Title	Router Architecture and Caching Mechanisms toRealize Information-Centric Networking
Author(s)	大岡, 睦
Citation	大阪大学, 2017, 博士論文
Version Type	VoR
URL	https://doi.org/10.18910/67171
rights	
Note	

The University of Osaka Institutional Knowledge Archive : OUKA

https://ir.library.osaka-u.ac.jp/

The University of Osaka

Router Architecture and Caching Mechanisms to Realize Information-Centric Networking

Atsushi Ooka

July 2017

List of publication

Journal papers

1. Atsushi Ooka, Shingo Ata, Kazunari Inoue, and Masayuki Murata, "High-speed design of conflict-less name lookup and efficient selective cache on CCN router," *IEICE Transactions on Communications*, vol. E98-B, no. 04, pp. 607–620, April 2015.

2. Atsushi Ooka, Suyong Eum, Shingo Ata, and Masayuki Murata, "Scalable Cache Component in ICN Adaptable to Various Network Traffic Access Patterns," *to appear in IEICE Transactions on Communications*, January 2018.

Refereed Conference Papers

1. Atsushi Ooka, Shingo Ata, Toshio Koide, Hideyuki Shimonishi, and Masayuki Murata, "OpenFlow-based Content-Centric Networking architecture and router implementation," in *Proceedings of Future Network and MobileSummit 2013 Conference*, Lisboa, pp. 1–10, July 2013.

2. Atsushi Ooka, Shingo Ata, Kazunari Inoue, and Masayuki Murata, "Design of a high-speed content-centric-networking router using content addressable memory," in *Proceedings of IEEE INFOCOM 2014 Workshop on Name-Oriented Mobility*, Toronto, pp. 1–6, April 2014.

Non-Refereed Technical Papers

1. Atsushi Ooka, Shingo Ata, Toshio Koide, Hideyuki Shimonishi, and Masayuki Murata, "A deployment of Content-Centric Networking by using OpenFlow networks," *IEICE Technical Report (IN2013-19)*, vol. 113, no. 36, pp. 43–48, May 2013.

2. Atsushi Ooka, Shingo Ata, Kazunari Inoue, and Masayuki Murata, "Hardware design and evaluation of a high-speed ccn router using cam," *IEICE Technical Report (IN2013-161)*, vol. 113, no. 473, pp. 105–110, March 2014.

3. Atsushi Ooka, Shingo Ata, and Masayuki Murata, "A proposal and evaluation of cache replacement policy for the implementation of ICN router," *Technical Committee on Information-Centric Networking (ICN)*, pp. 1–10, July 2015.

4. Atsushi Ooka, Suyong Eum, Shingo Ata and Masayuki Murata, "A proposal and evaluation of feasible cache replacement policy for ICN based on CLOCK-Pro," *Technical Committee on Information-Centric Networking (ICN)*, pp. 1–14, December 2016.

Preface

The Internet is used in a form and on a scale considerably different form the network envisaged at the time that the original design principles and assumptions were decided. The initial Internet aims to provide communication service within pairs of hosts. At present, however, the Internet is mainly used for the purpose of distributing and retrieving vast amounts of content such as HTML documents, images, and high-definition video. This mismatch between the host-centric design and the information-centric usage has given rise to many problems such as a large amount of redundant traffic, the excessive bandwidth costs, the inefficient network resource usage, and the lack of the content availability.

To overcome the limitations of the current Internet architecture, Information-centric networking (ICN) has been proposed. ICN is a network-layer architecture designed from scratch; that is, ICN ignores to be compatible with existing network technologies, including IP. ICN shifts the routing behavior based on from "where" to "what". The most significant feature in ICN is that a "name" is used for communication instead of an IP address. A name is assigned to each piece of content "what" although an IP address is assigned to each endpoint "where". The shift of the network concept realized by a name enables a network to support in-network caching, aggregating requests, multicasting, and built-in security for data.

Obviously, we must resolve many challenges to realize this new network architecture: ICN. First, there is a compatibility issue in the deployment of ICN devices in IP networks. Second, the comprehensive implementation of an ICN router hardware is necessary. Of course, the hardware must be sufficiently powerful to realize ICN communications. In the architecture of Content-Centric Networking (CCN), which is one of the ICN architectures, a router must handle two types of packets

and introduce three data structures: forwarding information base (FIB), pending interest table (PIT), and content store (CS) to realize the information-centric features. One of major challenges is the realization of such a high-performance router.

In this thesis, we elaborate our contribution to the realization of an ICN router. Although a number of research projects have been carried out with different ICN realizations, we focus on CCN due to its reliable operation and good documentation. We address the following three problems: the deployment of ICN, the hardware architecture of an ICN router including a high-speed forwarding mechanism, and the cache replacement mechanism that is low-overhead and suitable for ICN traffic.

First, to address the deployment issues, we focus on OpenFlow, which is a promising candidate to provide a programmable environment without disrupting existing networks. Detail implementations of CCN using OpenFlow been sufficiently investigated, although several conceptual designs were previously introduced. We present the detailed design and implementation of an OpenFlow-based CCN with the primary aim that achieves forwarding and end-to-end communication. Our approach can solve the issues in existing designs by using a map from content names to hierarchically structured hash values. We also discuss the advantage of retaining significant attributes of CCN and OpenFlow such as high-speed forwarding and network slicing which promotes the deployment of CCN and research for routing, caching, and security strategies.

Then, we provide a concrete hardware design for a CCN router. The router uses three basic tables (FIB, PIT, CS) and incorporates two entities that we propose. One of these entities is an name lookup entity, which looks up a name within a few cycles from content-addressable memory by the use of a Bloom filter; the other is an interest count entity, which counts interest packets that require certain content and selects content worth caching. Our proposed design achieves high throughput and low latency, and demonstrates feasible performance and cost on the basis of a concrete hardware design using distributed content-addressable memory.

Finally, we propose two low-overhead cache replacement algorithms that are suitable for network traffic. Although cache replacement algorithms have been intensively researched in the context of web-caching and content delivery networks previously, the conventional approaches cannot be directly applied to ICN due to its poor performance, the computational and memory cost, or the unsuitability for unknown ICN traffic. Our proposed algorithms achieve a high cache-hit rate,

and are scalable enough to address the challenge of implementing a cache mechanism in resource-restricted hardware. First, we propose Compact CLOCK with Adaptive Replacement (Compact CAR). Compact CAR has the mechanism that discards one-time requested content and identifies the content worth caching. The numerical simulation shows that Compact CAR can reduce the consumption of cache memory to one-tenth compared to conventional approaches.

Although Compact CAR can realize the low-overhead content store, the cost of a lookup table increases because we designed it under the conditions without the constraints of the memory cost of a lookup table. In addition, although Compact CAR is tolerant to the one-time requested content, it is not tolerant enough to access to smaller data units called chunks. Thus, we propose CLOCK-Pro Using Switching Hash-tables (CUSH), which reduces not only the cost of content store but also the cost of a lookup table. CUSH is also tolerant to access to chunks. The numerical simulation shows that CUSH can achieve cache hits against the traffic traces that simple conventional algorithms hardly cause any hits.

Acknowledgments

This thesis could not have been accomplished without the tremendous assistance of many people, and I would like to acknowledge all of them.

First and foremost, I have to express my sincere gratitude to my supervisor, Professor Masayuki Murata of Graduate School of Information Science and Technology, Osaka University, for the continuous support of my Ph.D study and communication, and for kind patience, immense knowledge, and insightful comments.

I am heartily grateful to the members of my thesis committee, Professor Takashi Watanabe, Professor Toru Hasegawa, and Professor Teruo Higashino of Graduate School of Information Science and Technology, Osaka University, and Professor Morito Matsuoka of Cyber Media Center, Osaka University, for their multilateral reviews and perceptive comments.

Furthermore, I would like to show my great thanks to Professor Shingo Ata of Graduate School of Engineering, Osaka City University, and Assistant Professor Suyong Eum, of Graduate School of Information Science and Technology, Osaka University. I am deeply indebted to them for sharing expertise, valuable guidance and encouragement extended to me.

I would also like to show my appreciation to Dr. Kazunari Inoue of poco-apoco Networks Company Limited, and Dr. Hideyuki Shimonishi and Dr. Toshio Koide of NEC Corporation, for their fruitful discussion and a great experience with them.

Additionally, I must acknowledge Associate Professor Shin'ichi Arakawa, Assistant Professor Yuichi Ohsita, Associate Professor Go Hasegawa, Assistant Professor Daichi Kominami, Assistant Professor Yuki Koizumi of Osaka University, for their productive comments and helps.

I take this opportunity to express gratitude to all of past and present colleagues, friends, and

secretaries of the Advanced Network Architecture Research Laboratory, Graduate School of Information Science and Technology, Osaka University.

Finally, I conclude my acknowledgment with expressing my thanks to my parents and family. Thank you for giving me invaluable supports throughout my life.

