	thousands	hundreds	tens
Best Case	0.156 Mpps	1.563 Mpps	15.625 Mpps
Worst Case	1.25 Mpps	12.5 Mpps	125 Mpps
Best Mem	0.625 MB	6.25 MB	62.5 MB
Worst Mem	5 MB	50 MB	500 MB

Edge routers connect consumers to ISP networks, and therefore the numbers of interfaces on edge routers are typically large, reaching 64 thousand [53]. The throughput is not high for edge routers, so in this chapter, we select 1 Gbps as the link rate. Network traffic aggregates at edge routers and then enters the backbone networks. Core routers are deployed in backbone networks, where the throughput rather than the number of interfaces is the primary concern. A high-end router may contain multiple line cards, and therefore its bandwidth can reach 10 Gbps, 100 Gbps, or more. In this case, the number of entries in the PIT is large, and the PIT needs to be updated efficiently. Generally, there are not as many features on core routers as on edge routers, and the number of faces is small, ranging from a few interfaces to tens of interfaces [16].

The number of packets arriving at each face is also affected by the packet sizes. Since NDN can run on top of Ethernet directly, the packet size could be as small as 64 bytes, or as large as 1500 bytes. Generally, Interest packets are relatively small, and Data packets are large. In a flow balance mode, one Data packet is transferred for one Interest packet. The best case can be configured as 100-byte Interest packets and 1500-byte Data packets. In the worst case, the Data packets can be as small as the Interest packets; therefore, we set 100 bytes for both of them. Table 5.1 lists both the operation frequency and memory requirements, assuming the round trip time $T_{rtt} = 80$ ms [53], $S_I = 100$ Bytes, the best case $S_D = 1500$

Bytes, and the worst case $S_D = 100$ Bytes. We are not considering the case where most of the Interest packets cannot be satisfied and have to wait for expiration, because this behavior could happen only when the PIT is under a flooding attack. PIT security issues are discussed in Section 5.7. From Table 5.1, for low-end routers, the memory size in the worst case can be easily fit into SRAM. Even for the 10 Gbps case, it is possible to fit them into RLDRAM. However, for the 100 Gbps case, its memory size cannot be fit into RLDRAM, thus DRAM has to be used. It is worth noting that we are designing the PIT for the worst case (i.e., $S_D = 100$ bytes), since we believe the PIT should be capable of handling the worst-case traffic. Moreover, even under other operation modes, our design could still save considerable memory space.

5.4 Fingerprint-only Pending Interest Table

In this section, we present the proposed fingerprint-only Pending Interest Table design in detail.

5.4.1 Design Overview

Our design is based on the ideas that storing fingerprints saves memory space, and that edge routers can aggregate most of the duplicate Interest packets. Thus, our system-wide solution to scalable PIT can relax the Interest aggregation requirement for the core routers. Figure 5.1 shows the system design. A wider arrow denotes a larger number of packets. In the figure, Interest packets are aggregated at the edge routers and then enter the core network. The core routers simply forward all the received Interest packets. Eventually, duplicate core

Interest packets are aggregated at the edge router before reaching the content provider. The content provider receives only one Interest request. Then one Data packet is replied and distributed to the users. The entire packet processing procedure is transparent to the users and content providers.

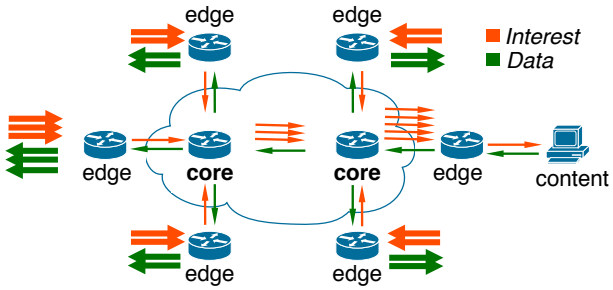
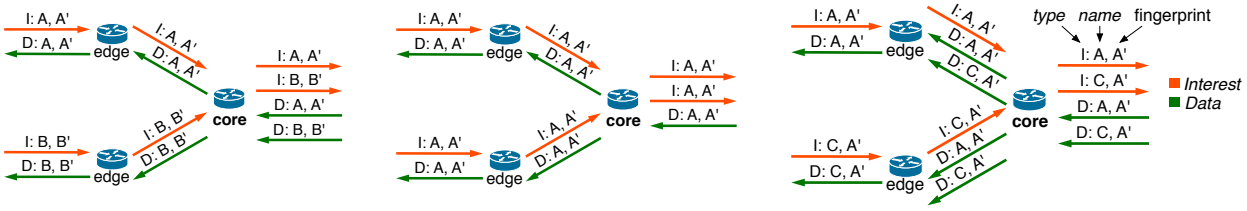


Figure 5.1: System Design

The PITs in edge routers operate as described in the NDN design [32], where name strings are stored, and Interest aggregation is supported. The PITs in the core routers store fixed-length fingerprints instead of name strings. Two challenges arise with this approach: *fingerprint collisions* and *duplicate Interest requests*. To guarantee packet delivery, Interest aggregation is not supported in the core routers when fingerprint collisions occur. Colliding fingerprints are not deleted from the PIT until they reach the expiration time T_{exp}, giving enough time to wait for potential multiple Data packets. To achieve this, each PIT entry in the core routers records if duplicate fingerprints have been received. The PIT entry expiration time and face list information are managed in the same fashion in edge and core routers. The second challenge is duplicate Interests. Although a duplicate Interest packet from a different face would make its PIT entry stay longer, we leverage the idea that most Interest packets are aggregated in the edge routers, and we will analyze the effect of duplicate Interest requests in Section 5.4.5.

While the edge routers follow the original operational flows as illustrated in [89], the operational flows of the core routers are modified. When an Interest packet arrives at the core router, the Content Store is queried to see if the content is cached. If the requested content is not cached, the PIT is consulted to see if it has already been requested. The lookup key is the fingerprint of the content name. If there is no match in the PIT, then this fingerprint is inserted, its expiration time is set, and its incoming face is recorded. If there is a match for this fingerprint in the PIT, then additional information about the collision is updated, the expiration time is refreshed, and the incoming face is added. In the end, the Interest packet is forwarded to the appropriate outgoing face by performing a FIB lookup, regardless of whether there is a PIT match or not. On the arrival of a Data packet, the packet will be selectively cached based on the Content Store caching policy. Then, the packet is looked up in the PIT, with the content name fingerprint as the key. If the fingerprint is found, then the Data packet is delivered to the face(s) stored in the face list. If this fingerprint has been received only once, it is removed from the PIT immediately; otherwise, it stays until the expiration time is reached. There are three operation situations, as shown in Figure 5.2, that could occur in this design.



(a) Normal Operation (b) Duplicate Interest Requests (c) Fingerprint Collisions

Figure 5.2: System Operations

Normal Operation. In the normal case, different Interest names map to different fingerprints. For example, name A maps to fingerprint A', and name B maps to B'. As a result, the operation is the same as the case where name strings are stored. No traffic overhead is introduced.

Duplicate Interest Requests. When duplicate Interest requests arrive at the core router from different faces, the fingerprint stays in the PIT longer. Hence, there is memory overhead compared with the normal operation, and there is Interest traffic overhead since duplicate requests are not aggregated.

Fingerprint Collisions. If two content names share the same fingerprint, then these two Interest packets take one PIT entry, rather than two, reducing the number of stored PIT entries. Still, the colliding PIT entry stays longer in the PIT, which introduces memory overhead compared with the normal case. Since the two Data packets are delivered to both faces, Data traffic overhead is also introduced. In a special case, if one Interest has been requested by many faces, then the absolute value of the additional traffic is much larger.

5.4.2 Data Structures

Our proposed PIT design is based on a d-left hash table [34], where the hash buckets are grouped to d subtables. To insert an item, all of the d subtables are visited and the item is inserted to the least loaded subtable. Figure 5.3 shows the PIT architecture with $d = 2$ and also a PIT entry example. Each hash table has a capacity of N/d entries, and it has B buckets, where each bucket holds up to E entries. The architecture also has an overflow table to store the items that cannot fit into the hash tables. The hash function determines

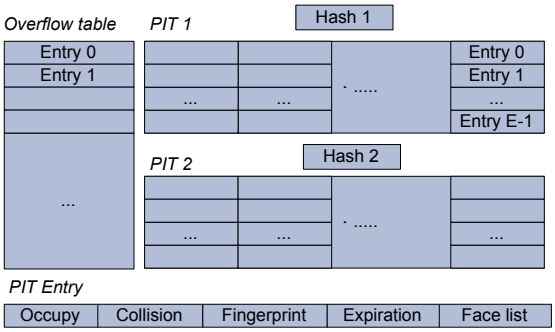


Figure 5.3: PIT Data Structures

the bucket location of a name string and also its fingerprint. As described in [7], the d hash functions in the PIT have to use permutation to generate the bucket indexes and fingerprints.

Each PIT entry consists of five parts: the Occupy bit, the Collision bit, the fingerprint, the expiration time, and the face list. The Occupy bit indicates whether this PIT entry space is occupied or not. It also enables lazy deletion, since deleting an entry requires only setting this bit to 0. The Collision bit shows whether more than one Interest requests have been received for this fingerprint. The fingerprints have fixed lengths, w-bit long. The expiration times also have a fixed length of t bits. The value of t is determined by the configured expiration time support and its granularity. In our design, $t = 10$ and its unit is 16 ms, thus the timer counts up to 16 seconds. The face list stores the incoming faces. In our design, it is a bit-vector of length n, where n is the number of faces. The bit vector is initialized to 0s, and the i^{th} bit is set to 1 when an inserted Interest comes from face i. In our analysis, we assume a core router with 16 ports. When the number of ports is larger, bit vector may not be the best face list representation, thus memory-efficient alternatives need to be explored, but it is beyond the scope of this dissertation.

Algorithm 1: INSERT AN ENTRY INTO THE PIT

Input: PIT T, Name $name$, Incoming face f_i

```
loc ← H(name) mod B
fp ← SHIFTRIGHT(H(name), log(B)) mod 2^w
foreach entry in T(loc) do
    if entry.occupy = 1 then
        if entry.expiration < current then
            RESET(entry)
        else
            if fp = entry.fp then
                fp is found
if fp is found at entry in T(loc) then
    entry.collision ← 1
else
    entry ← empty entry in T[loc], or overflow table
    entry.occupy ← 1
    entry.fp ← fp
entry.facelist[f_i] ← 1
entry.expiration ← current + T_exp
```

Algorithm 1 shows the steps for inserting an item into the PIT, while the method that handles timer rollover is not included here. We use a lazy expiration time check scheme, so the expiration time will not be examined unless it is visited during some operation. The deletion algorithm, which employs the lazy deletion method, is not shown here due to space limits.

5.4.3 Segregated Pending Interest Table

Although our fingerprint-only PIT significantly reduces the memory requirement, it may still exceed the size of a single memory chip when the link rates are high. We propose a segregated Pending Interest Table to address this problem. In this design, the single PIT is divided

into s segregated PITs in the router. Each segregated PIT keeps track of the unsatisfied Interest packets from n/s faces. As a result, the effective bandwidth of each segregated PIT is $1/s$ of the full link rate, and thus the required memory is much smaller. The Interest packets are sent to the appropriate segregated PIT based on their incoming faces, and the rest of the packet processing stays the same. In contrast, on the arrival of Data packets, all of the segregated PITs are queried to find the incoming face(s) of the corresponding Interest packets.

5.4.4 Popular Content Optimization

In this subsection, we first discuss the challenges of popular content, and then present optimization methods that address these problems.

Popular Content

Our fingerprint-only Pending Interest Table is based on the idea that most Interest packet aggregation happens at the edge routers, and therefore the core routers are more likely to receive unique Interest requests. We have two observations that further support this assumption. The first observation is that the effective Interest waiting time is short, generally equivalent to a packet round trip time T_{rtt}. In today's Internet, T_{rtt} is around 80 ms on average [53], thus the chance of receiving a duplicate Interest during such short period is low for general content. The second observation is that highly accessed pieces of content, as time goes on, are gradually cached in edge networks, so subsequent Interest requests can be satisfied locally and will not enter the core networks. Thus, the chance of receiving duplicate Interest requests during T_{rtt} is further reduced. However, certain types of content could still

be requested multiple times during T_{rtt}. For instance, live sports games and concerts could possibly be watched by millions of people from different edge networks at the same time. These Interest requests are more likely to be synced and arrive within T_{rtt}. In this case, the PIT entry stays at least T_{exp} long since a colliding entry cannot be removed until it has expired. In theory, the worst case could be that another Interest packet arrives right before the current entry expires, causing this entry to stay for at least another T_{exp} amount of time. Thus the entry could stay for approximately $T_{exp} \times (n-1)$ long in the worst-case scenario. Interest requests like this increase the PIT memory requirement and therefore need to be mitigated.

Optimization

We present three orthogonal methods to mitigate the popular content problem.

Content Store The worst case presented earlier assumes the requested Interest packets cannot be satisfied by the Content Store (CS), which caches Data packets in an NDN router. Interest packets do not query the PIT if they are satisfied by the Content Store. Storing popular content in the Content Store not only prevents colliding PIT entries being refreshed once the Data packets are cached, but also reduces the content delivery time for the users. Hence, a dynamic selective Content Store that gets invoked when a PIT entry becomes hot can be employed to mitigate the popular content problem. For instance, the Content Store can be configured to cache Data packets that have been requested more than once. Under this policy, each PIT entry stays at most $T_{rtt} + T_{exp}$ long. The size of the Content Store is determined by the number of duplicate Interest requests received.

Adaptive PIT The proposed PIT is designed for the worst case, where minimal-size Data packets are used. In practice, network traffic does not stay at peak all the time. As a result, we propose an adaptive PIT design, where fingerprints are stored under heavy traffic load, and full name strings are stored under light traffic load. Storing the names supports Interest aggregation, therefore the popular content problem would not occur. This idea is conceptually similar to the dynamic bit assignment method for fingerprint-based hash tables [8], which adjusts the fingerprint length based on the number of occupied entries in a hash bucket.

Segregated PIT The segregated PIT design can also be used in this context. Since the incoming traffic is divided into multiple sub-streams, the chance of getting duplicate Interest requests is reduced. In a special case, a smaller PIT can be deployed for each face. There are no duplicate Interest requests at each face due to Interest aggregation at the edge routers. Approaches that deploy a Bloom filter for each face [84][85][86] have been studied.

In addition, application-level optimizations, such as designated broadcast names, can be explored to help mitigate the popular content problem.

5.4.5 Analysis

Memory size and *network traffic overhead* are the two major metrics for evaluating the proposed Pending Interest Table design, and both of them are affected by the fingerprint distribution in the PIT. In this section, we first analyze the fingerprint-based hash table and derive an upper bound of the number of names that have duplicate requests, and then we present a detailed analysis of the memory size and network traffic overhead. Since the

actual characteristics of the traffic in the NDN core networks is not known, we use analytical modeling and Zipf-like fingerprint distributions instead. It is worth noting that relaxing Interest aggregation changes the dynamics in the core networks, and there are more Interest packets when duplicate Interest requests occur. For instance, when every content is requested twice, there are two Interest packets for every Data packet, so that 66.7% of the network packets are Interest packets. Also note that we assume there is only one core router in the network in this subsection, which is connected with multiple routers that perform both Interest aggregation and Data multicast. The case with multiple core routers is analyzed in Section 5.5.

Fingerprint-Based Hash Tables

Our analysis differs from previous studies: We are more interested in the fingerprint distribution than in the hash table false positive rates, which are typically considered because fingerprint-based hash tables are used as a membership query tool [7]. In these studies, duplicate items have no effect on how long the corresponding fingerprint is kept in the table. In contrast, our design uses hash tables to manage a stateful flow table. Duplicate Interest requests will cause the PIT entry to be recognized as collided and to be retained longer in the table.

The fingerprint distribution problem can be formulated as follows: Given an Interest name distribution function F for a hash table with capacity N, and w-bit long fingerprints, what is the corresponding fingerprint distribution function f? The value of F_i in function F is defined as the number of names that have exactly i duplicate copies (including itself) during T_{rtt}. For instance, there are F_1 number of Interest names being requested only once. Hence,

the total number of unique Interest names is $N_{total} = \sum_{i=1}^{n} F_i$, where n is the number of faces corresponding to this PIT. The total number of Interest packets is $P_{total} = \sum_{i=1}^{n} F_i \times i$.

Collisions affect the fingerprint distribution. A fingerprint collision occurs if and only if both the hash bucket locations and the fingerprints are identical. The average number of keys stored in each bucket is determined by the hash table load factor. In our design, each bucket has eight entries, and the load factor is 75% as this configuration provides good performance [7]. As analyzed in [7], the fingerprint collision rates are bounded by $E \times d \times ld \times (1/2^w)$, where E is the number of entries per bucket, d is the number of subtables, and ld is the load factor of the hash tables. In our case, when $d = 4$, $E \times d \times ld = 24$, thus the average fingerprint collision rates are bounded by $24/2^w$. As a result, the ratio of the names that involve fingerprint collisions is $24/2^{w-1}$. In addition, we derive that the probability of having i names collide is bounded by $\binom{24}{i-1}/2^{w\times(i-1)}$.

Deriving the entire fingerprint distribution function f seems possible, but requires complex mathematical work. Instead, we derive an upper bound on the number of duplicate fingerprints rather than seeking an accurate distribution function. The upper bound is sufficient for performing the worst-case analysis. The number of fingerprints that appear exactly once is

$$f_1 \geq F_1 \times (1 - 24/2^{w-1}). \tag{5.1}$$

In the worst case, each duplicate Interest name is requested exactly twice. Thus the number of duplicate fingerprints is

$$\sum_{i=2}^{\infty} f_i \leq (P_{total} - f_1)/2. \tag{5.2}$$

Memory Size

The PIT memory size is determined by the number of entries and the lifetime of each entry. Given a fingerprint distribution f, the total number of Interest packets is $P_{total} = \sum_{i=1}^{\infty} f_i \times i$. The PIT memory size is

$$M_{total} = \left(f_1 + \sum_{i=2}^{\infty} (f_i \times \frac{T_i}{T_{rtt}}) \right) \times \frac{M_{bucket}}{ld}, \tag{5.3}$$

where ld is the load factor of the hash table, and T_i is the average Interest lifetime of the names that have exactly i duplicate Interest requests. In our design, 10-bit expiration timestamps, 16-bit fingerprints, and 16-bit face list bit vectors are used, therefore $M_{bucket} = 2 + 10 + 16 + 16 = 44$ bits.

Worst Case Analysis When there is no Content Store, the worst-case lifetime of an entry can be as long as $T_{exp} \times (n-1)$ in theory, where n is the number of interfaces in the router. The traffic pattern that causes this worst-case behavior should rarely happen for general content. In the case where they might occur, such as a live sports broadcasting, the Content Store should be deployed as we have discussed in Section 5.4.4. To make the worst case manageable, we assume there is a Content Store. With a proper CS caching policy, Data packets corresponding to duplicate fingerprint entries are cached. Thus the duplicate Interest requests that arrive during T_{rtt} will be inserted into the PIT and then forwarded, while the ones arriving after T_{rtt} will be satisfied by the CS. The content needs to be cached for at least

T_{exp} long, so that the PIT entry can expire and get deleted. In this case, the Content Store size is $r \times T_{exp}/T_{rtt}$, where r is the number of names that have duplicate requests during T_{rtt}. In this scenario, each duplicate fingerprint entry stays for at most $T_{rtt} + T_{exp}$ long in the PIT. The worst case is that every content is requested by exactly two faces. Thus we have the memory size as

$$M_{total} = \left(f_1 + \frac{P_{total} - f_1}{2} \times \frac{T_{rtt} + T_{exp}}{T_{rtt}} \right) \times \frac{M_{bucket}}{ld}. \tag{5.4}$$

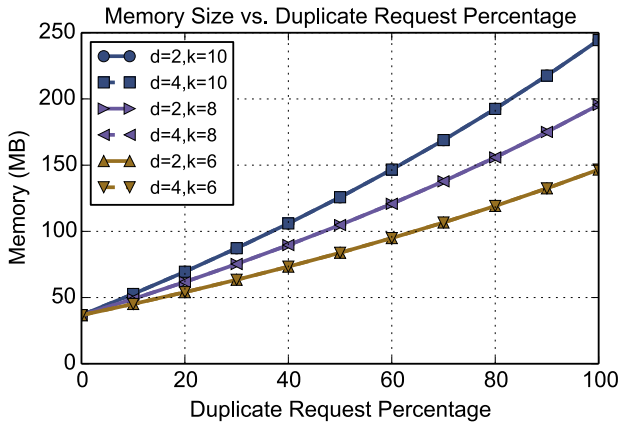


Figure 5.4: Memory Requirement

We define $T_{life} = T_{rtt} + T_{exp}$ as the Interest lifetime for the collided fingerprint. In addition, we let $T_{life} = k \times T_{rtt}$ and call k the lifetime factor. Figure 5.4 presents the memory requirement with different numbers of subtables, lifetime factors, and duplicate request traffic percentages. In Figure 5.4, we use a 100 Gbps link, assuming the Interest and Data packet sizes are $S_I = S_D = 100$ bytes, $T_{rtt} = 80$ ms, load factor $ld = 75\%$, the lifetime factor k is increased from 6 to 10, and the duplicate request percentage is increased from 0% to 100%. The fingerprint length is set to 16 bits, and the number of subtables d is set to 2 and 4. In both cases, the memory size increases almost linearly as the percentage of duplicate traffic

increases. The memory requirement of 4-left hash tables is slightly higher than the one of 2-left hash tables due to a higher fingerprint collision rate. The lifetime factor k increases the memory requirement linearly at each duplicate traffic percentage. With $k = 10$, in the ideal case when there is no duplicate Interest traffic, the memory requirement is 5.34% of the original hash table that stores name strings with the same 75% load factor; in the worst case that all traffic is duplicate, the memory requirement could be as high as 35.51% of the original. At 100 Gbps, the ideal case requires 36.77 MB, which can be stored in SRAM; and the worst case needs 244.44 MB, which can be implemented using RLDRAM.

Zipf-like Fingerprint Distributions Zipf-like distributions have been observed in many network activities, such as the content request patterns to web caching proxies [9]. We use Zipf-like distributions to simulate the Interest fingerprint distribution. Zipf-like distributions are configured with an exponent characterizing parameter, S. And we have $f_i \times i = P_{total}/(H \times i^S)$, where $H = \sum_{i=1}^{16} 1/i^S$. Four Zipf distributions are used so that we could see the trend of the memory requirement, and the real-world cases are more likely to be covered. Table 5.2 lists the percentage of unique fingerprints and the memory requirement with $d = 4$ and $k = 10$ at 100 Gbps. The memory sizes are also compared with the original hash table and the ideal case. From Table 5.2, when $S = 1$, which yields the largest memory requirement, the memory size is 14.7% of the original hash table. Thus in practice,

Table 5.2: Memory Requirement of Zipf-like Fingerprint Distributions

S	**1**	**2**	**3**	**4**
Unique (%)	29.6	63.1	83.3	92.4
Memory (MB)	101.1	79.1	58.6	47.4
vs. Original (%)	14.7	11.5	8.5	6.9
vs. Ideal	2.7x	2.2x	1.6x	1.3x

the PIT is lightly loaded when the worst-case memory size, 35.51% of the original hash table, is configured.

Dynamic Expiration Time Management Configuring an appropriate expiration time T_{exp} is important in practice since it has a linear effect on the memory requirement. A longer T_{exp} increases the memory requirement, while it provides a stronger guarantee to receive potential multiple Data packets. The lifetime factor k is set to 10 in Figure 5.4 and Table 5.2 because we believe $T_{exp} = (10 - 1) \times T_{rtt}$ is long enough for the Data packets to arrive. In practice, the values of T_{rtt} vary for packets that going to different websites. Popular websites usually support reliable short response times, and $T_{exp} = 9 \times T_{rtt}$ could be overkill. To optimize the PIT memory requirement, T_{exp} can be dynamically configured for each PIT entry. The strategy layer introduced in NDN [32] could provide T_{rtt} for each name prefix, thus each name prefix could maintain a recommended T_{exp}. For instance, T_{exp} can be reduced to 2 or 3 times of T_{rtt} for popular websites. In the case where the content cannot be fetched quickly on the server, the pending Interest would expire, but the application will resend an Interest request. And this time, the object could be retrieved quickly by the server since it has just been requested.