Contents

List of publication **i**

Preface **iii**

Acknowledgments **vii**

1 Introduction **1**

1.1 Background 1

1.2 Outline of Thesis 4

2 OpenFlow-based CCN Architecture and Router Implementation **9**

2.1 Introduction 9

2.2 Background 11

2.2.1 Content-Centric Networking 11

2.2.2 OpenFlow 13

2.3 Architectural Description 14

2.3.1 Design Principles 14

2.3.2 Specification of Packets 14

2.3.3 Matching and Forwarding 15

2.3.4 Caching 17

2.3.5 Description of Communication 17

2.4 Implementations of Each Component 19

2.4.1 End-User . . . 19
2.4.2 Configuration of Flow Entries in the OpenFlow Switch . . . 19
2.4.3 OpenFlow Controller . . . 20
2.5 Discussion . . . 22
2.5.1 Comparison with Existing Solutions . . . 22
2.5.2 Contribution to Deployment Issues . . . 23
2.5.3 Consideration of OpenFlow 1.3 Specifications . . . 23
2.6 Summary . . . 24

3 High-speed Design of Conflict-less Name Lookup on CCN Router 25
3.1 Introduction . . . 26
3.1.1 Background . . . 26
3.1.2 Related Work . . . 27
3.1.3 Objectives . . . 28
3.2 CCN . . . 29
3.2.1 Principles of CCN . . . 29
3.2.2 Name Lookup Algorithm . . . 31
3.3 Architecture . . . 38
3.3.1 Name Lookup Entity . . . 39
3.3.2 Interest Count Entity . . . 42
3.4 Hardware Design . . . 44
3.4.1 Name Lookup Entity . . . 44
3.4.2 Interest Count Entity . . . 45
3.4.3 Summary . . . 46
3.5 Evaluation . . . 48
3.5.1 Memory Size and Cost . . . 49
3.5.2 Throughput . . . 52
3.6 Summary . . . 54

4 Compact CAR: Low-overhead cache replacement policy for an ICN router 57

4.1 Introduction . . . 57

4.2 Related works . . . 59

4.3 Design Considerations of Cache Replacement Algorithm for ICN . . . 60

4.3.1 Access Patterns of Traffic in the Network . . . 60

4.3.2 Computational Power and Memory Limitations . . . 61

4.3.3 Adaptable Parameter Tuning . . . 62

4.4 Compact CLOCK with Adaptive Replacement (Compact CAR) . . . 63

4.4.1 Data Structure of Compact CAR . . . 63

4.4.2 Operation of Compact CAR . . . 64

4.4.3 Design of Low-overhead Variable-sized CLOCK List . . . 66

4.4.4 Replacement Algorithm of Compact CAR . . . 69

4.5 Performance Evaluation . . . 70

4.5.1 Simulation Setup and Configuration . . . 72

4.5.2 Cache Hit Rate with a SCAN Access Pattern using Synthetic Traffic . . . 74

4.5.3 Cache Hit Rate with a SCAN Access Pattern using Real Traffic Trace . . . 77

4.5.4 Simulation with the Line Topology . . . 79

4.5.5 Dynamic Parameter Tuning . . . 81

4.5.6 Analysis on Space and Time Complexities of CAR and Compact CAR . . . 83

4.5.7 Time and Space Complexity of the Remaining Policies . . . 86

4.6 Discussion on the Implementation of Compact CAR for High Performance ICN Core Router . . . 91

4.6.1 Feasibility of Hardware Implementation . . . 91

4.6.2 Computational Overhead of Variants of CLOCK . . . 92

4.7 Summary . . . 92

5 Scalable Cache Component in ICN Adaptable to Various Network Traffic Access Patterns 95

5.1 Introduction . . . 95

5.2 Related Works 96
5.3 Design Considerations of Cache Replacement Algorithm for ICN 99
5.3.1 Access Patterns of Traffic in the Network 99
5.3.2 Computational and Memory Cost 100
5.3.3 Overhead on Lookup Table caused by Ghost Caches 101
5.4 CLOCK-Pro Using Switching Hash-table (CUSH) 103
5.4.1 Data Structure of CUSH 103
5.4.2 Operation of CUSH 105
5.4.3 Collision-free Lookup Table vs Hash-tables Accepting Collision 108
5.4.4 Management of Hash-tables to deal with LOOP 108
5.5 Performance Evaluation 109
5.5.1 Evaluation of Property Resistant to Access patterns using Synthetic Traffic 110
5.5.2 Evaluation of the Influence of Ghost Caches to LOOP-resistant Property . . 112
5.5.3 Cache Hit Rate with Real Traffic Trace 114
5.5.4 Analysis on Space and Time Complexities 116
5.6 Summary 119

6 Conclusion **121**

Bibliography **125**

List of Figures

2.1 UDP payloads representing Interest packets (left) and Data packets (right) 15
2.2 Detailed description of communication: (1) Sending an Interest packet 18
2.3 Detailed description of communication: (2) Receiving an Interest packet 18
2.4 Detailed description of communication: (3) Sending and receiving a Data packet . 18
2.5 Flowchart of the Packet-In event (main process) 21
2.6 Flowchart of the Packet-In event (processing Interest) 21
2.7 Flowchart of the Packet-In event (processing Data) 21
2.8 Probability of a hash collision ($B = 4$) 24
2.9 Probability of a hash collision ($B = 16$) 24

3.1 Example of Using EM (upper: $Case(1)$, lower: $Case(2)$) 35
3.2 Example of Using BPM (upper: $Case(1)$, lower: $Case(2)$) 35
3.3 CCN Router Architecture 40
3.4 Definition of CAM Entry 41
3.5 Example of CAM and RAM Entries in NLE 41
3.6 Hardware Design of NLE 44
3.7 Hardware Design of ICE 46
3.8 Packet Processing in the Proposed CCN Router 47
3.9 The Size of SRAM Required for NLE 51

4.1 Data Structure of Compact CAR 63

4.2 Illustration of Computational and Memory Costs in the Inserting Operation in the Different Data Structures 66
4.3 Example of Moving a Chunk a_i from T_1 to T_2 by Replacing the Edge Chunk a_n . . 67
4.4 Popularity Distribution of Real Trace 75
4.5 Results for Synthetic Traffic in Units of Content 76
4.6 Results for Synthetic Traffic in Units of Chunks 77
4.7 Results for Real Traffic Trace 78
4.8 CDF of RD in Various Workloads 79
4.9 Results for Simulation with the Line Topology 80
4.10 Comparison between Non-cooperative Caching and Ideally-cooperative Caching . 81
4.11 Dynamics of Hit Rate of CFR(q) and Adaptive Parameter q 82
4.12 Description of calculating ω of CLOCK 88
4.13 Space Complexities of CAR and Our Proposal(Compact CAR) 91

5.1 Ideal Lookup Table for the Cache Replacement Policy using Ghost Caches (e.g., CLOCK-Pro) 102
5.2 Data Structure of CUSH 103
5.3 The Operation of CUSH in the Case where e_x Becomes Hot 105
5.4 The Operation of CUSH in the Case where e_x Becomes Cold 105
5.5 Variations of Lookup Table for Ghost Caches 107
5.6 Evaluation for SCAN-resistant Property with Synthetic Workloads in Units of Content 110
5.7 Evaluation for LOOP-resistant Property with Synthetic Workloads in Units of Chunks 110
5.8 Effect of Implementation of Ghost Cache on LOOP-Resistant Property 113
5.9 Effect of the Amount of Ghost Caches on LOOP-Resistant Property (Using Synthetic Trace in Units of Chunks with$\alpha = 1.2$ and $L = 1.5\mathrm{KB}$) 113
5.10 Popularity Distribution of a Real Trace 114
5.11 Results for Real Traces 115
5.12 Space Complexities of CLOCK, CLOCK-Pro, and Our Proposal When $n = 10^7$ . . 116

List of Tables

2.1 Implementation of ICN architectures 10
2.2 State transition rules and operations on flow entries 20

3.1 Summary of Matching and Selecting Algorithms 38
3.2 Name Lookup Algorithm Applied to Our Implementation 39
3.3 Performance of Lookup Mechanisms (Number of Entries: 10 Million.) 49
3.4 Number of Read/Write Accesses in the Name Lookup Process, by Memory Type . 52

4.1 Number of Chunks Per Second $[\mathrm{pck/s}]$ 74
4.2 Statistics of Workloads in Units of Chunks 75
4.3 Time Complexity of Cache Replacement Algorithm's Overhead 84
4.4 Space Complexity of Cache Replacement Algorithm's Overhead 84

5.1 Cache Policies Used to Analyze Effect of Ghost Cache 112
5.2 Number of Chunks Per Second$[\mathrm{pck/sec}]$ 114
5.3 Statistics of Workloads in Units of Chunks 115
5.4 Space Complexity of Cache Replacement Algorithm's Overhead 116
5.5 Average hand movement count of CLOCK-based policies 117

Chapter 1

Introduction

1.1 Background

The Internet has been known for half a century since its inception, and has now grown into a global network of networks. The Internet, however, is used in a form and on a scale considerably different from the network envisaged at the time that the original design principles and assumptions were decided. The initial Internet aims to provide communication service within pairs of hosts. At present, however, the Internet is mainly used for the purpose of distributing and retrieving vast amounts of content such as HTML documents, images, and high-definition video.

The mismatch between the design and the usage has given rise to many problems. At the inception, the Internet has only the minimum functionality required for a network layer due to strict technical limitations. Because the Internet was assumed to be responsible only for the communication between pairs, it is equipped with an IP address to identify the endpoints and the mechanisms to realize the communication between two endpoints. On the other hand, the current network is used to distribute/retrieve content. The functionality of the Internet is not designed for the change of such uses.

To overcome the limitations of the current Internet architecture, Information-centric networking (ICN) has been proposed. ICN is a network-layer architecture designed from scratch; that is, ICN ignores to be compatible with existing network technologies, including IP. ICN shifts the routing

behavior based on from “where” to “what”. The most significant feature in ICN is that a “name” is used for communication instead of an IP address. A name is assigned to each piece of content “what” although an IP address is assigned to each endpoint “where”. The shift of the network concept realized by a name enables a network to relax both the spatial and temporal constraints on communication in the current Internet, which would significantly improve the flexibility of the placement of contents, efficiency of network resource usage, and the quality of experience (e.g., content availability and response time).

For example, consider how the Internet and ICN differ in the access to popular content. First, the Internet realizes them as massively redundant communications between a content provider and consumers. Because each communication is handled separately, the aggregation point of the accesses in a network becomes a bottleneck: degrading the network performance. A large amount of redundant traffic carrying the identical content also increases the load of the provider and reduces the availability. In order to provide popular content, current content providers use different services such as Content Delivery Networking (CDN) or Peer to Peer (P2P) networking, which are expensive and application-specific. These technologies, which work at endpoints, are difficult to adequately utilize the resources in a network. Although current IP networks barely manage to function to support content dissemination services, a new network technology to replace these makeshift solutions is desired because of the emergence of state-of-the-art technologies such as high-definition (4K/8K) video streaming and IoT, which are estimated to increase network traffic explosively.

On the other hand, ICN natively supports aggregating requests, multicasting, and in-network caching. Although a number of research projects have studied ICN approaches, here we focus on Content-centric networking (CCN) [1], which is one of major ICN realizations. Communication via CCN is realized through the exchange of two types of packets: *Interest* packets and *Data* packets (abbreviated to Interest and Data). An Interest packet is sent by a data consumer and contains the name of content that is being requested. A Data packet is sent back by a data producer in response to the Interest packet from the consumer and contains the actual data. Communication begins when an Interest packet constructed by the data consumer is sent, and ends when the consumer is satisfied by Data packets sent by the original data producer or some caches in the network.

To implement the forwarding functions in CCN including request aggregation, multicasting,

caching, and a loop-free architecture, the CCN router uses three data structures: the forwarding information base (FIB), pending interest table (PIT), and content store (CS). The FIB is a table used for determining a proper interface for forwarding Interest packets. The PIT works as the "bread crumb", that is, the PIT remembers the interfaces from which Interest packets arrived in order to send back the corresponding Data packets. Interest packets that have duplicate names (i.e., that have already been recorded in the PIT) at the PIT are dropped. In this way, the PIT realizes request aggregation, multicasting and a loop-free architecture, and resolves the bottleneck at the aggregation point. The CS serves as an in-network cache for Data packets. Since Data packets with the same content are assigned an identical name, cached Data packets can be reused.

A number of research projects have studied ICN approaches [2], such as CCN [1], Named-Data Networking (NDN) [3], PURSUIT [4], and Network of Information (NetInf) researched in SAIL [5]. These all share the common concept of addressing the content: communication by a "name", which is assigned to each chunk of data. They also try to natively implement functions such as in-network caching, multicasting, and built-in security for data. In this thesis, we focus on CCN/NDN, which has a useful features and have achieved a dominant position. We will use ICN and CCN/NDN interchangeably.

We aim to demonstrate the feasibility of ICN to support the current usage of network. Obviously, many challenges must be resolved to realize ICN, which is a "clean-slate" network (i.e., it starts with a blank sheet rather than an incremental improvement), such as security, mobility, and ICN deployment, among others [3, 6, 7]. First, it is unclear how to deploy the equipment for ICN because ICN is incompatible with IP [8, 9, 10]. It is impossible to make all current IP routers support ICN in a day. Second, the comprehensive implementation of an ICN router hardware is necessary. Of course, an ICN router must support the content dissemination service sufficiently. An overlay network that consists of software routers cannot achieve the performance to serve the purpose. Therefore, we needs the feasible hardware architecture of an ICN router. The hardware must be sufficiently powerful to realize ICN communications, where the router requires more advanced mechanisms than IP routers such as in-network caching and packet forwarding based on a name address, which is hierarchically-structured and variable-length. In addition, there are various problems such as a naming scheme, a scalable routing protocol based on a name address, and a

security.

Most previous studies have focused on isolated components or techniques at the router level, such as name lookup mechanisms [11, 12, 13, 14, 15, 16] and caching [17, 18, 19, 20, 21, 22, 19]. However, the designs and simulations of ICN routers shown in the existing studies are not based on hardware designs or implementations. The existing studies on in-network caching use only blunt cache replacement policies unsuitable for network traffic, or propose statistical approaches that are too expensive to be employed in an ICN core router due to computational and memory costs.

In this thesis, we elaborate the realization of an ICN router. We address the following three problems: the deployment of ICN, the hardware architecture of an ICN router including a high-speed forwarding mechanism, and the cache replacement mechanism that is low-overhead and suitable for ICN traffic. Furthermore, we investigate cache replacement mechanisms. Although there are interesting challenges in the implementation of an actual ICN router such as routing and security, we address these as major difficult challenges of the ICN implementation. We also aim to demonstrate the feasibility of ICN, whose realization has been considered as impossible.

1.2 Outline of Thesis

OpenFlow-based CCN Router [23, 24]

In Chapter 2, we address the deployment issues of ICN. Because ICN is incompatible with IP [8, 9, 10], It is impossible to make all current IP routers support ICN in a day. ICN-over-IP network employing software ICN routes can be easily deployed; however, the overlay network cannot exploit the features of ICN such as the flexibility of the placement of contents and the efficiency of network resource usage. Thus, we focus on OpenFlow, which is a promising candidate to provide a programmable environment without disrupting existing networks. OpenFlow implements the software-defined networking (SDN) concept [25]. This allows researchers to test novel networking protocols like ICN in an actual network environment [9]. The possibility of deployment supported and promoted by OpenFlow has also been investigated [8, 10].

While the benefits of designing a network using OpenFlow based on information-centric principles have been actively discussed, there has been a lack of discussion on the major characteristics of CCN/NDN and a lack of concrete implementations in the early stages of research on ICN. Most existing research works differ from the CCN/NDN concept in the details of the implementation, for instance some features are not supported such as aggregation scaling routing states and availability of uniquely named content. There has not been much discussion on how to implement the flow tables and OpenFlow controller although they consider the locations and abstract roles of OpenFlow controllers, switches, and caching entities.

In order to deal with these issues, we present the detailed design and implementation of an OpenFlow network in which ICN packet forwarding is realized based on hierarchically-structured hash values of a name. A hierarchically-structured name helps an ICN router to decide a Data packet to be sent immediately after receiving a corresponding Interest packet, because the name carried in the Interest packet and in the corresponding Data packet may be different. This prompt decision is difficult in existing solutions which give them different hash values that are not hierarchically structured. We employ Trema as the OpenFlow framework for implementation and testing.

High-speed ICN router [26, 27, 28, 29]

All ICN architectures share the common concept of assigning a name, which helps a network to natively support in-network caching, multicasting, built-in security for data and others. Because the name is variable-length and longer than an IP address, we need a more enhanced router that can handle the complex name. Thus, in Chapter 3, we focus on one of major challenges: implementation of an ICN router. To handle explosively increasing network traffic, a hardware ICN router is required. It is also essential to demonstrate the feasibility and the specific performance of a ICN router.

Most previous studies have focused on isolated components or techniques at the router level, such as caching and name lookup mechanisms. For example, Caesar [11] aims to implement a scalable high-speed forwarding table. DiPIT [12] and NameFilter [13] focus on pending interest tables (PITs) and propose a very fast inexpensive architecture consisting of two-level Bloom filters, but

the probabilistic nature of that model means that false positives can never be completely eliminated. NCE [14], ENPT [15] and ATA(MATA) [16] adapt highly memory-efficient name lookup mechanisms by using a trie-like structure. MATA achieves line speed (i.e., near-real-time performance) by means of a highly parallelized architecture using graphics processing units, although it is difficult to reduce the latency.

Among the existing complete router designs [30, 31], content-addressable memory (CAM), which has the potential to become a major lookup technology, has not been sufficiently researched because of its high price. Nevertheless, the estimation concludes that it would be impracticably difficult to support an Internet-scale ICN deployment using CAM, although the analysis indicates that at the content-delivery network or Internet service provider scale, it could be easily afforded and would make a significant contribution to investigating the feasibility of ICN/CCN. In addition, the designs and simulations of ICN routers shown in the existing studies are not based on hardware designs or implementations. We plan to eventually demonstrate a router design and hardware architecture with the same level of concreteness shown in [32]. In [32], the implementation of reconfigurable match tables for software-defined networking (SDN) is proposed and detailed design and evaluation are described for a hardware implementation. The proposed techniques for fast matching a number of entries in SDN could help to implement feasible matching mechanisms for ICN/CCN, but the techniques that would be useful in SDN cannot be directly applied to ICN router because of a variable-length name supported by CCN.

Of course, the ICN router hardware must be sufficiently powerful to realize ICN communications. The realistic performance of a hardware router is required to estimate performance at the network level and evaluate whether various proposals for ICN are reasonable. First, we study the available algorithms for lookup in detail. Then, we propose the router architecture customized by the name lookup entity (NLE) and the interest count entity (ICE). NLE, which consists of many small CAMs and DLB-BF, realizes high-throughput and reasonable costs. ICE can assist in adaptive caching. Finally, we evaluate its throughput, size of memory, and cost based on our hardware design.

Low-overhead Cache Replacement Policy Adaptable to Various Traffic Patterns [33, 34, 35, 36]

ICN is built on the idea of name-based routing which enables each ICN router to be aware of users' requests as well as their counterpart responses. Thus, individual ICN routers can be turned into caching devices by simply providing physical cache memory for them. This feature of ICN that all network devices have caching capability is called in-network caching function, which helps to reduce network access latency, alleviate network traffic, balance network load, and achieve robustness against a single failure scenario. Broadly speaking, the popularity of Internet traffic approximately follows a Zipf distribution, where a large amount of packets carrying redundant data are biasely transferred. In [37], content requested more than once occupies 50% or more of the volume of network traffic, with the value depending on the observation period. The potential of caching is explored in terms of which content should be cached at the router and how to choose a "victim" chunk. The former and the latter are known as content placement and cache replacement problems, respectively.

The cache replacement mechanism used in ICN requires low computational and memory overhead because the computing and storage resources of an ICN router are very limited. However, most previous studies on cache replacement policies in ICN have adopted cache replacement policies that are too blunt to achieve significant performance improvement, such as first-in first-out (FIFO), random, and least-recently-used (LRU), or are so complex and costly (often using statistical information), such as least frequently used (LFU), that they are impractical. Although there are a number of research studies on the policies from the viewpoint of a computer system (e.g., CPU, I/O buffer, and virtual memory) that have proposed scalable cache replacement policies, it is not obvious that the knowledge learned from them can be applied to in-network caching since the studies cover access patterns of computer applications only, rather than those of network traffic.