Network Traffic Overhead

Network traffic overhead is introduced due to the relaxation of Interest aggregation and fingerprint collisions. Since the Interest packets are always forwarded, the Interest traffic overhead is $T_I = P_{total} - N_{total}$. It should be noted that the Interest traffic never exceeds the link capacity, since the link rates are configured to support the case where every Interest is unique. The Data traffic could exceed the link capacity, but as we will show, the overhead

is very small. The Data traffic overhead is caused by fingerprint collisions. Assuming each Interest name is requested by p faces, and that q packets collide, then the total traffic overhead for these q packets is bounded by $p \times (q-1) \times q$. To provide an estimation, we use the average number of duplicate Interest requests, P_{total}/N_{total}, for each name. As described earlier, the probability of having i colliding fingerprints is $\binom{24}{i-1}/2^{w\times(i-1)}$, thus we have

$$T_D \approx P_{total} \times \sum_{i=2}^{\infty} \frac{\binom{24}{i-1} \times (i-1) \times i}{2^{w\times(i-1)}}. \tag{5.5}$$

When 16-bit long fingerprints are used, the equation is reduced to $T_D = 7.3281 \times 10^{-4} \times P_{total}$. The total amount of additional traffic in the network compared with the ideal design is $T_{total} = T_I \times S_I + T_D \times S_D$, where S_I is the Interest packet size and S_D is the Data packet size.

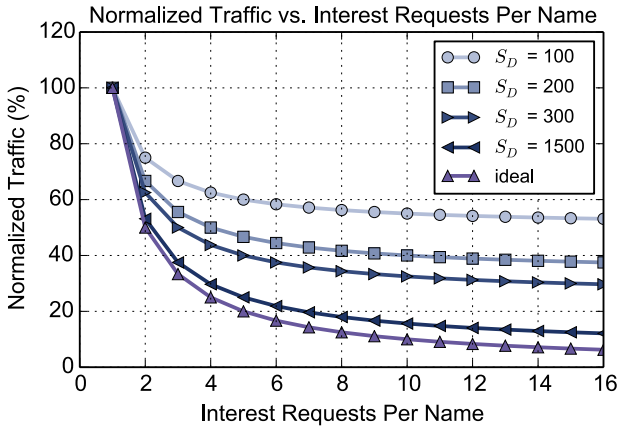


Figure 5.5: Network Traffic Overhead

Figure 5.5 shows the normalized network traffic of the proposed design and the ideal case, where Interest aggregation and Data multicast are supported by storing name strings. The

traffic is normalized to the Internet traffic for delivering the same amount of content, and neither Interest aggregation nor Data multicast is supported in the Internet case. The difference between the traffic of the proposed design and the ideal case shows the overhead. From Figure 5.5, the traffic overhead increases as α increases, mostly due to the Interest traffic overhead. In the worst case, when $\alpha = 16$ and $S_D = 100$ bytes, the normalized traffic of the proposed design (53.16%) is 8.51 times of the ideal case (6.25%). The traffic overhead decreases as S_D increases. When $\alpha = 16$ and $S_D = 1500$, the traffic is 1.95 times of the ideal. The Data traffic overhead increases as α or S_D increases, and the peak Data traffic overhead is 1.10%, where $\alpha = 16$ and $S_D = 1500$. When $\alpha = 1$ and $S_D = 1500$, the normalized traffic of our design is 100.069%, which exceeds the link capacity by 0.069% due to Data traffic overhead. It is worth noting that the average number of requests α generally is very low, thus the traffic overhead would be small. Moreover, the proposed design overloads the link capacity by at most 0.069%, because the ideal design also needs to support the case where every Interest is unique. When the Interest traffic overhead is a concern, optimization methods, such as storing names for popular requests, can be applied.

Table 5.3: Traffic Overhead of Zipf-like Fingerprint Distributions

S	**1**	**2**	**3**	**4**	**Data**
Unique (%)	29.6	63.1	83.3	92.4	NA
100 B (%)	56.8	16.1	5.5	2.2	0.078
200 B (%)	37.9	10.7	3.7	1.5	0.104
300 B (%)	28.5	8.1	2.8	1.2	0.117
1500 B (%)	7.2	2.1	0.8	0.3	0.147

We again use Zipf-like distributions to study the traffic overhead in general networks. Table 5.3 lists the percentage of the traffic overhead compared with the ideal case using multiple Zipf distributions and Data packet sizes. The maximum Data traffic overhead is also listed in Table 5.3. When the exponent characterizing factors S is greater than 2, the traffic overhead

is always less than 16.1%. The overhead is always small with large Data packets. In addition, the peak Data overhead is 0.147%, which is negligible in most network environments.

Segregated PIT Analysis

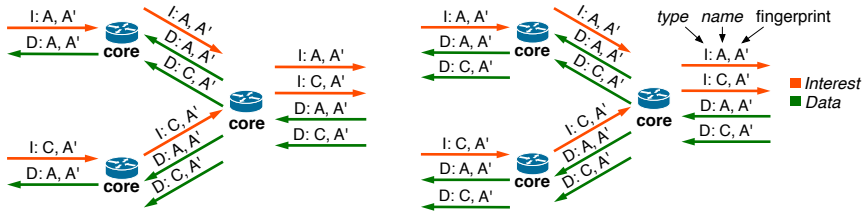
The segregated PIT design reduces the memory requirement for each smaller PIT since the number of entries is only $1/s$ of the single PIT. Moreover, the entry size of the segregated PITs is also smaller because the face list length is reduced to $\lceil n/s \rceil$. However, the fingerprint length needs to be increased slightly. Assuming the fingerprint collision rate of a single PIT is f_p, then the fingerprint collision rate of the segregated PITs is $1 - (1 - f_p)^s \approx s \times f_p$. To maintain the same overall fingerprint collision rate, the length of the fingerprints stored in the smaller PITs needs to be $\lceil log(s) \rceil$ bits longer. Thus, each PIT entry size is reduced by $n - (\lceil log(s) \rceil + \lceil n/s \rceil)$ bits. For instance, when a 16-face router is divided into two smaller PITs, the PIT entry size is reduced by seven bits. On the other hand, the memory bandwidth of the segregated PITs needs to be increased by s times, because all s segregated PITs are queried when Data packets arrive.

5.5 The Case with Multiple Core Routers

Previous sections have presented the design and analysis with the assumption that there is only one core router in the network, which is connected with multiple routers that perform both Interest aggregation and Data multicast. In this section, we discuss the issues with supporting multiple core routers in a network and analyze on the memory requirements and network traffic overhead.

5.5.1 Supporting Multiple Core Routers

The issues with supporting multiple core routers arise because routers are aware of only local fingerprint collisions and have no knowledge of remote fingerprint collisions that happen at other forwarding nodes along the path. For example, consider an Interest packet with name `A` that does not have a collision at the current core router 1, but collides with another Interest packet with name `C` at the next hop core router 2. In this case, both Data packets A and C are delivered to the core router 1 because the PIT entry is marked as collided. If the same hash function is used at core router 1, then the first arriving packet removes the PIT entry right away because there is no known collision locally. However, this Data packet is not necessarily the one that is actually requested, and the requested packet will not be delivered to the consumers because the PIT entry has already been removed when the Data packet finally arrives. Figure 5.6(a) shows an example that Data packet with name `C` is never delivered.



(a) Data packet C is not delivered

(b) Data packet C is delivered because packets A and C are marked

Figure 5.6: Supporting Multiple Core Routers

To solve this issue, essentially each router needs to be aware of collisions that occur on upstream paths. As a result, one solution is that, when a Data packet is delivered from a core router, whenever its corresponding PIT entry is being marked as collided, a special flag needs to be marked in the Data packet to indicate that this packet has encountered a

collision. When a Data packet is marked as collided, the core router keeps the corresponding PIT entry until it expires, regardless of whether it has been marked as locally collided or not. Thus, if a fingerprint collision happens on a core router, then all of the downstream routers keep the corresponding PIT entries for the colliding packets until they expire, in case there is potentially another Data packet being delivered. Figure 5.6(b) shows that Data packets with name `A` and `C` are both delivered because the corresponding flags are marked in the packets.

5.5.2 Analysis

Memory requirements The PIT memory requirements for multiple core routers are higher that those for a single core router case. Two factors contribute to the increased memory requirements.

The first factor is the fingerprint collision rate. The effective fingerprint collision rate is higher because an Interest packet may experience collisions at every core router it goes through. To mitigate this issue, the fingerprint length can be increased to reduce the fingerprint collision rates. But overall, the additional memory required to handle this type of collision is small.

The second factor, duplicate traffic, has a larger impact on memory size, especially in the worst case. In the worst case, every Interest has a duplicate request in the core network, and it is likely that the collision would happen closer to the content publisher. For most of the downstream routers, the returned Data packet is marked with an additional flag indicating that it has encountered a collision. Hence, every entry in the PIT will be marked as collided.

Recall that the worst case for the single core router is that, each arriving Interest packet has exactly two copies: Hence, the number of entries occupied in the PIT is 50% of its full capacity. In the case with multiple core routers, the worst-case memory requirement is double that of the single core router case.

In reality, because of Interest aggregation at edge routers, the duplicate traffic in the core network is expected to be reasonably low. As a result, it is likely that the memory requirement can be lower in the worst-case analysis.

Traffic overhead The additional traffic is introduced by only Data packets. The reason is that Interest traffic is not affected, because all of the Interest packets are already forwarded in the core network.

As the case with a single core router, the additional duplicate Data traffic is caused by fingerprint collisions. The duplicate Interest requests are still aggregated by edge routers before they reach the content publishers. As a result, the Data traffic entering the core network is not affected, but the additional Data traffic is determined by the hash collision rates. Although the hash collision rate is higher when more hops are went through, the overall impact is expected to be small. In addition, additional bits can be allocated for each entry to reduce the hash collision rates.

5.6 Performance

In this section, we evaluate the performance of the proposed Pending Interest Table design.

5.6.1 Experimental Setup

We implemented the fingerprint-only Pending Interest Table in C++. The hardware platform used was a machine equipped with eight Intel Xeon E5540 cores, 8 MB of L3 cache, and 12 GB of DDR3 memory. Because real-world NDN core router traces were not available, a synthetic Interest trace was generated by appending a number ranging from 0 to 999 to each domain name in the Alexa top 1 million websites [2].

5.6.2 Simulation

We measured the fingerprint collision rates and the number of overflowed names with different PIT size N, and different numbers of subtables d. Each bucket has $E = 8$ entries, the PIT load factor ld was set to 75%, 16-bit fingerprints were used, and the lifetime factor k was set to 10. In the experiment, m names were inserted in the first phase. A certain percentage, denoted as d_p, of the names are not unique (requested twice), and d_p was set to 0%, 20%, 40%, 60%, 80%, and 100%. The value of m was adjusted accordingly for each d_p so that the targeted number of occupied entries in the PIT was always close to $ld \times N$. In the second phase, names were updated; and the i^{th} name was deleted and then the $(i+m)^{th}$ name was inserted, where $i \in \{0...25 \times m\}$. In the simulation, we use the Interest name index, i, as a timing tool. The T_{life} is configured to be the number of names, m_{10}, inserted during $10 \times T_{rtt}$. The time at any point was set to the index of the latest inserted name. Thus, an Interest with index i would be considered as expired once the $(i + m_{10})^{th}$ name was inserted. The largest measured fingerprint collision rates, denoted as f'_p, and the number of the overflowed names, denoted as M, are listed in Table 5.4. When the PIT has $d = 4$ subtables, it never

overflowed, thus its overflow size is not listed. In the table, the PIT with size $1,048,576$ is denoted as $1M$.

Table 5.4: Fingerprint Collision Rates and Overflow Sizes

Type	**d**	**1M**	**2M**	**4M**	**8M**	**16M**
$f'_p(10^{-4})$	2	1.79	1.78	1.77	1.77	1.75
	4	3.55	3.53	3.56	3.57	3.53
M(10^5)	2	0.08	0.17	0.34	0.67	1.32

From Table 5.4, the fingerprint collision rates in both $d = 2$ and $d = 4$ cases are close to the theoretic values, which is expected. When there are $d = 4$ subtables, overflow never occurs as each inserting name has 32 choices. But when $d = 2$, each inserting name has only 16 choices, overflow occurs as names being updated. The overflow size increases linearly as the PIT table size increases. When the PIT size $N = 16$ million, 12.53 million names are stored, and the overflow size is 132.23K. Even in this case, the overflow table memory requirement is less than 0.73 MB, which can be stored on SRAM or TCAM.

5.6.3 Latency Measurement

We measured the PIT operation latency of the software implementation of the proposed PIT. The experimental setup was the same except that we fixed d_p at 0% since this case required the highest update frequency. Hash values were computed using the 64-bit CityHash function [15]. Figure 5.7 shows that the average latency of each operation increases as the PIT size increases. The average latency with 4 subtables is approximately twice of the case with 2 subtables, since the number of memory accesses is doubled. In hardware designs, multiple words can be read from SRAM chips in parallel, which reduces the latency. At 1 Gbps, the PIT size is close to $N = 65,536$, and the measured latency is about 1.2μs.

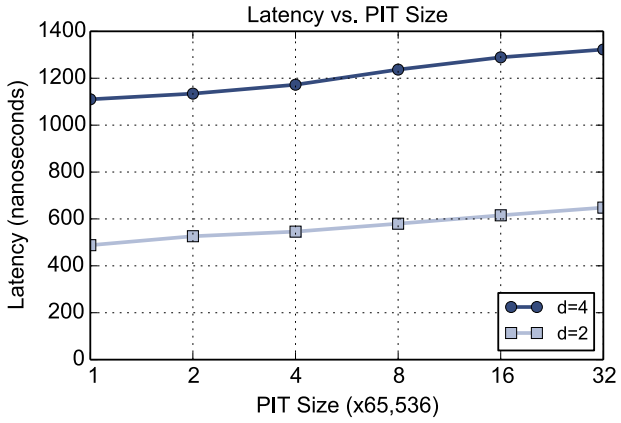


Figure 5.7: PIT Operation Latency

Note that the performance can be improved further using software optimization techniques, such as memory prefetching and large pages, that we have explored in the previous two chapters. As pointed out in [65], secure hash functions are better to be employed for the PIT and CS. Employing secure hash functions in software increases the update latency, but the performance can be improved via hardware-based hashing units.

5.7 Discussion

In this section, we discuss the modification of the proposed Pending Interest Table when false positives are allowed, and also consider the PIT security issues.

5.7.1 Allowing False Positives

In networks where a small number of false positives are allowed, the proposed fingerprint-only Pending Interest Table can be modified to support Interest aggregation, just like name

strings are being stored. This way, only one of the colliding Interest names is forwarded and the rest are dropped. The false positive rates of the fingerprint-only PIT in this case are the same as the previous analysis on the case with no duplicate Interest requests. At 100 Gbps, with the same configuration, the memory requirement of the modified PIT is 36.65 MB, slightly less than 36.77 MB, because collided fingerprints do not stay longer in the table. Bloom filters have also been considered to implement PIT when false positives are allowed [84][85]. In these designs, a Bloom filter is deployed for each face, therefore the Interest aggregation is not supported. Overall, when false positives are allowed, the fingerprint-only PIT is preferred because Interest aggregation can be supported, and the expiration time and face list can be easily stored for each entry.

5.7.2 Pending Interest Table Security

The proposed Pending Interest Table is designed for the worst-case flow balance mode, but it will not be able to handle an Interest flooding attack, where every Interest packet stays for T_{exp}. In this case, statistics collected on the routers should be able to detect the attacks and then apply countermeasures. In addition, our proposed architecture could potentially address the Interest flooding problem by designing a fingerprint-only PIT that is large enough to hold all the flooded packets. The memory requirement may still be acceptable since only a fixed-length fingerprint is stored for each Interest.

5.8 Related Work

Prior hash-table based works on scalable PIT designs all support Interest aggregation in both the edge and core networks, while our solution is the only one that relaxes the Interest aggregation feature. The principles of scalable PIT designs are highlighted in [89]. The hash-based methods in [65] store both fingerprints and name strings in the PIT. Although string comparison can generally be avoided by checking their fingerprints, at least one DRAM write operation is required to store the content name. Encoding methods [17] have been proposed to reduce the PIT memory size, while additional lookups that encode the content name are required before querying the PIT. Moreover, the transition arrays need to be dynamically updated, increasing the complexity of the system. The performance of the encoding methods and hash-based methods are compared in [72]. When false positives are allowed, our design can be modified and supports Interest aggregation as described in Section 5.7; Distributed Bloom filters [84] [85] have also been used as PITs in this context.

5.9 Summary

Fast and scalable Pending Interest Table design is a challenge in Named Data Networking. In this chapter, we propose a Pending Interest Table design that guarantees packet delivery and significantly reduces the memory requirement by storing fingerprints rather than name strings. The Interest aggregation feature in the core routers is relaxed so that packets are guaranteed to be delivered even when fingerprint collisions occur. We have studied the memory requirement and network traffic overhead analytically, and demonstrated that the

additional network traffic is acceptable. We have also measured the performance of a software implementation of the proposed design.

Our results show that 37 MB to 245 MB are required at 100 Gbps for a single router case, so that the PIT can fit into SRAM or RLDRAM chips. When multiple core routers are used, the memory requirements are twice of that of a single core router. As a result, 74 MB to 490 MB are required in the worst case, which is still more memory-efficient than a standard hash table-based design.

Chapter 6

In-network Caching Elements

In this chapter, we focus on discussing the feasibility of implementing in-network caching elements in NDN.

The Content Store (CS) is a temporary packet storage residing in the NDN forwarding plane. Its basic functions are similar to those of the packet buffers in current IP routers and to web caching proxies. The CS is equipped with an indexing structure that supports efficient lookup methods and cache replacement policies. Based on our experience with the FIB and PIT designs, the CS indexing data structure can be implemented efficiently using hash tables. As a result, in the first part of this chapter, we discuss the requirements of the Content Store and present data structure designs to support CS lookups and cache-replacement policies, including both Least Recently Used (LRU) and Least Frequently Used (LFU) policies.

In contrast to the CS, the NDN Repository (Repo) is a long-term persistent content storage. A Repo can be deployed for a specific application, where the Repo acts like a database; it can also be analogous to a content distribution node in current content delivery networks (CDNs) to server large Interest requests. Thus, the Repo is required to support fast content retrieval

and needs to be scalable to store large numbers of pieces of content. As a result, the Repo requires efficient storage systems, which already exist in IP networks, such as NoSQL databases and CDN storage systems. Thus, in the second part of this chapter, we demonstrate how existing storage systems can be employed by the NDN Repo implementations. Specifically, we present the performance results of the open-source NDN Repo implementation, Repo-ng, using Redis [59], a popular key-value store, as the backend storage.

6.1 Content Store

Devices with network interfaces must maintain a set of packet buffers to receive incoming frames and to prepare outgoing frames for transmission. The Content Store in NDN utilizes these packet buffers for these reasons, but also maintains an index over the data names associated with each packet buffer. While traditional networking stacks empty each packet buffer after receipt, the Content Store in NDN keeps the content until the packet buffer needs to be repurposed to hold other data. Compared to IP routers, in which packet buffers merely hold the frames while packets are being processed, the additional Content Store index in NDN enables serving Interest requests with the stored packet buffers in forwarding nodes, improving the efficiency of network infrastructure.

Improving packet buffer performance in IP networks has been studied extensively [3, 20, 31, 39]. Previous studies focused on reducing the memory requirements and more efficient use of the memory bandwidth. These techniques can be applied directly in NDN forwarding designs. Cache-replacement policies have also been studied extensively in contexts such as web caching. We focus on LRU or LFU policies because they can be implemented efficiently in software. Existing NDN router designs mostly choose to implement the Content Store in

memory [54, 65], just like the packet buffers in IP. Recent designs that leverage solid state drives (SSDs) to provide a larger secondary storage space have also been proposed [43, 64]. In the aforementioned designs, the lookup structures are typically implemented as hash tables because they are simple and efficient. In addition, analytical modeling of the CS has also been studied to characterize the caching behavior and benefits [18]. Our work in this chapter focuses on designing an efficient indexing structure that supports both fast lookups and cache-replacement policies.

In the rest of the section, we first discuss the memory requirements of the Content Store, and then discuss the Content Store lookup data structure design. Finally, we present the designs that support cache-replacement policies.

6.1.1 Requirements

The two major requirements for Content Store designs are *memory size* and *line-rate operation support*. We follow the design principle that core routers and edge routers should have different goals, and we focus on Content Store design for the core routers in this dissertation. Intuitively, the odds of getting a cache hit in core routers should be smaller than in edge routers, because duplicated requests are aggregated and possibly satisfied in the edge networks. In addition, as we have mentioned in Chapter 1, a recent study [24] reported that the benefits of ubiquitous caching in information-centric networking can be largely realized by caching content in edge networks, which supports our intuition. As a result, the Content Store size in core routers can be small, and the routers could even operate without a Content Store. In general, a small Content Store is still preferred because popular content distribution, such as live streaming, can be supported more efficiently. In addition, to support our

fingerprint-only PIT design presented in Chapter 5, the core routers need to employ a small Content Store to cache the Data packets that have been requested multiple times.

Memory Size. The Content Store includes packet buffers and a lookup structure that indexes the cached packets. The memory requirements of both the packet buffers and lookup structure are proportional to the number of packets in the Content Store. When all the Data packets are cached, the memory requirement is determined by the link rate and how long each packet is cached. Assuming the link bandwidth capacity is C and the average packet caching time is T_{cache}, the memory of the packet buffers is close to $C \times T_{cache}$ because Interest packet sizes are relatively small. The memory requirement of the lookup structure is much smaller than that of the packet buffers, because the packet names stored in the lookup structure are much shorter than the entire Data packets. Moreover, the lookup structure can store the memory address of the name string in the packet buffer for each packet, reducing the lookup structure memory requirement further.

Assuming the fingerprint-only PIT design is employed, then T_{cache} should be at least $T_{rtt} \times k$ long, so that a duplicated fingerprint PIT entry is guaranteed to expire as discussed in Chapter 5. In the worst case, when all the traffic is duplicated, all the Data packets must stay in the Content Store. Hence, the memory requirement is $C \times T_{rtt} \times k$. When $C = 100$ Gbps, $T_{rtt} = 80$ ms, and $k = 10$, the memory requirement is 10 GB. This requirement is about k times larger than the current IP router packet buffer, whose size is usually $C \times T_{rtt}$ [3]. Designing a scalable packet buffer for IP routers has been investigated [20, 31, 39], and the results can be applied to the Content Store design. What is more, the worst case discussed above is unlikely to happen in the core routers, and k can be dynamically configured to smaller values for prefixes with reliable T_{rtt}. Hence, the minimum Content Store memory size can be smaller in practice.

The actual Content Store size eventually is determined by the availability of the physical resources and the benefits of having a larger CS. In addition, the required memory size is also related to how packet buffers are allocated. For instance, if the same packet buffer size is used, then the memory requirements are larger than what is calculated here, because a large amount of memory is wasted if the packets are small.

Line Rate Operation Support. The Content Store is part of the packet processing pipeline and therefore needs to support line rate operations, including name lookups and cache replacements. It is worth noting that the original Content Store in the NDN reference design supports additional features such as lexicographical-order lookups, which allows multiple versions of a content be stored and indexed with the same name. Supporting this feature increases design complexity and the benefit is not obvious in core networks; thus, we consider only exact name lookup in the Content Store for core routers. An effective cache-replacement policy and its implementation are also crucial for the Content Store design. On the arrival of a Data packet, the cache-replacement policy determines if this Data packet should be cached. During an insertion, if the Content Store is full, then a packet needs to be evicted to make space.

6.1.2 Content Store Data Structures

Packet Buffers. When the link speed is 1 Gbps, following the previous worst-case analysis, the packet buffer size is 100 MB, which can be implemented with a single DRAM module because DRAM access time (around 50 nanoseconds) is shorter than the packet inter-arrival time. If the line rate is higher than 10 Gbps, the packet buffer size is more than 1 GB, and has to be implemented using DRAM because SRAM is too small. However, the DRAM

access time is longer than the packet inter-arrival time. As a result, parallel DRAM accesses must be employed. For general-purpose multicore processor platforms, multiple memory controllers and channels provide sufficient bandwidth for packet buffering.

The differences between packet buffers in NDN and the ones in IP are twofold. First, packet buffers are not immediately released when packets are transmitted: Packet buffers cannot be recycled if they are cached in the Content Store. When a Data packet is evicted from the Content Store, its packet buffer is also released. Second, packet buffers are indexed and can be used to satisfy Interest packets.