We propose two low-overhead cache replacement algorithms that are suitable for network traffic. The algorithms achieve a high cache-hit rate, and are scalable enough to address the challenge of implementing a cache mechanism in resource-restricted hardware. First, we propose Compact

CLOCK with Adaptive Replacement (Compact CAR) inspired by CAR [38] in Chapter 4. Compact CAR has the mechanism that discards one-time requested content and distinguishes the content worth caching. The numerical simulation shows that Compact CAR can reduce the consumption of cache memory to one-tenth compared to conventional approaches.

Although Compact CAR can realize the low-overhead content store, the cost of a lookup table increases because we design it under the conditions without considering the memory cost of a lookup table. In addition, although Compact CAR is tolerant to the one-time requested content, it is not tolerant enough to the accesses in the unit of chunks. Thus, we propose CLOCK-Pro Using Switching Hash-tables (CUSH) inspired by CLOCK-Pro [39] in Chapter 5. CUSH reduces not only the cost of content store but also the cost of a lookup table. CUSH is also tolerant to the accesses in the unit of chunks. The numerical simulation shows that CUSH can achieve cache hits against the traffic traces that simple conventional algorithms hardly cause any hits.

Chapter 2

OpenFlow-based CCN Architecture and Router Implementation

2.1 Introduction

The Internet, which is now a global network of networks, is used in a form and scale considerably different from the original design principles and assumptions, and this gives rise to many problems. Information-Centric Networking (ICN) and Content-Centric Networking (CCN) [1], which provide a location-independent network architecture in which the content is named, has been proposed as a measure for coping with these problems facing the Internet; the goal is for CCN to become the Internet of the future.

A number of research projects have studied ICN/CCN approaches such as Named Data Networking (NDN) [3], PURSUIT [4], and Network of Information (NetInf) researched in SAIL [5]. These all share the common concept of addressing the content that is exchanged in communication by "name", which is assigned to each chunk of data, not the location of a host. They also try to natively implement functions such as in-network caching, multicasting, and built-in security for data.

We summarize the differences between them in Table 2.1. CCN and NDN shares a design principle. In CCN/NDN, the communication is initiated by sending a request for content. The name in

Table 2.1: Implementation of ICN architectures

	Communication	Naming	Name Resolution
CCN/NDN	Request-driven	Hierarchically-structured and human-readable	Each router
PURSUIT	Publish-subscribe	Flat and unreadable	Rendezvous node
NetInf	Request-driven	Flat and unreadable	Name resolution system

the request is hierarchically-structured and human-readable. The name is resolved at each router and forwarded to a content publisher. PURSUIT adopts the publish-subscribe communication model. For security purposes, a name is hashed in PURSUIT; therefore, a name is flat (not hierarchy) and unreadable. PURSUIT introduces rendezvous nodes to determine the path to delivery data based on its name. NetInf adopts the request-driven communication model and a flat name. In NetInf, Name Resolution System resolves the name of content and determine a delivery path. In this thesis, we focus on CCN/NDN because their hierarchically structured name and distributed name resolution enable flexible content retrieval.

The features of CCN create key challenges for implementation. There are feasibility problems for achieving routing, caching, and security and deployment problems about a gradual approach to ICN/CCN deployment, costs, and business models, and they have been analyzed and evaluated [30, 31, 7]. It is necessary to demonstrate that CCN could become the successor to the Internet and to encourage general network users to migrate to the CCN.

Taking this situation into account, OpenFlow is expected to aid with the development of CCN. OpenFlow implements the software-defined networking (SDN) concept [25], which provides for programmable networks by separating the rules for handling packets (called the control-plane) from the rules for forwarding in hardware (called the data plane). This allows researchers to test novel networking protocols like CCN in an actual network environment [9]. The possibility of deployment supported and promoted by OpenFlow has also been investigated [8, 10].

While the benefits of designing a network using OpenFlow based on content-centric principles have been actively discussed, there has been a lack of discussion on the major characteristics of CCN/NDN and a lack of concrete implementations. Most existing research differs from the

CCN/NDN concept in the details of the implementation, with some features such as aggregation scaling routing states and availability of uniquely named content sacrificed. There has not been either much discussion on how to implement the flow tables and OpenFlow controller.

In order to deal with these issues, we present the detailed design and implementation of an OpenFlow network in which packet forwarding is realized based on hierarchically structured hash values of names. We employ Trema as the OpenFlow framework for implementation and testing.

2.2 Background

2.2.1 Content-Centric Networking

CCN is a novel network architecture that was designed with a focus not on the location of data but on the content of data. This approach has the following advantages:

- Content-centric rather than host-centric communication
- *Names* that provide each chunk of data with unique, human-readable, and hierarchical structured addresses
- Mechanisms for native multicasting, in-network caching, and security that is built into the data

The content-centric design was inspired by recent developments in the utilization of the Internet. A main use of current networks is the distribution and retrieval of vast amounts of data such as HTML documents, images, and high-definition video. However, specifying the locations of providers and consumers is not necessary for this purpose. CCN is a solution for dealing with the incompatibilities and security concerns by shifting the routing behavior based on from “where” to “what”.

In CCN, it is not necessary to know where the device we want to communicate with is located, and instead we directly identify the name of data that we want. Names enable us to do this by providing each chunk of data with a unique, human-readable, and hierarchical address. They allow those who use networks and develop network applications to eliminate the complexity of identifying hosts

and to directly specify the identifier of the content. For example, a picture of an apple produced by XYZ could be named "`/XYZ/pictures/apple.jpg`." The name could also contain the version and segmentation of data. For example "`/ XYZ /pictures/apple.jpg/v1/s2`" could indicate the second chunk of version 1 of the image. In addition, longest prefix match (LPM) look-up on names is implemented, which enables to aggregate routing state just like Internet.

Communication via CCN is realized through the exchange of *Interest* packets and *Data* packets (abbreviated to Interest and Data). Interest packets are sent by data consumers and contain the request for and the name of the content. Data packets are sent back by the data producer in response to Interest packets received by the producer and contain the actual data. Communication begins when a consumer sends an Interest packet constructed by the data consumer. Then, the communication ends when the consumer is satisfied with the Data packets sent by the original data producer or some cache in the network.

To implement forwarding functions in a CCN including multicasting, caching, and a loop-free architecture, the CCN router contains three data structures: the forwarding information base (FIB); pending interest table (PIT); and content store (CS). The FIB is a table used for determining the proper interface for sending Interest packets that have arrived at the router. The PIT remembers the interfaces from which Interest packets arrived in order to send back the matching Data packets which are subsequently received by the router. Interest packets that have duplicate names (i.e., that have already been recorded in the PIT) leave only the trace of the route, with the forwarding skipped in order to realize multicasting and a loop-free architecture. The CS serves as a cache for Data packets. Since Data packets are addressed using identical names, cached Data packets can be reused independently of the requester and time. This is implemented by longest prefix match (LPM) look-up on names, with a single index table implemented and interconnected for the actual retrieval.

CCN supports inherent security and privacy protection. Every packet is verified using signatures. There is demand for security that is embedded in the content data itself, and research is ongoing into how to implement these inclusive structures. Because of this and our intention to implement forwarding independently from other functions, we do not take account of the security.

2.2.2 OpenFlow

OpenFlow was implemented as the first standard interface for an SDN architecture. SDN provides network operators with programmable management which is decoupled from the underlying network infrastructure for current L2/L3 switches. This programmability removes the barriers to experimenting with new network protocols in a real network infrastructure.

The separation of data paths from network intelligence results in two basic components called the switch and controller, respectively. An OpenFlow switch is a programmable physical switch that processes and forwards packets. An OpenFlow controller is the entity that configures all the switches in the network in order to realize the desired protocols or packet routing strategy. This centralized system allows operators, enterprises, vendors, and end users to run experiments for evaluation and deployment.

The operational instructions which the controller gives to the switches are written to a flow table contained in the switch as individual flow entries. The flow entries mainly consist of matching rules and actions. The controller gives the switches the operational instructions, which are written to a flow table contained in the switch as individual flow entries. When the switch receives a packet, it searches the flow table for corresponding entries by using the matching rules, and then processes the packets according to the actions in the entries. The variety of possible combinations of flow entries create the potential for numerous possible networks.

Note that the first version and the latest version of the OpenFlow switch specifications differ in terms of the matching rules and actions. The OpenFlow 1.0 specifications, which were the first version, are employed by almost all products. In this version, the matching rules support the port number in the transport layer, the IPv4 address in the network layer, and the MAC address in the link layer. IPv4 LPM is also supported. The OpenFlow 1.0 specifications also support instructions as actions, including modifications to address and port fields and forwarding matching packets to specified physical ports. On the other hand, the newer specifications than OpenFlow 1.3 contain significant advancements with support not only for matching and modification to IPv6 addresses and provider backbone bridging, but also for improvements in matching by using arbitrary bitmasks.

2.3 Architectural Description

2.3.1 Design Principles

In this section, we describe an architecture of OpenFlow-based CCN. This guarantees a high-speed forwarding by using hierarchically structured hashes of names. In order to decide the flow for sending back a Data packet immediately after receiving a corresponding Interest packet, it is necessary to give these names hierarchically structured hash values for matching them by using LPM. Because a name of an Interest packet may be different from one of a Data packet satisfying it, this quick decision is impossible in existing solutions which give them different hash values that are not hierarchically structured.

We explain a detailed design including configuration of the flow entries, which is missing in the existing researches. In accordance with the design principles of CCN, a forwarding strategy is separate from the routing strategy. This promotes the deployment of CCN concurrently with research on routing systems. Owing to our concern about the need of early-deployment, the design is based on UDP and OpenFlow 1.0 specifications, both of which are currently available. UDP, which is so popular protocols for many network applications, makes it easy to develop applications. OpenFlow 1.0 specifications are adopted by most of current equipments and frameworks while OpenFlow 1.3 specifications are rarely adopted. In addition, our solution can be easily expanded by new technologies such as OpenFlow 1.3 specifications.