Lookup Structure. The lookup structure indexes the cached packets and maintains the statistical information required by the cache-replacement policies. We call this lookup structure the *Content Table*. The Content Table needs to support fast lookups and updates, similar to the design goals of the Pending Interest Table. Our fingerprint-based PIT has shown good performance results; hence, we propose a Content Table design based on the d-left hash table.

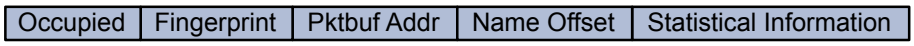


Figure 6.1: Content Table Entry

Figure 6.1 shows that in each Content Table entry, an Occupied bit denotes that this entry is occupied, so that only occupied entries are visited during a lookup. A fingerprint of the packet name is stored to avoid unnecessary and expensive string matching operations. When there is a fingerprint match, the name of the stored packet can be retrieved using the packet buffer memory address and the Name offset fields. The Statistical info field stores

the information, such as last reference time and number of references, required by the cache-replacement algorithms. The lengths of these fields depend on the configuration used in practice.

Fingerprint-collisions are allowed in the d-left hash table based design, and multiple entries sharing the same fingerprint within a hash bucket is possible. Overflow can be handled by introducing an additional overflow table, as in the PIT design discussed in Chapter 5. In addition, a packet that belongs to the same hash bucket as the inserting packet can be evicted to make space.

6.1.3 Supporting Cache Replacement Policies

Fast and efficient cache-replacement policies have already been studied extensively [56]. In this section, we present the design of two simple caching-replacement policies as examples that can be implemented efficiently in software.

Least Recently Used (LRU) Policy. The LRU cache-replacement policy can be implemented with $O(1)$ operation complexity using a doubly linked list and a hash table. In an LRU cache, whenever there is a cache hit, the corresponding node is moved to the head of the doubly linked list. When a Data packet needs to be inserted, it either becomes the head node or the current tail node is evicted from the hash table and the newly inserted Data packet becomes the head of the linked list. Figure 6.2 shows a Content Table entry and a doubly linked list that implements the LRU policy.

The hash table is keyed by fixed-length fingerprints of Data packet names, and each hash table entry stores the packet buffer memory address, the offset of the name field, and the

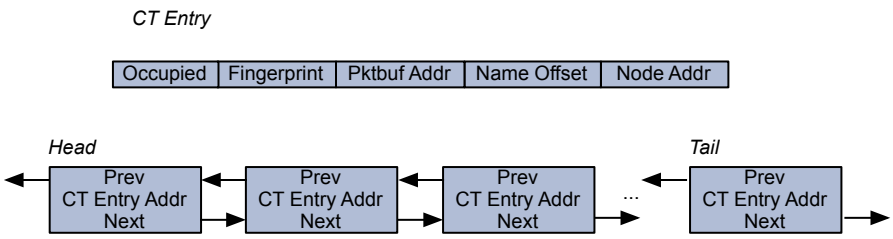


Figure 6.2: Doubly Linked List for the LRU Policy

memory address of the corresponding node in the linked list. The hash table can be implemented as either a low-load factor hash table with chaining to resolve hash collisions, or a d-left hash table. For software-based implementations, similar to our proposed FIB and PIT design, each hash bucket has the size of a cache line, e.g., 64 bytes.

During a lookup, the Data packet names are compared only when fingerprints match. To insert a Data packet into the Content Store when it is full, the Data packet corresponding to the tail node is evicted from the CS, and then the new Data packet is inserted into the hash table. The hash table entry and the tail node are updated accordingly. The tail node then becomes the head of the linked list. All of these procedures require constant number of operations.

Least Frequently Used (LFU) Policy. To support the LFU policy, each hash table entry is associated with a counter. When a Data packet needs to be evicted to make space for a new packet, the Data packet with the smallest counter value is evicted. When the number of items stored in the Content Store is large, finding the hash table entry with the smallest counter value is challenging. As a result, we present a design that performs LFU cache-replacement within a small number of candidates. For instance, the search range can be

restricted to the items visited during a hash table insertion operation. We call this scheme as local searching.

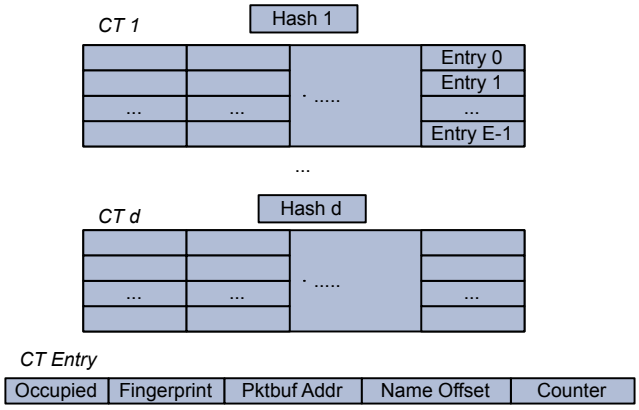


Figure 6.3: Data Structures for the LFU Policy

As an example, with the d-left hash table design, for each insertion, $d \times E$ entries are visited, where E is the number of entries in each hash bucket. If the Content Store is full, the entry with the smallest reference count among these $d \times E$ entries is evicted. In fact, local searching can also be applied to support LRU policies where each entry stores a timestamp of the most recent visit. Similarly, the local searching scheme can be extended to other performance metrics.

In addition to the two replacement policies, selective caching can be employed. The information stored in the PIT can be used to select the packets to be cached. For example, only packets that have been requested more than k times can be cached.

6.2 Repo

In this section, we present performance studies of a reference NDN Repo implementation, named Repo-ng, and of a modified Repo-ng that leverages Redis [59], a popular key-value store, as the backend storage. We demonstrate that existing large-scale key-value stores can be employed for NDN Repo implementation. Employing key-value stores directly as the backend storage not only reduces the burden of developing a scalable NDN Repo from scratch but also enables a smooth transition from IP network storage systems to NDN in-network caching elements.

An NDN Repo is a long-term persistent content storage, and applications can push content into the Repo. A Repo can be a process that runs in the same node as the application, or it can be a dedicated machine or a cluster of machines. The Repo registers name prefixes to neighboring NDN routers so that Interest packets that match those name prefixes are forwarded to the Repo. The content stored in the Repo can serve Interest requests from consumers directly. For instance, the NDN video application [37]and its successors, NDNlive and NDNtube [76], stream both live and prerecorded video by pushing the video content into a Repo. Content consumers fetch the video from the Repo directly, and the video content is stored in the Repo until it is deleted. As video content sizes are large, a large persistent in-network storage element, i.e., a Repo, is required to support video distribution. The Content Store, because of its limited size and temporary nature, is only able to help distribute Data packets when they become popular.

The Repo has two fundamental requirements. First, the NDN Repo protocol [47] needs to be supported so that applications are able to send commands to the Repo. For instance, to store a piece of content, an application sends a signed Interest packet to the Repo, then the

Repo processes this command and sends Interest packets to the application to fetch Data packets. These fetched Data packets are then stored in the Repo. Second, the Repo needs to be scalable to store a large amount of content. When a Repo is employed for a specific application that does not publish much content, a Repo with a small capacity may work fine; but if a Repo is used as a content distribution node in IP networks to deliver massive amount of content, then it needs to have a large storage space and support high bandwidth.

The Repo supports read and write operations. The read operations fetch Data packets from the Repo to satisfy Interest requests, and the write operations push Data packets into the Repo. The performance requirements for these two operations depend on the specific use cases, and in general a more frequently used operation should have better performance. For a general-purpose Repo implementation, we expect that read and write operations would have comparable performance. If the Repo is used as a content distribution node, then the read performance requirement should be higher, because the write operations are less frequent. Take the video application as an example, assuming a 4K resolution video is streamed, because the bitrate is 24-35 Mbps [27], about 2-3K write operations are required every second with 1500-byte packets. The read performance requirement is similar for a single stream, but the performance requirements increase linearly when multiple videos are streamed. In general, the NDN Repo performance requirements should be comparable to those of the databases and storage systems used in today's web.

A reference implementation of the NDN Repo, namely Repo-ng, has been implemented by the NDN team. Repo-ng supports the Repo protocol and provides a flexible framework to experiment with different types of backend storage systems. In other words, Repo-ng handles control messages and provides interfaces to handle operations related to the backend storage systems. The Repo-ng implementation employs SQLite, a self-contained, server-less SQL

database engine [68]. In addition, like the CS, the Repo supports advanced Interest selector schemes so that the most suitable Data packet is returned when multiple candidates can satisfy the request. If the Interest request is directly queried in the database, supporting the Interest selector scheme requires multiple database accesses when more than one candidates exist, thus increasing the processing time. To reduce the database operation overhead, Repo-ng maintains an indexing data structure that keeps the names of the stored Data packets. This way, the Interest selectors can be processed without requiring database operations, and only the most suitable Data packet is fetched from the database. Repo-ng has been deployed for several NDN applications, however, its performance has not been discussed in the literature. Thus, we present the performance of the original Repo-ng in this section.

NoSQL databases, such as Redis, Cassandra, MongoDB, and Memcached, have become used widely as caching or storage elements in popular web services [49]. Due to the simple lookup operations required for the backend storage of Repo-ng, NoSQL databases can be employed. Compared to SQLite, used in the original Repo-ng, NoSQL databases can perform better because they run as stand-alone programs and support non-blocking I/O operations by design. In this section, Redis is used as an example to demonstrate the performance of a NoSQL-based Repo-ng implementation. We first measure the performance of a Redis-based Repo-ng on a single node, where the performance bottleneck is the NDN Forwarding Daemon (NFD) [1] and the original Repo-ng interface. To demonstrate its scalability, we measure the aggregated throughput from multiple clients to the single Redis backend storage server. We also benchmark the Redis key-value store to understand its maximum throughput.

Our key findings in this subsection are:

- Redis-based Repo-ng implementation performs slightly better than SQLite-based Repo-ng using a single node. With one million packets being processed by a single client thread, the read throughputs for SQLite-based and Redis-based Repo-ng are 5.3K and 5.8K operations per second, respectively. The optimized Redis-based Repo-ng, which removes the in-memory indexing data structure, supports about 6.0K read operations per second. The write throughputs for SQLite-based, Redis-based, and optimized Redis-based Repo-ng are 7.1K, 8.2K, and 11.8K operations per second, respectively. As can be seen, both the read and write performance are low, and the read performance is worse than the write performance due to the performance limits of the client program that fetches data from the Repo. With two client threads, the read and write throughputs are comparable, and the performance bottlenecks for the read and write throughputs are the NFD daemon and the Repo-ng interface, respectively.

- We have benchmarked the single-node Redis performance and 160K read or write operations can be performed every second. We have also demonstrated that the measured Repo throughput can be improved by using multiple clients, each with its own NFD and Repo-ng. Using three clients, each running five client threads, we have demonstrated that the optimized Redis-based Repo-ng supports 15K read and 24K write operations per second. However, to achieve 160K read or write performance, many more clients are required, which is unacceptable in practice. As a result, these performance issues with Repo-ng may hinder NDN application development, and performance optimization is a near-term concern.

6.2.1 Performance Evaluation

We first compare the performance in a single node, where SQLite and Redis reside on the same node as the Repo-ng interface. To demonstrate the scalability of Redis-based Repo-ng, we evaluate its performance with multiple client nodes.

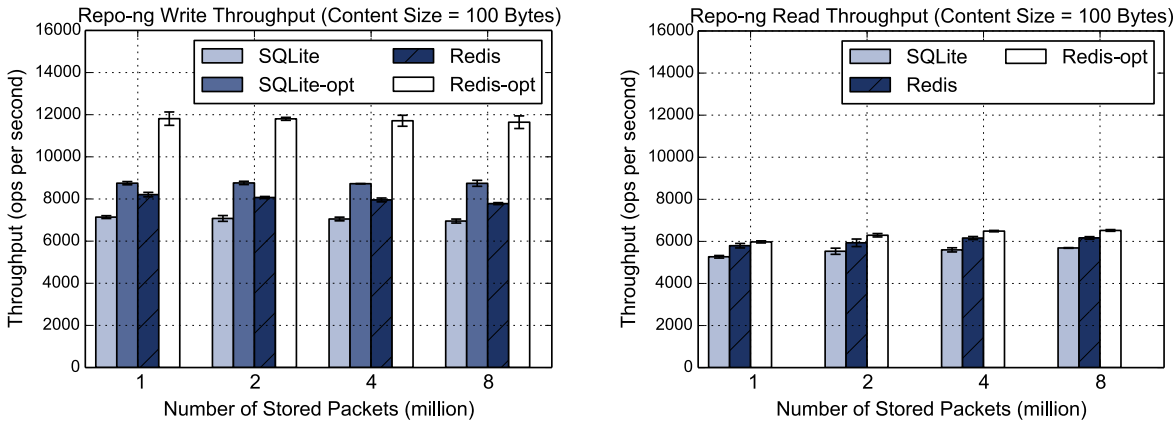
Performance with a Single Node

In the single node scenario, all of the programs reside on the same node. The machine used for the experiments was equipped with two six-core Intel Xeon E5-2630 processors and 64 GB of DDR3 memory.

For both the original Repo-ng and the Redis-based Repo-ng, the same NFD daemon (version 0.3.3) was used, and the Content Store size was configured to be one, so that the CS lookup time could be minimized. For the Redis-based Repo version, the Redis server (version 3.0.2) was connected with the Repo-ng interface via a Unix socket, which has better performance than a TCP connection. The Repo-ng was configured with a capacity of 10 million entries. To study the impact of the additional in-memory indexing data structure, we also restricted the lookup scheme to exact string matching, and thus the indexing data structure could be removed. This way, the backend storage system essentially became a large hash table.

We measured the Repo-ng write and read throughputs separately. To measure the write throughput, we modified the `ndnputfile` program to generate Data packets with random payloads and then insert them into the Repo. The generated Data packets share the common prefix `ndn:example/data/1/hello`, which is the default prefix in the sample Repo-ng configuration file, and each Data packet has its own segment number as the suffix. We started

by inserting one million Data packets, and then doubled the number of packets in each experiment until eight million entries were inserted each time. The size of the content buffer to be written into the Repo is 100 bytes. The read throughput experiments were measured after the write throughput tests. Similarly, we modified the `ndngetfile` program to fetch the Data packets from the Repo. The fetched Data packets were discarded immediately to reduce the client software overhead. For each configuration, the experiment was repeated three times. The average throughputs in terms of numbers of operations per second are presented in Figure 6.4 with 95% confidence intervals. The performance results of the Repo-ng that without the indexing structure are labeled as SQLite-opt and Redis-opt. In addition, the read performance of SQLite-opt is not reported because the in-memory data structure in Repo-ng maps packet names to internal IDs. In the optimized SQLite-based Repo-ng, the packets were still stored using the IDs as the lookup keys. Thus, without the indexing data structure, it was impossible to perform read operations with the SQLite-based Repo-ng.



(a) Write (b) Read

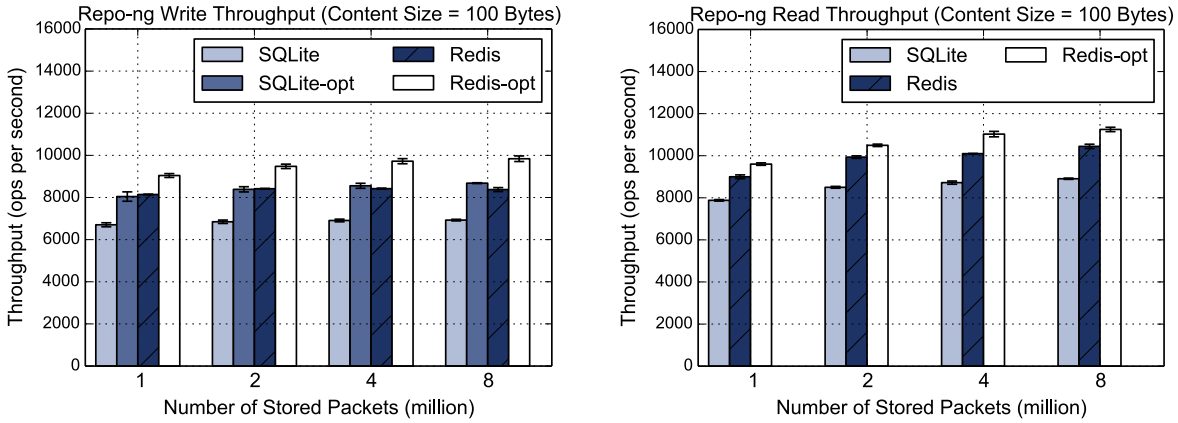
Figure 6.4: Single-node Repo-ng Throughput Performance (Single Thread)

According to Figure 6.4(a), the performance of the optimized version that eliminates the in-memory indexing data structure improved the performance of both the original Repo-ng and

the modified Redis-based Repo-ng. In both cases, the Redis-based Repo-ng outperformed the original Repo-ng. As can be seen, the single-threaded Redis-based Repo-ng can support about 11.8K write operations per second.

The read performance is shown in Figure 6.4(b). Similarly, the throughput of Redis is higher than the one of SQLite, and the optimized Redis version further improves the performance. As can be seen, about 6K read operations per second was achieved. The reason that the read performance is slower than the write performance is that the throughput is limited by the performance of the client program `ndngetfile`. Running more client threads can improve the read throughput, but eventually the throughput will be limited by the performance of the NFD forwarding daemon.

We have also evaluated running two client threads in each experiment. Each client thread runs on a dedicated processor core and handles half of the workload, i.e., 500K packets when one million packets are requested. The processing time is the difference between the earlier starting time and the later ending time of these two client threads.



(a) Write (b) Read

Figure 6.5: Single-node Repo-ng Throughput Performance (Two Threads)

Figure 6.5 shows the measured throughputs when running two client threads in each experiment. As can be seen, the read throughputs are much higher than in the single-thread case, and are comparable to the write throughputs. In addition, the write throughputs are slightly lower than in the single-thread case, which is likely due to the overhead introduced when the Repo-ng interface handles multiple client threads. According to the CPU utilization information, the write throughput is limited by the Repo-ng interface, and the read throughput is limited by the NFD throughput. Hence, read operations are faster than write operations in Repo-ng. In addition, the overall write throughput is expected to be higher if a dedicated Repo-ng process is available for each client thread.

In summary, we demonstrate that employing Redis as the backend storage improves the performance of a single-node Repo-ng. The performance bottlenecks include the Repo-ng interface, the NFD daemon, and the client program, which deserve more optimization. Nonetheless, our experience shows that the Repo-ng can incorporate other storage systems easily.

Performance with Multiple Nodes

In the above experiments with a single node, the performance bottleneck was the Repo-ng interface, the NFD daemon, or the client programs when Redis was used as the backend storage. To explore the scalability of Redis-based Repo-ng, we used multiple client nodes to generate enough requests.

The experiments with multiple client nodes were performed in the ONL. The Redis server ran on the same machine as before. The clients were machines equipped with two four-core Intel Xeon E5520 processors and 12 GB of DRAM. The client nodes ran the NFD daemon

and the Repo-ng interface. The Redis server was connected to each Repo-ng interface via a TCP connection because they resided on different physical machines. The network latency between the Redis server and the client nodes was much larger than that of a Unix socket, so the throughput of the single-threaded Redis-based Repo-ng became much lower. As a result, each client node ran five Repo-ng processes and five client programs to saturate the NFD daemon. It is worth noting that we measured only the optimized Redis-based Repo-ng, i.e., the indexing data structure was removed, because otherwise an additional Redis query is required to get the number of items stored in the database and this number will be the internal ID for the next inserting item. In the optimized Redis-based Repo-ng, the Redis database is keyed by the packet names directly rather than the internal numerical IDs.

As in the single-node experiments, the write and read throughputs were measured separately. For each test, all of the clients had the same configuration, and each client node reads or writes one million packets. Scripts were organized so that the clients started the process at the same time, and the processing time were recorded in each client. When the experiments were done, the longest processing time among all the clients was taken as the processing time. The measured throughputs with multiple clients are shown in Figure 6.6.

Clearly, higher operation throughputs can be achieved with a larger number of client nodes. In Figure 6.6, the performance bottleneck is the NFD implementation because the in-memory index structure in Repo-ng has been removed and each client program is connected to a dedicated Repo-ng process. In addition, the CPU core that runs the NFD process is saturated. The aggregated read throughput is less than the write throughput, thus the cost of processing packets for the read tests is higher than that of the write test in the NFD daemon, which is consistent with our findings in the single-node experiments.

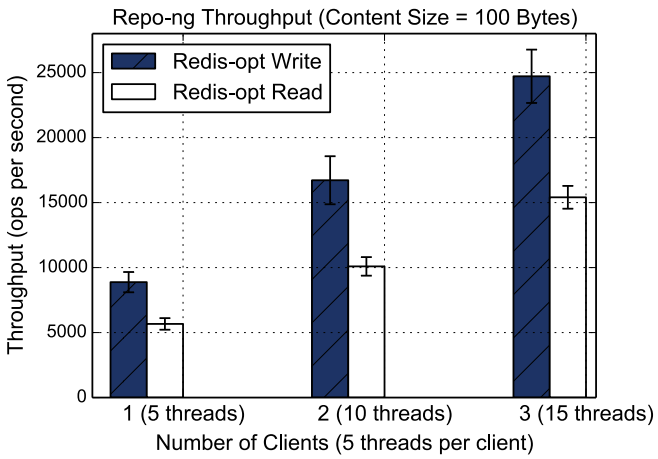


Figure 6.6: Repo-ng Performance with Multiple Client Nodes

Redis Benchmark

To understand the maximum throughput of Redis, we have measured the throughput of the `GET` and `SET` operations using the benchmark tool distributed with Redis. Similarly, the number of keys stored in the Redis server was configured to be one, two, four, and eight million. The size of the value, i.e., content size, was set to be 100 bytes. The Redis server ran on the same machine as before, and the benchmark tool ran on either the same machine as the Redis server or as the client node. In the experiments, the benchmark tool was connected to the Redis server via a Unix socket, a local TCP connection, and a remote TCP connection. In addition, the benchmark tool provides a pipelining feature to improve performance, in our experiments, this feature was disabled. The performance results are shown in Figure 6.7.

As can be seen, the maximum throughput of Redis was about 160K operations per second when a Unix socket was used. When TCP connections were used, the throughput was about 90K operations with a local connection and about 80K with a remote connection. The read

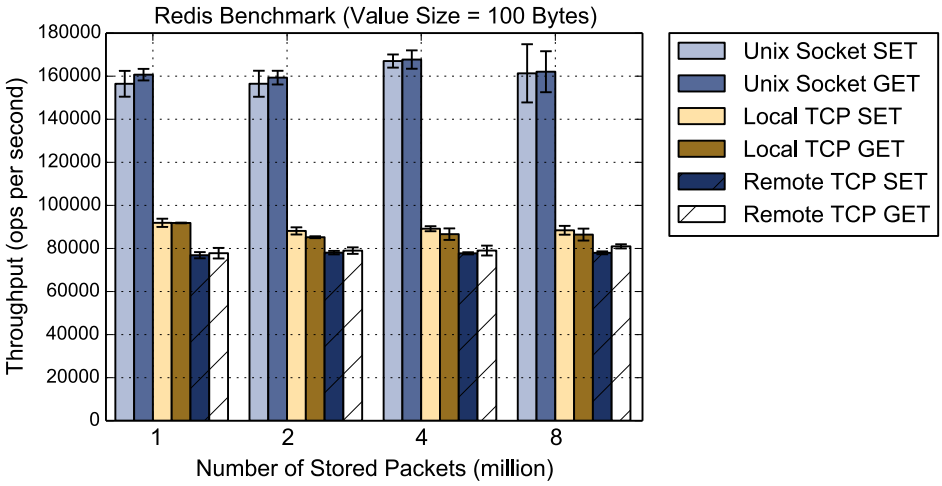


Figure 6.7: Redis Benchmark

and write throughputs were comparable for all of the cases. Obviously, NoSQL key-value stores such as Redis provide much higher throughput. As a result, we need more efficient Repo-ng and NFD implementations that leverage multi-threading, event-driven programming, and non-blocking I/O operations to improve performance.

Future Directions

The Redis-based Repo-ng stores Data packets in memory; thus, its capacity is limited by the memory available on a machine or on a cluster of machines. The available memory can possibly satisfy the requirements of specific applications, but is unlikely to meet the needs of large-scale content distribution. In the latter case, disk storage must be employed. Fortunately, existing IP content distribution nodes are already capable of supporting 100+ TB of content at 10+Gbps throughput. For example, Table 6.1 lists the characteristics of the OpenConnect content distribution node provided by Netflix [48].