2.3.2 Specification of Packets

UDP packets are defined specifically for CCN and are treated as Interest packets and Data packets. End-users are able to send and receive these packets natively.

In order to simplify application developments, we propose that a predefined set of fields in the packet headers are examined, and have adopted the following rules:

UDP Port Number Used for distinction between Interest packets and Data packets. It is necessary to assign port numbers that is uniquely used in the network. We denote their port numbers as 50001 and 50002 respectively for purposes of illustration in this chapter.

IPv4 Address Associated with the name of the content through an appropriate hash function (detailed in Subsection 2.3.3).

MAC Address Determines whether the I/O port where the packet has arrived is inbound or outbound. Specially defined MAC address must be assigned to the packets. 11:11:11:11:11:11 indicates unprocessed packets, and 22:22:22:22:22:22 indicates processed packets by OpenFlow switches for purposes of illustration in this chapter.

The CCN packets (i.e., the UDP payloads) that we use are simplified as much as possible and do not support security features such as signatures since our aim is only to demonstrate the feasibility of packet exchange using CCN. The packets therefore contain the minimal information for communicating on CCN, as shown in Figure 2.1.

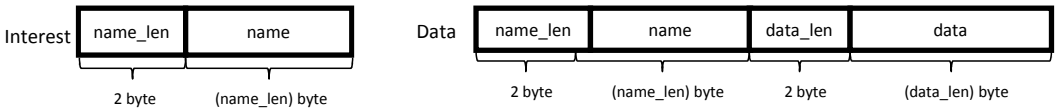


Figure 2.1: UDP payloads representing Interest packets (left) and Data packets (right)

The Interest packets contain the name of the requested content and the length of the name. The Data packets also contain these information as well as the actual data and the length of the data. Since each of the length fields are 2 bytes long, the sizes of the name and data are limited to less than 64 Kbytes. This maximum length is reasonable because it is obvious that the length of the names is less than 65,536, and Data packets where the entire length is more than 1,500 bytes are first split up in order to avoid fragmentation in the lower layers. These UDP packets customized for CCN allow end-users to retrieve content without needing equipment based on OpenFlow.

2.3.3 Matching and Forwarding

In content-centric solutions, only the names of content are examined for forwarding and routing decisions except in the case of security strategies. CCN routers require the ability to perform LPM on names. However, OpenFlow does not offer any matching rules for inspection of the UDP payload. Instead, the IP address field can be subnet masked in the OpenFlow 1.0 specifications, and

we therefore propose a solution that writes hash values that are converted from the name into the IP address field.

Each component of the name is assigned 4 bits, allowing names consisting of up to 8 components to be handled. Each component is converted to a value from 1 to 15, with the value 0 indicating that the corresponding component does not exist. For example, consider the case of content named "`/pict/a.jpg/v1/s2`." The name contains four components: "`pict`", "`a.jpg`", "`v1`", and "`s2`". If these components are converted by a hash function to 1, 3, 4, and 8, respectively, the IP address becomes "19.72.0.0/16" in CIDR notation or "0x13480000" in hexadecimal. When a Data packet with the name "`/pict/a.jpg/v1/s2`" is cached, an Interest packet named "`/pict/a.jpg`", which converts to "19.0.0.0/8", would match the Data packet by using LPM on the IP address. OpenFlow switches performs forwarding based on this virtually implemented LPM by name, so the LPM rule for the IP address is common to all flow entries.

Actions targeting Interest and Data packets must include forwarding of the packet to the appropriate destination I/O port and controller. The destination I/O ports of Interest packets are looked up in the routing table and of Data packets are determined when Interest packets that have the same name pass through the switch. In OpenFlow networks, which is usually composed of multiple switches, the packets are simply transfered from the source host to the destination host depending on the flows.

On sending a packet out to an external host, it is necessary to make the following two modifications to the packet before forwarding: (1) adjust the IP and MAC destination addresses to those of the destination host; and (2) change the IP and MAC source addresses to those of a virtual gateway. Note that (2)'s modifications make it possible to prevent end-hosts from becoming confused in the ARP table by providing end-hosts with non-duplicate IP addresses and the uniquely defined MAC address 11:11:11:11:11:11 . On receiving a packet from an external host, which are assigned MAC destination address of 11:11:11:11:11:11 , this MAC destination address is modified to 22:22:22:22:22:22 for multihop transmission in OpenFlow networks.

In addition, Network slicing can be easily implemented as long as port number is defined appropriately. We can isolate the CCN traffic from the IP traffic and handle them by examining the field of UDP port number, i.e, UDP packets with port number 50001 or 50002 are processed as the

CCN traffic, and others are processed as the IP traffic.

In accordance with the design principles of CCN, this forwarding strategy is separate from the routing strategy. This promotes the deployment of CCN concurrently with research on routing systems.

2.3.4 Caching

CCN routers possess a native cache for Data packets called the CS. However, OpenFlow switches do not have this kind of caching mechanism. We therefore introduce a software cache in the OpenFlow controller. In particular, whenever the OpenFlow controller receives Data packets, the controller stores the packets for later requests.

Although the controller is useful for caching, the software-based implementation lacks the speed of the CS which is implemented in hardware. A dedicated storage device for caching would certainly have the potential for achieving performance comparable with that of the CS. However, we have not adopted this device as it lacks the flexibility required for testing during development.

2.3.5 Description of Communication

Figures 2.2, 2.3, and 2.4 give a detailed description of the communication in our architecture. There are a single CCN router comprising an OpenFlow switch and controller, and two hosts that are exchanging a piece of content named "`/pict/a.jpg/v1`". The host denoted "A" is a consumer who sends Interest packets and receives Data packets, and the host denoted "B" is a content producer who does the opposite.

Figure 2.2 illustrates the beginning of a scenario in which the requester sends an Interest packet named "`/pict/a.jpg`". The requester computes the hash value of the name to determine the IP destination address before sending the packet. The MAC destination address is resolved by the CCN router. The flow for forwarding an Interest packet named "`/pict/a.jpg`" is preliminarily written in a flow table in the switch. In addition to sending the packet, the requester needs to prepare the UDP server for receiving the Data. Figure 2.3 shows the forwarded Interest packet arriving at the producer. Both the IP address and MAC address of the packet are modified properly. What needs

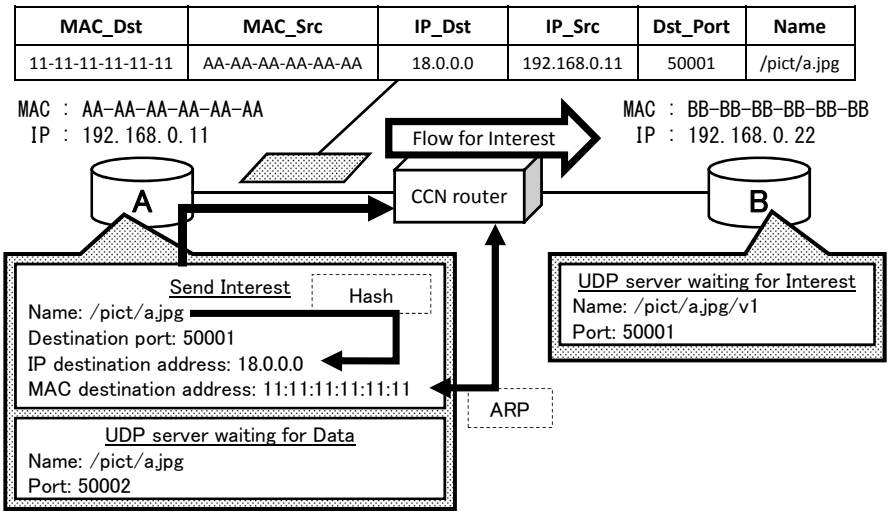


Figure 2.2: Detailed description of communication: (1) Sending an Interest packet

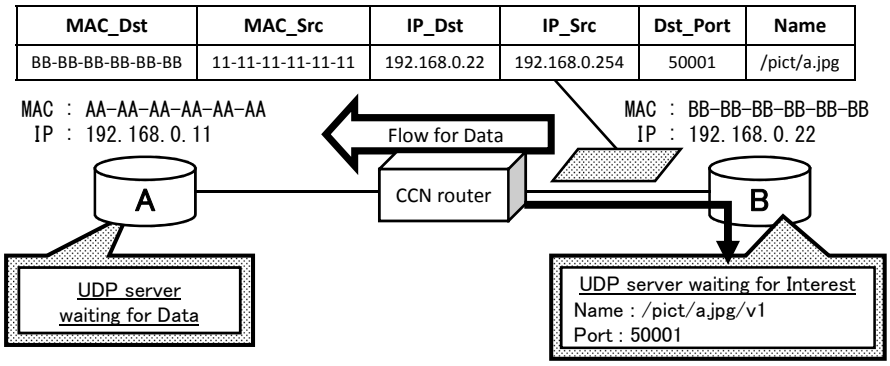


Figure 2.3: Detailed description of communication: (2) Receiving an Interest packet

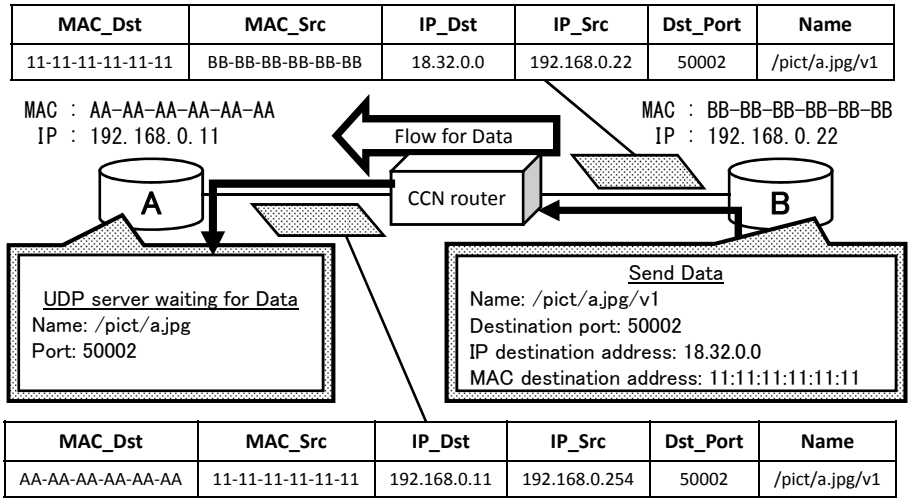


Figure 2.4: Detailed description of communication: (3) Sending and receiving a Data packet

to be emphasized is that the flow is also rewritten for Data packets. Finally, Figure 2.4 shows the flow of the Data packet. In response to the Interest packet, the producer sends a Data packet named "`/pict/a.jpg/v1`". The Data packet is modified and forwarded by the CCN router similarly to the Interest packet. The consumer subsequently receives the packet and the request is satisfied. What is important is that all packets contain names with different hashes, but these do not disrupt the processing owing to the LPM and hierarchically structured hashes.