Table 6.1: Netflix OpenConnect System Specification

	Rev. A	Rev. C	Rev. D
Operational Throughput	7 Gbps	9 / 12 Gbps	13-17 Gbps
Storage Capacity	100 TB	120 / 160 TB	14 TB (SSD)
Rack Space	4U	4U	1U

As can be seen, these disk storage capacities are significantly larger than what we have experimented with. We expect future NDN Repo implementations can apply the techniques used in existing content distribution nodes. In addition, high-performance key-value stores [41, 42] can also be employed.

6.3 Summary

This chapter reviewed the design issues for in-network caching elements, including both the Content Store and the Repo. We discussed the requirements of the Content Store and presented data structure designs that support LRU and LFU cache-replacement policies. As a persistent content storage, the NDN Repo needs to be scalable. We showed the limitations of the existing Repo-ng implementation and demonstrated that the Redis-based Repo-ng outperformed the original implementation. In addition, the measured Redis benchmark shows the potential for much better performance once the Repo-ng and NFD are improved. As a result, future research efforts should focus on a scalable Repo-ng implementation that efficiently handles Repo commands and content requests. Once these front-end throughput problems are solved, we can use scalable web technologies like Redis to more easily implement large-scale NDN Repos.

Our work in this chapter represents only a first step in exploring scalable in-network caching elements design. Because the CS and IP caching proxies are similar, as are the Repo and content distribution nodes, the NDN in-network caching elements can take advantage of more efficient implementations of caching proxies, key-value stores, and file systems.

Chapter 7

Conclusion

Recent years have witnessed the explosive growth of Internet services. However, the original design goals of the Internet Protocol (IP) do not align well with how the Internet is currently being used. Named Data Networking (NDN) is a clean-slate network architecture that has been proposed to address the shortcomings of IP networks and leverage their advantages. As a core building block of the NDN architecture, name-based packet forwarding is believed to be more difficult than in IP. This dissertation focuses on data structure and algorithm designs for scalable NDN forwarding. Because the performance of packet forwarding on general-purpose multicore platforms has been improved considerably due to advances in both processor architectures and memory technologies, we implement the proposed designs in software and demonstrate their performance.

Forwarding Information Base

The FIB requires longest name prefix lookup with a larger number of name prefixes than in IP. We have proposed a reliably scalable name prefix lookup design based on the binary search of hash tables. By leveraging large pages, memory prefetch instructions, and the

DPDK packet I/O framework, we have demonstrated 10 Gbps FIB forwarding throughput on a general-purpose multicore platform with 256-byte packets and one billion longest name prefix rules, each containing up to seven name components.

We have also explored new characteristics in real-world name prefix datasets. First, we note that a large number of prefixes have many suffixes, and therefore, we have proposed level pulling, a general method to improve the average hash-based longest name prefix matching performance by reducing the number of required hash lookups. Second, in collaborative work, we have proposed the speculative forwarding method, in which the core routers relax the string matching requirements, and thus the memory requirements are reduced significantly for datasets that contain mostly name prefixes with only one name component. In this dissertation, we have proposed fingerprint-based methods to further improve name-based forwarding throughput for these datasets. The proposed fingerprint-based Patricia trie reduces the average depth of the trie, and thus effectively decreases the number of pipeline nodes in hardware-based designs. We have proposed a fingerprint-based hash table design that stores only fingerprints, and requires only 3.2 GB to store 1 billion names, with a lookup latency of 0.29 μs in the single-threaded implementation.

Pending Interest Table

The Pending Interest Table (PIT) keeps track of the forwarded Interest packets and stores the forwarding information for the replied Data packets. The PIT requires fast updates. Our fingerprint-only PIT design takes the approach that by reducing the memory requirements, high-speed memory devices, such as SRAM, can be employed.

The fingerprint-only PIT design relaxes the Interest aggregation feature in the core routers, and therefore a PIT entry needs to be retained longer if there is a fingerprint collision. We have studied the memory requirements and the introduced additional network traffic. When there are multiple core routers, the memory requirement is doubled because remote fingerprint collisions also cause PIT entries to be retained longer. The benefits are diminished, but the memory requirements are still less than the one required by a standard hash table-based design.

In-network Caching Elements

The Content Store and Repo are both in-network caching elements. We have discussed the Content Store design and demonstrated that existing key-value stores, such as Redis, can be employed as the backend storage of the Repo. We also show that Redis-based Repo-ng outperformed the original Repo-ng, although the performance of the existing Repo-ng and NFD require more optimization.

7.1 Future Research Directions

We present the following four research directions to further improve scalable NDN forwarding.

7.1.1 A Full-fledged Forwarding Engine

In this dissertation, we studied the design of the FIB, PIT, and CS individually. A complete NDN forwarding engine requires efficiently integrating these components.

For software-based implementations, the key to achieving higher performance is leveraging both pipelining and parallelism. As in [65], multiple processes were used to leverage pipeline and parallelism. In our FIB implementation, pipelining and thread-level parallelism were also used. The full-fledged forwarding engine can be implemented by extending our existing FIB lookup engine. Multiple worker threads, each residing on a dedicated processor core, process packets in parallel. The PIT and CS support only exact matching and are distributed to each worker thread, thus locking is not required. The FIB, which is read-heavy, is shared among the worker threads. The entire packet processing procedure is divided into three stages. First, packets are received by I/O threads, and then they are distributed to worker threads based on the hash values of their full names. After the lookup, packets are sent to the I/O threads for transmission. This three-stage packet forwarding pipeline has also been applied in our FIB performance study, as shown in Figure 3.13.

As pointed out by [65], secure hash functions are desired for the PIT and CS. Secure hash values can be computed either in a dedicated I/O thread or using dedicated hardware cryptography units. The computed hash values can be sent to the worker threads using ring buffers together with the original packets. Regarding the FIB, the same secure hash function employed in the PIT and CS can be used. It is also possible to use a non-secure but more efficient hash function for the FIB because it is read-heavy.

Another question deserves more exploration is whether the PIT and CS can be combined as one structure. The CS is a cache for Data packets, and the PIT essentially is a cache for Interest packets, so they do share some common characteristics. Combining these two structures together reduces the number of lookups [65] for each packet. The research challenge arises from the fact that the update frequency for the PIT and the CS can be different depending on CS cache-replacement policies, and thus may affect the hash table performance.

Overall, based on our experience with designing each individual component, the measured longest name prefix lookup performance of the FIB with real network packets, and the forwarding performance reported in [65, 54], we believe 10 Gbps can be achieved in software with large forwarding datasets.

7.1.2 Forwarding Information Base

We have implemented the proposed designs only on general-purpose multicore platforms. Hardware-based and hardware-assisted implementations are expected to improve the lookup performance. For instance, hardware-based hash functions or dedicated hash units can offload the CPU-intensive hash computation functions. Finer-grained control over memory accesses and deeper pipeline stages can also improve the performance.

Our FIB designs are based on the assumption that FIB updates are relatively less frequent than FIB lookups, and that FIB updates are managed by the control plane based on the Routing Information Base. When FIB updates become very frequent, different and more efficient solutions that better support FIB updates can be explored.

When the name prefix strings are stored in the FIB, the required memory size is large. Although the server machine we used in the experiment already supports up to 768 GB of memory, it is always beneficial to have compact FIB representations. Compact FIB implementations not only reduce the cost, but also enable duplicating data structures across NUMA nodes.

Lastly, we believe more efficient FIB designs can be engineered based on the specific characteristics of the datasets. When large-scale NDN forwarding rules become available, more options can be explored to exploit their characteristics.

7.1.3 Pending Interest Table

Our approach to designing a scalable PIT focuses on reducing the memory requirements and supports only exact matching to leverage parallelism. An alternative approach, a centralized PIT shared by all the threads, can be explored. The centralized PIT requires locking to avoid race conditions. But a centralized PIT supports the all-prefix lookup scheme, which was proposed in the original NDN design.

Another interesting aspect is that the PIT can be viewed as a cache for Interest packets, thus its implementation can also be similar to the Content Store. Specifically, if full packet names are recorded in the PIT, the entire Interest packet buffer can be retained in the PIT instead of copying the packet names from packet buffers. The Interest packet buffer is released when the corresponding PIT entry is removed.

7.1.4 In-network Caching Elements

A thorough performance evaluation of the Content Store would be helpful in more fully understanding its performance. In addition, there may be new and interesting problems in buffer management, such as whether Interest buffers and Data buffers can be designed differently.

Our study demonstrated that Repo-ng, as a framework, supports integrating other backend storage types. However, we also discovered inefficiency in the current Repo-ng implementation. The research challenge is designing a Repo-ng interface that efficiently handles NDN Repo commands and translates NDN packets to database operations.

References

[1] Alexander Afanasyev et al. NFD Developer's Guide. Technical Report NDN-0021, NDN Technical Report, 2014.

[2] Alexa. `http://www.alexa.com/topsites/`.

[3] Guido Appenzeller, Isaac Keslassy, and Nick McKeown. Sizing Router Buffers. In *Proceedings of the 2004 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '04, pages 281–292, New York, NY, USA, 2004. ACM.

[4] Katerina Argyraki et al. Can Software Routers Scale? In *Proceedings of the ACM Workshop on Programmable Routers for Extensible Services of Tomorrow*, PRESTO '08, pages 21–26, New York, NY, USA, 2008. ACM.

[5] Hirochika Asai and Yasuhiro Ohara. Poptrie: A Compressed Trie with Population Count for Fast and Scalable Software IP Routing Table Lookup. volume 45, pages 57–70, New York, NY, USA, August 2015. ACM.

[6] 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.

[7] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. An Improved Construction for Counting Bloom Filters. In *Proceedings of the 14th Conference on Annual European Symposium - Volume 14*, ESA'06, pages 684–695, London, UK, UK, 2006. Springer-Verlag.

[8] Flavio Bonomi, Michael Mitzenmacher, Rina Panigrahy, Sushil Singh, and George Varghese. Bloom Filters via d-left Hashing and Dynamic Bit Reassignment. In *Proceedings of the Allerton Conference on Communication, Control and Computing*, 2006.

[9] L. Breslau, Pei Cao, Li Fan, G. Phillips, and S. Shenker. Web Caching and Zipf-like Distributions: Evidence and Implications. In *INFOCOM '99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 1, pages 126–134 vol.1, Mar 1999.

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

[11] Jeff Burke, Alex Horn, and Alessandro Marianantoni. Authenticated Lighting Control Using Named Data Networking. Technical Report NDN-0011, NDN Technical Report, 2012.

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

[13] H. Jonathan Chao. Next Generation Routers. In *Proceedings of the IEEE*, pages 1518–1558, 2002.

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

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

[16] Cisco Carrier Routing System. `http://www.cisco.com/c/en/us/products/routers/carrier-routing-system/index.html`.

[17] Huichen Dai, Bin Liu, Yan Chen, and Yi Wang. On Pending Interest Table in Named Data Networking. In *Proceedings of the Eighth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '12, pages 211–222, New York, NY, USA, 2012. ACM.

[18] Mostafa Dehghan, Bo Jiang, Ali Dabirmoghaddam, and Don Towsley. On the Analysis of Caches with Pending Interest Tables. In *Proceedings of the 2nd International Conference on Information-centric Networking*, ICN '15, 2015.

[19] Frank K. H. A. Dehne, Jrg-Rdiger Sack, Nicola Santoro, and Sue Whitesides, editors. *Algorithms and Data Structures, Third Workshop, WADS 93, Montral, Canada, August 11-13, 1993, Proceedings*, volume 709 of *Lecture Notes in Computer Science*. Springer, 1993.

[20] Sarang Dharmapurikar, Sailesh Kumar, John W. Lockwood, and Patrick Crowley. Optimizing Memory Bandwidth of a Multi-Channel Packet Buffer. In *GLOBECOM*, page 6. IEEE, 2005.

[21] Dmoz. `http://www.dmoz.org/`.

[22] 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`.

[23] Bin Fan, Dave G. Andersen, Michael Kaminsky, and Michael D. Mitzenmacher. Cuckoo Filter: Practically Better Than Bloom. In *Proceedings of the 10th ACM International on Conference on Emerging Networking Experiments and Technologies*, CoNEXT '14, pages 75–88, New York, NY, USA, 2014. ACM.

[24] Seyed Kaveh Fayazbakhsh, Yin Lin, Amin Tootoonchian, Ali Ghodsi, Teemu Koponen, Bruce Maggs, K.C. Ng, Vyas Sekar, and Scott Shenker. Less Pain, Most of the Gain: Incrementally Deployable ICN. In *Proceedings of the ACM SIGCOMM 2013 Conference on SIGCOMM*, SIGCOMM '13, pages 147–158, New York, NY, USA, 2013. ACM.

[25] Ali Ghodsi, Teemu Koponen, Jarno Rajahalme, Pasi Sarolahti, and Scott Shenker. Naming in Content-Oriented Architectures. In *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking*, ICN '11, pages 1–6, New York, NY, USA, 2011. ACM.

[26] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. PacketShader: A GPU-accelerated Software Router. volume 40, pages 195–206, New York, NY, USA, August 2010. ACM.

[27] Ultra-high-definition Television. `https://en.wikipedia.org/wiki/Ultra-high-definition_television/`.

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

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

[30] IRCache. `http://www.ircache.net/`.

[31] Sundar Iyer, Ramana Rao Kompella, and Nick McKeown. Designing Packet Buffers for Router Linecards. *IEEE/ACM Trans. Netw.*, 16(3):705–717, June 2008.

[32] Van Jacobson et al. Networking Named Content. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 1–12, New York, NY, USA, 2009. ACM.

[33] Petri Jokela et al. LIPSIN: Line Speed Publish/Subscribe Inter-networking. In *Proceedings of the ACM SIGCOMM 2009 Conference on Data Communication*, SIGCOMM '09, pages 195–206, New York, NY, USA, 2009. ACM.

[34] 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.

[35] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2nd ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[36] Teemu Koponen, Mohit Chawla, Byung-Gon Chun, Andrey Ermolinskiy, Kye Hyun Kim, Scott Shenker, and Ion Stoica. A Data-oriented (and Beyond) Network Architecture. In *Proceedings of the 2007 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '07, pages 181–192, New York, NY, USA, 2007. ACM.

[37] Derek Kulinski and Jeff Burke. NDNVideo: Random-access Live and Pre-recorded Streaming using NDN. Technical Report NDN-0007, NDN Technical Report, 2012.

[38] Sailesh Kumar and Patrick Crowley. Segmented Hash: An Efficient Hash Table Implementation for High Performance Networking Subsystems. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems*, ANCS '05, pages 91–103, New York, NY, USA, 2005. ACM.

[39] Sailesh Kumar, Patrick Crowley, and Jonathan. Turner. Design of Randomized Multichannel Packet Storage for High Performance Routers. In *High Performance Interconnects, 2005. Proceedings. 13th Symposium on*, pages 100–106, Aug 2005.

[40] Sailesh Kumar, Jon Turner, and Patrick Crowley. Peacock Hashing: Deterministic and Updatable Hashing for High Performance Networking. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages –, April 2008.

[41] Sheng Li et al. Architecting to Achieve a Billion Requests Per Second Throughput on a Single Key-value Store Server Platform. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 476–488, New York, NY, USA, 2015. ACM.

[42] Hyeontaek Lim, Dongsu Han, David G. Andersen, and Michael Kaminsky. MICA: A Holistic Approach to Fast In-Memory Key-Value Storage. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 429–444, Seattle, WA, April 2014. USENIX Association.

[43] Rodrigo Mansilha et al. Hierarchical Content Stores in High-speed ICN Routers: Emulation and Prototype Implementation. In *Proceedings of the 2nd International Conference on Information-centric Networking*, ICN '15, 2015.

[44] B. Scott Michel, Konstantinos Nikoloudakis, Peter Reiher, and Lixia Zhang. URL Forwarding and Compression in Adaptive Web Caching. In *INFOCOM 2000. Nineteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 2, pages 670–678 vol.2, 2000.

[45] N. Sertac Artan and H. Jonathan Chao. TriBiCa: Trie Bitmap Content Analyzer for High-Speed Network Intrusion Detection. In *26th Annual IEEE Conference on Computer Communications (INFOCOM 2007)*, pages 125–133, 2007.

[46] NDN Testbed. http://ndnmap.arl.wustl.edu/.

[47] NDN Repository Protocols. http://redmine.named-data.net/projects/repo-ng/wiki.

[48] OpenConnect Deployment Guide. http://oc.nflxvideo.net/docs/OpenConnect-Deployment-Guide.pdf.

[49] Rajesh Nishtala et al. Scaling Memcache at Facebook. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 385–398, Lombard, IL, 2013. USENIX.

[50] ntop. PF˙Ring ZC. http://www.ntop.org/products/packet-capture/pf_ring/.

[51] R. Olsson and S. Nilsson. TRASH A dynamic LC-trie and hash data structure. In *High Performance Switching and Routing, 2007. HPSR '07. Workshop on*, pages 1–6, May 2007.

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

[53] Diego Perino and Matteo Varvello. A Reality Check for Content Centric Networking. In *Proceedings of the ACM SIGCOMM Workshop on Information-centric Networking*, ICN '11, pages 44–49, New York, NY, USA, 2011. ACM.

[54] Diego Perino, Matteo Varvello, Leonardo Linguaglossa, Rafael Laufer, and Roger Boislaigue. Caesar: A Content Router for High-speed Forwarding on Content Names. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 137–148, New York, NY, USA, 2014. ACM.

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

[56] Stefan Podlipnig and Laszlo Böszörmenyi. A Survey of Web Cache Replacement Strategies. *ACM Comput. Surv.*, 35(4):374–398, December 2003.

[57] Lucian Popa, Ali Ghodsi, and Ion Stoica. HTTP As the Narrow Waist of the Future Internet. In *Proceedings of the 9th ACM SIGCOMM Workshop on Hot Topics in Networks*, Hotnets-IX, pages 6:1–6:6, New York, NY, USA, 2010. ACM.

[58] Sriram Ramabhadran, Sylvia Ratnasamy, Joseph M. Hellerstein, and Scott Shenker. Brief Announcement: Prefix Hash Tree. In *Proceedings of the Twenty-third Annual ACM Symposium on Principles of Distributed Computing*, PODC '04, pages 368–368, New York, NY, USA, 2004. ACM.

[59] Redis. https://www.redis.io.

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

[61] Luigi Rizzo. netmap: A Novel Framework for Fast Packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*, pages 101–112, Bellevue, WA, August 2012. USENIX Association.

[62] Devavrat Shah and Pankaj Gupta. Fast Updating Algorithms for TCAMs. *IEEE Micro*, 21(1):36–47, January 2001.

[63] SipHash. `https://131002.net/siphash/`.

[64] Won So, Taejoong Chung, Haowei Yuan, David Oran, and Mark Stapp. Toward Terabyte-scale Caching with SSD in a Named Data Networking Router. In *Proceedings of the Tenth ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '14, pages 241–242, New York, NY, USA, 2014. ACM.

[65] Won So, 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*, ANCS '13, pages 215–226, Piscataway, NJ, USA, 2013. IEEE Press.

[66] Haoyu Song, Sarang Dharmapurikar, Jonathan Turner, and John Lockwood. Fast Hash Table Lookup Using Extended Bloom Filter: An Aid to Network Processing. In *Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, SIGCOMM '05, pages 181–192, New York, NY, USA, 2005. ACM.

[67] Tian Song, Haowei Yuan, Patrick Crowley, and Beichuan Zhang. Scalable Name-Based Packet Forwarding: From Millions to Billions. In *Proceedings of the 2nd International Conference on Information-centric Networking*, ICN '15, 2015.

[68] SQLite. `https://www.sqlite.org`.

[69] The Squid Project. `http://www.squid-cache.org`.

[70] URL Blacklist. `http://urlblacklist.com/`.

[71] Zartash Afzal Uzmi et al. SMALTA: Practical and Near-optimal FIB Aggregation. In *Proceedings of the Seventh COnference on Emerging Networking EXperiments and Technologies*, CoNEXT '11, pages 29:1–29:12, New York, NY, USA, 2011. ACM.

[72] Matteo Varvello, Diego Perino, and Leonardo Linguaglossa. On the Design and Implementation of a Wire-Speed Pending Interest Table. In *Computer Communications Workshops (INFOCOM WKSHPS), 2013 IEEE Conference on*, pages 369–374, April 2013.

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

[74] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High Speed IP Routing Lookups. In *Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, SIGCOMM '97, pages 25–36, New York, NY, USA, 1997. ACM.

[75] Marcel Waldvogel, George Varghese, Jon Turner, and Bernhard Plattner. Scalable High-speed Prefix Matching. *ACM Trans. Comput. Syst.*, 19(4):440–482, November 2001.

[76] Lijing Wang, Ilya Moiseenko, and Lixia Zhang. NDNlive and NDNtube: Live and Prerecorded Video Streaming over NDN. Technical Report NDN-0031, NDN Technical Report, 2015.

[77] Yi Wang et al. Scalable Name Lookup in NDN Using Effective Name Component Encoding. In *Proceedings of the 2012 IEEE 32Nd International Conference on Distributed Computing Systems*, ICDCS '12, pages 688–697, Washington, DC, USA, 2012. IEEE Computer Society.

[78] Yi Wang et al. NameFilter: Achieving Fast Name Lookup with Low Memory Cost via Applying Two-Stage Bloom Flters. In *INFOCOM, 2013 Proceedings IEEE*, pages 95–99, April 2013.

[79] Yi Wang et al. Wire Speed Name Lookup: A GPU-based Approach. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*, pages 199–212, Lombard, IL, 2013. USENIX.

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

[81] Charlie Wiseman et al. A Remotely Accessible Network Processor-based Router for Network Experimentation. In *Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '08, pages 20–29, New York, NY, USA, 2008. ACM.

[82] G. Xylomenos, C.N. Ververidis, V.A. Siris, N. Fotiou, C. Tsilopoulos, X. Vasilakos, K.V. Katsaros, and G.C. Polyzos. A Survey of Information-Centric Networking Research. *Communications Surveys Tutorials, IEEE*, 16(2):1024–1049, Second 2014.

[83] Tong Yang, Gaogang Xie, YanBiao Li, Qiaobin Fu, Alex X. Liu, Qi Li, and Laurent Mathy. Guarantee IP Lookup Performance with FIB Explosion. *SIGCOMM Comput. Commun. Rev.*, 44(4):39–50, August 2014.

[84] Wei You, B. Mathieu, P. Truong, J. Peltier, and G. Simon. DiPIT: A Distributed Bloom-Filter Based PIT Table for CCN Nodes. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–7, July 2012.

[85] Wei You, B. Mathieu, P. Truong, J. Peltier, and G. Simon. Realistic Storage of Pending Requests in Content-Centric Network Routers. In *Communications in China (ICCC), 2012 1st IEEE International Conference on*, pages 120–125, Aug 2012.

[86] Minlan Yu, Alex Fabrikant, and Jennifer Rexford. BUFFALO: Bloom Filter Forwarding Architecture for Large Organizations. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, pages 313–324, New York, NY, USA, 2009. ACM.

[87] Haowei Yuan and Patrick Crowley. Scalable Pending Interest Table Design: From Principles to Practice. In *Proc. of the 33rd Annual IEEE Conference on Computer Communications (INFOCOM'14)*, 2014.

[88] Haowei Yuan and Patrick Crowley. Reliably Scalable Name Prefix Lookup. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, ANCS '15, pages 111–121, 2015.

[89] Haowei Yuan, Tian Song, and Patrick Crowley. Scalable NDN Forwarding: Concepts, Issues and Principles. In *Computer Communications and Networks (ICCCN), 2012 21st International Conference on*, pages 1–9, July 2012.

[90] Lixia Zhang et al. Named Data Networking (NDN) Project. Technical Report NDN-0001, NDN, 2010.

[91] Dong Zhou, Bin Fan, Hyeontaek Lim, Michael Kaminsky, and David G. Andersen. Scalable, High Performance Ethernet Forwarding with CuckooSwitch. In *Proceedings of the Ninth ACM Conference on Emerging Networking Experiments and Technologies*, CoNEXT '13, pages 97–108, New York, NY, USA, 2013. ACM.

[92] N. Zilberman, Y. Audzevich, G.A. Covington, and A.W. Moore. NetFPGA SUME: Toward 100 Gbps as Research Commodity. *Micro, IEEE*, 34(5):32–41, Sept 2014.