2.4 Implementations of Each Component

In this section, we describe the implementation of three components: an end-user, an OpenFlow switch, and an OpenFlow controller.

2.4.1 End-User

End-users generate packets according to the specifications shown in Figure 2.1 and set up a socket to wait for Interest and Data packets. Note that Data packets must be split into segments having lengths of less than 1,500 bytes.

2.4.2 Configuration of Flow Entries in the OpenFlow Switch

The flow entries in the OpenFlow switch are managed according to the states computed by the OpenFlow controller. The state assigned to each name can be defined as the data structure (i.e., FIB, PIT, or CS) where the packet with that name should be processed. An index table is used for looking up the state of the name.

Suppose an Interest packet that requests content named "`/pict/a.jpg`" reaches the switch for the first time. In this case, the FIB should be looked up to process this packet since the CS has not yet cached the Data packet for this name and the PIT entries have not been registered yet either. The state for "`/pict/a.jpg`" therefore needs to be pre-defined as FIB, with the controller having already registered the corresponding flow entries in the switch. The flow entries then enable the switch to forward the packet appropriately, and the new state becomes PIT since the unsatisfied Interest packet has been added to the PIT.

The state transition rules and the operations performed on received packets are listed in Table 2.2. Operations to add and remove entries are intended to not only deal with the flow entries in the switch but also to change the state of the controller.

Table 2.2: State transition rules and operations on flow entries

State	On Receiving Interest	New State	On Receiving Data	New State
FIB	Add the PIT entry	PIT	Cache Data and add the FIB entry	CS
PIT	Add PIT entry	PIT	Remove PIT entry, and add FIB entry	CS
CS	Send back cached Data	CS	Do nothing	CS

2.4.3 OpenFlow Controller

In our solution, the OpenFlow controller is programmed to behave as a CCN router in combination with the OpenFlow switch. It also needs to ensure consistency of communication with the end-users. To that end, the controller needs to manage the state of each content name, configure flow entries (as detailed in Subsection 2.4.2), handle ARP packets properly, and associate the I/O port to which the host is directly connected.

As noted in Subsection 2.3.3, the virtual CCN router implemented by the OpenFlow controller and switch handles ARP table confusion. Without ARP, it is impossible for hosts to communicate over Ethernet, and UDP is no exception. However, it is quite likely that the large number of dynamic IP addresses derived from the names would trigger an explosion of entries in the ARP table. As a consequence, the CCN router must provide fixed IP addresses and MAC addresses and act as a virtual gateway router.

Figures 2.5, 2.6 and 2.7, show flowcharts of the Packet-In event processing and summarize a significant part of the implementation of the controller. First, it is obvious that the controller handles two kinds of packets: ARP packets and UDP packets. On receiving an ARP request packet, the controller sends back an ARP reply to notify the MAC address of the virtual gateway.

UDP packets that are received are processed according to the states listed in Table 2.2. Figure

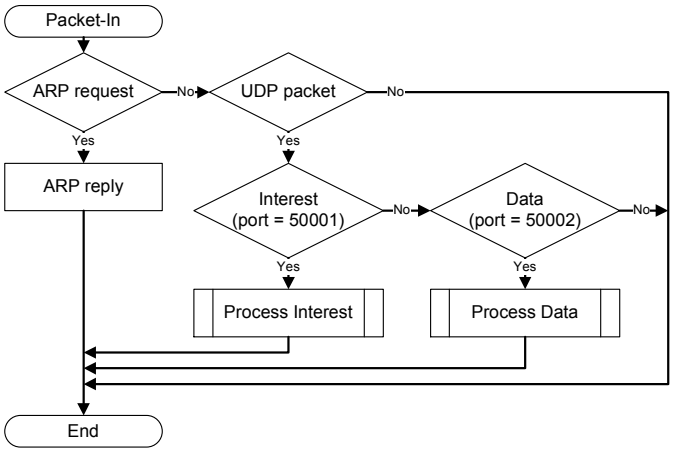


Figure 2.5: Flowchart of the Packet-In event (main process)

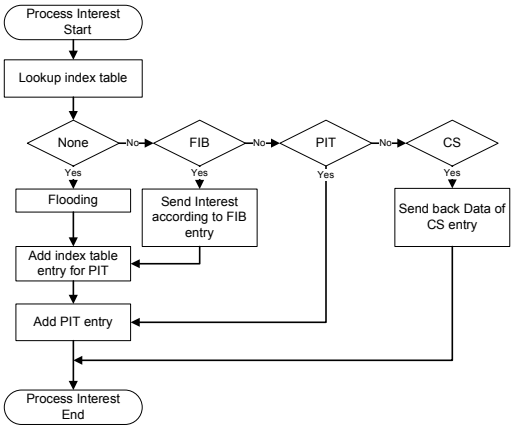


Figure 2.6: Flowchart of the Packet-In event (processing Interest)

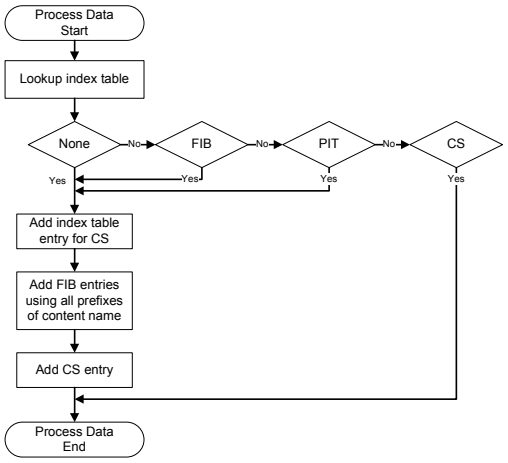


Figure 2.7: Flowchart of the Packet-In event (processing Data)

2.6 shows the process for Interest packets, which have a destination port number of 50001, and Figure 2.7 shows the process for Data packets, which have a destination port number of 50002. After looking up the state corresponding to the name of the packet in the index table, each packet is handled according to the state as described in Subsection 2.4.2.

2.5 Discussion

2.5.1 Comparison with Existing Solutions

Existing solutions implement a dynamic mapping between content names and fixed-length hash values [8, 9]. However, those solutions lack some of the important characteristics of CCN. A hierarchically structured name is required for performing the LPM in CCN in order to allow for different names between the Interest and Data packets.

Consider two packets, an Interest packet with the name "`/pict/a.jpg`" and a Data packet with the name "`/pict/a.jpg/v1/s2`". Obviously, the two names give different hash values, which are not hierarchically structured and have nothing in common in the existing solutions. This makes it impossible to use the LPM between two names from the hash values. It is therefore unreasonable for a CCN router that have received an Interest packet both to register all flow entries for forwarding conceivable Data packets, and to determine the single name of the corresponding Data packet. Because of this, the router cannot create the flow entries before receiving the Data packet.

In our proposed solution, by contrast, the content names are mapped to IPv4 addresses while maintaining the hierarchical structure, which makes it possible to look up content names using the LPM on IPv4 addresses. Since a CCN router can create the flow entries immediately on receiving an Interest packet, a Data packet is sent back quickly along the path of switches. It should also be noted that this also facilitates new routing system research which employs aggregation based on hierarchically structured names.

2.5.2 Contribution to Deployment Issues

The network implemented according to our specifications meets the minimal requirements for CCN with regard to realizing end-to-end communication. Forwarding based on a hierarchically structured name is performed by using the IP address field to store hierarchically hashed names. Routing strategies are easy to implement and expand, including strategies that use the LPM on the content name, owing to their independence from the forwarding strategies.

In addition to this feasibility, CCN implemented using OpenFlow can be deployed in a way that allows for gradual migration from the Internet. While the deployment of OpenFlow networks represents a cost to network operators, isolating experimental networks from the existing network by network slicing potentially reduces the overall burden including the migration.

The transition from OpenFlow-based CCN to pure CCN remains a topic for further research. However, we are not yet concerned with this issue because the specifications of CCN are still being explored.

2.5.3 Consideration of OpenFlow 1.3 Specifications

The OpenFlow 1.3 specifications have the potential to improve the probabilities of hash collisions by supporting matching on IPv6 addresses. This makes it possible to use arbitrary bitmasks for matching IPv6 address fields and MAC addresses. We focus on applying this to the probabilities of hash collisions, i.e., to use the LPM on IPv6 and MAC addresses to reduce the rate of collisions.

Assuming each component of a name is assigned B bits, the probabilities P_B^{protocol} for the three protocols (IPv4, MAC, and IPv6) are given by the following equation:

$$P_B^{\mathrm{protocol}} = \left(1 - \prod_{i=1}^{N} \frac{(2^B - 1) - i}{2^B - 1}\right)^{\frac{len}{B}} \tag{2.1}$$

where N is the number of names and len is the bit-length of the addresses in the protocol. Figures 2.8, 2.9 show the probabilities P_B^{IPv4}, P_B^{MAC} and P_B^{IPv6} for $B = 4$ and 16 respectively.

The graphs show that the parameter B is much more influential on the probability of a hash collision than the difference between protocols. This means that it is better to adopt as large a value

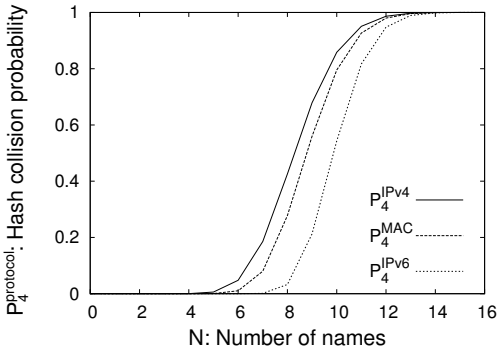


Figure 2.8: Probability of a hash collision ($B = 4$)

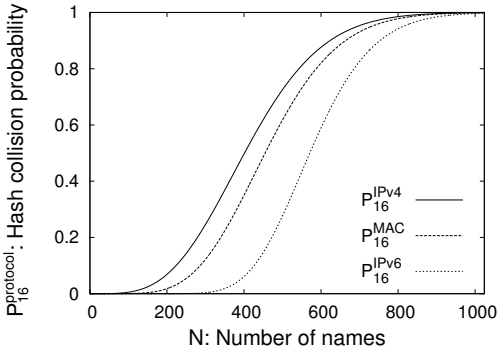


Figure 2.9: Probability of a hash collision ($B = 16$)

of B as possible. However, 99.9% of URLs consist of less than 30 components according to [30]. It is therefore preferable to adopt matching on IPv6 addresses with $B = 4$. The trade-off between the hash collision probability and the number of components is left for investigation in future work.

2.6 Summary

With the growing recognition that the Internet is reaching its limits, CCN has been drawing attention around the world. This promising architecture offers tremendous advances including native support for multicasting, in-network caching, and security that depends on the content. OpenFlow, which is also gaining an increasing presence, is expected to become the intermediary for the deployment of ICN devices.

This chapter presented the architecture and implementation of an OpenFlow-based CCN. Our examination demonstrated the feasibility and possibility for deployment, and the ability to simultaneously study challenges such as routing and caching strategies by using hierarchically structured hashes of content names. The trade-off between hash collision probabilities and the number of components that can be handled in names will be investigated once products and a framework are released.

Chapter 3

High-speed Design of Conflict-less Name Lookup on CCN Router

The ICN architecture requires a router with performance far superior to that offered by today's Internet routers. Although many researchers have proposed various router components, such as caching and name lookup mechanisms, there are few router-level designs incorporating all the necessary components. The design and evaluation of a complete router is the primary contribution of this chapter. We provide a concrete hardware design for a router model that uses three basic tables —forwarding information base (FIB), pending interest table (PIT), and content store (CS)—and incorporates two entities that we propose. One of these entities is the name lookup entity, which looks up a name address within a few cycles from content-addressable memory by use of a Bloom filter; the other is the interest count entity, which counts interest packets that require certain content and selects content worth caching. Our contributions are (1) presenting a proper algorithm for looking up and matching name addresses in CCN communication, (2) proposing a method to process CCN packets in a way that achieves high throughput and very low latency, and (3) demonstrating feasible performance and cost on the basis of a concrete hardware design using distributed content-addressable memory.

3.1 Introduction

3.1.1 Background

The Internet, which is now a global network of networks, is used in a form and on a scale considerably different than the network envisaged at the time that the original design principles and assumptions were decided, and this has given rise to many problems. The initial implementation of the Internet was developed to provide the ability to communicate within pairs of hosts. At present, however, the Internet is used for the purpose of distributing and retrieving various types of content, with this content going to and coming from global networks. This is quite different from communicating with a specific host. Nevertheless, Internet protocol (IP) datagrams require that an IP address be assigned, which specifies the network interface. In addition, there is no guarantee that the location of the requested data will be constant, because the data may be moved or deleted or the server providing that data may become temporarily inaccessible.

Information-centric networking (ICN), also known as content-centric networking (CCN) [1], has been proposed as a measure for overcoming the limitations of the current Internet architecture. The most significant feature in ICN/CCN is that a "name" address, which is variable-length and human-readable in a way similar to uniform resource locators (URLs), is assigned to each piece of content. Using the name address, content distribution applications (e.g., YouTube and Twitter), which have been becoming more and more popular, can be supported efficiently and securely by adhering to the end-to-end principle. The fundamental motivation for introducing ICN/CCN is to realize a content delivery channel that requires only directly specifying the name of the desired content at the network layer. An additional benefit of such a system would be that in-network caching could relax both the spatial and temporal constraints on communications that are present in the current Internet, which would significantly improve the flexibility of the placement of contents, efficiency of network resource usage, and the end-users quality of experience (e.g., content availability and response time). Against this background, ICN/CCN has proved increasingly attractive in recent years, and many researchers are involved in study and development of this area (see [2] and references therein).

Obviously, many challenges must be resolved to realize CCN, which would be a "clean-slate"

network (i.e., a replacement rather than an incremental improvement). First, we need new name resolution and routing mechanisms that are based on the name addresses used in CCN. Second, the "bread crumb" forwarding technique, which uses Interest and Data packets and which naturally incorporates multicast and Interest aggregation into the network, requires lookup tables that can be updated much more quickly than IP address tables. Most research focuses on in-network caching mechanisms because they can cache content more efficiently and thus require fewer resources [40, 31]. In addition, there are a number of problems that have been analyzed and evaluated: security, mobility, and CCN deployment, among others [3, 6, 7].

3.1.2 Related Work

A number of research projects have studied ICN/CCN approaches, such as CCNx [41], NDN [3], PURSUIT [4], and SAIL [5]. These all share the common concept of addressing the content that is to be exchanged in the communication by a "name", which is mnemonic and unique to each chunk of data. They also try to natively implement functions such as in-network caching, multicasting, and built-in security for data. In this chapter, we focus on CCN/NDN, which is characterized by a hierarchical structure and variable-length names.

Most previous studies have focused on isolated components or techniques at the router level, such as caching and name lookup mechanisms. For example, Caesar [11] aims to implement a scalable high-speed forwarding table. DiPIT [12] and NameFilter [13] focus on pending interest tables (PITs) and propose a very fast inexpensive architecture consisting of two-level Bloom filters, but the probabilistic nature of that model means that false positives can never be completely eliminated. NCE [14], ENPT [15] and ATA(MATA) [16] approach highly memory-efficient name lookup mechanisms by using a trie-like structure. MATA achieves line speed (i.e., near-real-time performance) by means of a highly parallelized architecture using graphics processing units, although it is difficult to reduce the latency.

Among the existing complete router designs [30, 31], content-addressable memory (CAM), which has the potential to become a major lookup technology, has not been sufficiently researched because of its cost. Nevertheless, the estimations conclude that it would be impracticably difficult

to support an Internet-scale CCN deployment using CAM, although the analysis indicates that at the content-delivery network or Internet service provider scale, it could be easily afforded and would make a significant contribution to investigating the feasibility of ICN/CCN. In addition, the designs and simulations of CCN routers shown in the existing studies are not based on hardware designs or implementations. We plan to eventually demonstrate a router design and hardware architecture with the same level of concreteness as that in [32]. In [32], the implementation of reconfigurable match tables for software-defined networking (SDN) is proposed and a detailed design and evaluation are described for a hardware implementation. The proposed techniques for quickly matching a number of entries in SDN could help to implement feasible matching mechanisms for ICN/CCN, but the techniques that would be useful in SDN cannot be directly applied to the router for use in CCN because of the variable-length name addresses allowed by CCN.

3.1.3 Objectives

We address one of the biggest challenges to implementing a CCN router to demonstrate the feasibility and specific performance of a CCN router. Of course, the hardware must be sufficiently powerful to realize CCN communications. The realistic performance of a hardware router is required to estimate performance at the network level and evaluate whether various proposals for CCN are reasonable. However, there are few studies offering a comprehensive design for a CCN router; instead, most previous studies have focused on isolated components or techniques at the router level, such as caching and name lookup mechanisms. We investigated a design for a CCN router in [27] and evaluated its performance; however, the discussion about the design did not completely elucidate the available algorithms for lookup. In this chapter, we described such algorithms in detail. Furthermore, we evaluate the scalability of a distributed-and-load-balancing Bloom filter (DLB-BF) [42], which could increase the utility of memory space when implemented in our previous work [27].

In this chapter, we propose a complete CCN router design that can be implemented on existing hardware and show the feasibility and performance of the router. In Section 2, we describe an accurate communication model for CCN that properly handles all packets. In Section 3, customizing

the router architecture by using the name lookup entity (NLE) and the interest count entity (ICE) constructs is proposed, and the hardware design using CAM and a Bloom filter is demonstrated in Section 4. We design a feasible hardware architecture by means of dividing CAM into smaller parts. In Section 5, we comprehensively evaluate the throughput, size of memory, and cost of the CCN router. Finally, we give a conclusion and discuss areas for future research.

3.2 CCN

3.2.1 Principles of CCN

CCN is a novel network architecture that was designed with a focus on the content of data, rather than on the location of that data. This approach has the following advantages.

- Content-centric, rather than host-centric, communication
- *Names* that provide each chunk of data with unique, human-readable, and hierarchical addresses
- Mechanisms to natively support multicasting, in-network caching, and built-in security for data

The content-centric design was inspired by recent developments in the utilization of the Internet. A main use of current networks is the distribution and retrieval of vast amounts of data, such as HTML documents, images, and high-definition video. Specifying the locations of providers and consumers is not necessary for this purpose. However, the protocol for Internet addressing, which is the dominant tool for networked communication, imposes these kinds of unnecessary processes. In addition, data sharing systems that are unaware of network structure and packet content, such as CDNs and peer-to-peer (P2P) applications are costly and inefficient. CCN is a solution for dealing with these incompatibilities and security concerns by shifting the routing behavior from focusing on “where” to focusing on “what”.

The concept of *name*, with each chunk of data assigned a name, plays a major role in CCN as a replacement for the IP addresses that are presently assigned to device interfaces. In CCN, it

is not necessary to know the location of the device that we want to communicate with; instead, we identify the name of the data chunk that we want. Names enable us to do this by providing each chunk of data with a unique, human-readable, and hierarchical address. They allow those who use networks and develop network applications to eliminate the complexity of identifying hosts and to directly specify the identifier of the content. For example, a picture of an apple produced by XYZ could be named "`/XYZ/pictures/apple.jpg`." The name could also contain the version and segment number of data to permit versioning and segmentation. For example "`/XYZ/pictures/apple.jpg/v1/s2`" could indicate the second chunk of version 1 of the image.

Communication Model

CCN's communication model is request-driven through the exchange of *Interest* packets and *Data* packets (abbreviated to Interest and Data below). To begin, a data consumer requests content by sending an Interest, which contain the name of the content. In response to the Interest, the content provider sends Data, which contain the actual data. Finally, the consumer receives all the Data and the request is satisfied.

The name written in an Interest request may be just a prefix of the requested content. For example, when a consumer requests a video named "/video/a.mpg", the producer may send the Data with the name "/video/a.mpg/v1/s1", so that the name contains the version and segment number of the data. This dynamic naming method is referred to as *active naming* in this chapter. In addition, we must consider the case where a name and its prefix (e.g., "/video/a.mpg" and "/video/a.mpg/v1/s1") refer to different content. In this chapter, we call pairs with this relationship *name siblings*. Name siblings complicate the handling of name addresses in a router, but there is no inherent reason to forbid the use of name siblings by applications running on CCN. It is not a foregone conclusion that name siblings will be accepted for CCN communications, but we include name siblings in our discussion here.

Router Behavior

Although a number of research projects approach ICN/CCN in a different way, we adopt the design principles of Named Data Networking (NDN) [3] in this chapter. To implement forwarding functions in a CCN that includes multicasting, caching, and a loop-free architecture, the CCN router contains three data structures: a forwarding information base (FIB), a PIT, and a content store (CS). The FIB is a table used for determining the proper interface for forwarding Interest requests that have arrived at the router. The PIT remembers the interfaces from which these requests have arrived so that it can send back the matching Data that will be subsequently received by the router. Interest requests that specify duplicate names (i.e., that have already been recorded in the PIT) leave only the trace of the route and forwarding is skipped so as to aggregate requests and realize multicasting and a loop-free architecture. The CS serves as a cache for Data. Because identical Data are addressed by identical names, cached Data can be reused independently of the requester and time.

3.2.2 Name Lookup Algorithm

To demonstrate the utility and limitations of advanced functions in CCN, we consider all possibilities for name lookup algorithms that could be used on a CCN router, describing them in this subsection. An algorithm for implementing lookup tables in a CCN router is not trivial, and to the best of our knowledge has never been discussed. The tables contained by the router (i.e., FIB, PIT, and CS) are not simple hash tables with uniquely keyed entries; instead, a single retrieval key could match multiple entries in the table because of prefix matching and active naming. We need to consider how to match entries in the tables and select one of them so that packets are appropriately processed without conflict between the matching policies and implementations. The Interests and Data must be looked up in the FIB, PIT, and CS tables. There are five possible combinations (rather than six, because Data are not looked up in the FIB table).

Matching and Selecting Algorithms

A matching algorithm is an algorithm to decide whether the search key (denoted by K_S) matches the key stored in the table entry (denoted by K_E), and we must consider the case where a given key

matches the prefix of another key K (denoted by $P(K)$). The following four matching algorithms are available:

- *Exact Match (EM)*, which matches when $K_S = K_E$;
- *Search-key Prefix Match (SPM)*, which matches when $P(K_S) = K_E$;
- *Entry-key Prefix Match (EPM)*, which matches when $K_S = P(K_E)$; and
- *Both-keys Prefix Match (BPM)*, which matches when $K_S = K_E$or when one is identical to the prefix of the other (*not* $P(K_S) = P(K_E)$, that is, when both keys have the same prefix).

EM is used for strict matching that disallows any ambiguities, such as for determining the existence of a prefix aggregation. SPM is a familiar matching algorithm that is employed in current IP routers as a longest-prefix-matching (LPM) algorithm, although LPM is not the only possibility for selecting from among distinct Data found by SPM. EPM and BPM have not been used in IP routers; however, these matching algorithms should be taken into account because they potentially allow CCN routers to take advantage of active naming.

The non-exact matching algorithms might retrieve multiple entries, and so algorithms for choosing one of the retrieved entries should be described. Such algorithms are called selecting algorithms. When the matching algorithm is SPM, selecting the longest entry is suitable; this is just the LPM algorithm that is used in conventional IP routers. Selections from results provided by EPM and BPM are more complex. For example, when $K_S$is "/video/A.mpg", the $K_S$will match both "/video/a.mpg/v1/s1" and "/video/a.mpg/v2/s4". Since LPM cannot deterministically select only one of the entries when those entries are same the length, other criteria for use in selecting algorithms are needed. One strategy is to prioritize the time when the entries are registered or the number of requests. We can also adopt a simpler strategy when matching from FIB: select all matching entries. In that case, Interest requests are multicast from all ports corresponding to the matched entries, although this might generate excessive traffic. Richer strategies would enable a CCN router to use known preferences or the information on the publisher, scope, and other attributes (such as in the ChildSelector [1, 43] framework) can be used if the CCN router is powerful enough to employ those strategies.

Algorithms Suitable for Each Table

The combination of SPM and LPM is the most suitable for FIB. As with the strategy for current IP routers, a name address is hierarchically structured so that multiple name addresses having the same prefix can be aggregated into one entry. The other algorithms cannot aggregate entries.

When looking up Data in CS, EM should be used because the name assigned to the Data must not be an abbreviated active name and must, instead, be a complete name that identifies specific content. Additionally, the other algorithms do not support name siblings. For the process to look up Data in CS it is essential to avoid duplicate entries, but it is possible to skip this process when the Data is so unpopular that PIT does not have any matching entries.

When looking up an Interest in CS, either EM or EPM should be used because it would be undesirable for an Interest to match a Data or cache entry with a name shorter than the one specified in the Interest request. Thus, although SPM and BPM, which allow $K_S$to match a shorter $K_E$in CS, are unsuitable, EM and EPM cause no problems. We note that EPM requires that the priority rules select exactly one entry when a single $K_S$matches multiple K_E. If name siblings are allowed, then EM must be used in order to prevent a router from returning undesirable Data in response to the Interest request. Otherwise, EPM could improve the cache hit rate by exploiting an advantage of active naming. This advantage occurs in cases such as when there is an entry K_E^1 = "/video/a.mpg/v1/s1" in CS. When searching for an Interest named K_S^2 = "/video/a.mpg", K_E^1 can match K_S^2 by using EPM.

For looking up Data in PIT, SPM should be chosen: Active naming and name siblings cannot be supported by the other matching algorithms. For the selecting algorithm, we can select both the entry with the longest key and all entries that match the search key. While using LPM is a risk-free approach, it is more efficient to satisfy multiple Interests at once if name siblings are disallowed. As an example, suppose there are the entries K_E^1 = "/video/a.mpg/v1/s1", K_E^2 = "/video/a.mpg/v2/s6" and K_E^3 = "/video/a.mpg" in PIT. When the Data named K_S= "/video/a.mpg/v1/s1" arrived at the router, the Data were able to satisfy not only K_E^1 but also K_E^3 by selecting all matching entries. Needless to say, because an Interest request for content named "/video/a.mpg" will not be satisfied by content named "/video/a.mpg/v1/s1", the all-hits algorithm cannot work when name siblings are

allowed.

When looking up an Interest in PIT, both EM and BPM are more suitable matching algorithms than the others. To see why, consider two cases: (1) PIT contains K_E^1 = "/text/A.txt/v1/s1" and K_E^2 = "/text/A.txt/v2/s6", and an Interest whose name is K_S^1 = "/text/A.txt" is looked up; and (2) PIT contains K_E^3 = "/text/A.txt", and an Interest whose name is K_S^2 = "/text/A.txt/v1/s1" is looked up.

First, an example of using EM is shown in Figure 3.1. In both $Case(1)$ and $Case(2)$, Interest matches none of the entries and is registered as a new entry. Thus, EM cannot aggregate Interests whose names are identical to the prefixes of entries in PIT, but EM is the only solution that can handle name siblings.

In contrast, BPM makes full use of active naming, as shown in Figure 3.2. Using BPM, Interests with names shorter than $K_E$are aggregated into the entries, as in $Case(1)$. Of course, BPM also requires priority rules to select a single entry. In $Case(2)$, K_E^3 is shorter than the Interest's K_S^2. However, K_S^2 cannot be aggregated to K_E^3 since the shorter key may match Data whose name is different from K_S^2 (e.g., "/text/A.txt/v2/s6") and the original Interest will be unsatisfied. Therefore, any matching entry with a shorter key is re-registered as a new entry with the longer key assigned to the Interest. It should be noted that this re-registering process takes advantage of active naming but disrupts the use of name siblings by eliminating the Interests with shorter names.

Finally, SPM and EPM are unsuitable for use in this context. Compared to BPM, these two matching algorithms cannot take full advantage of active naming; SPM cannot aggregate entries in $Case(1)$, and EPM cannot aggregate entries in $Case(2)$.

Table 3.1 summarizes the available algorithms for matching and selecting the entry in cases other than looking up an Interest in FIB or looking up Data in CS. The combination of SPM and LPM is used for FIB, and EM is used for looking up Data in CS. Only combination (I) is able to cope with name siblings, although all combinations support active naming.

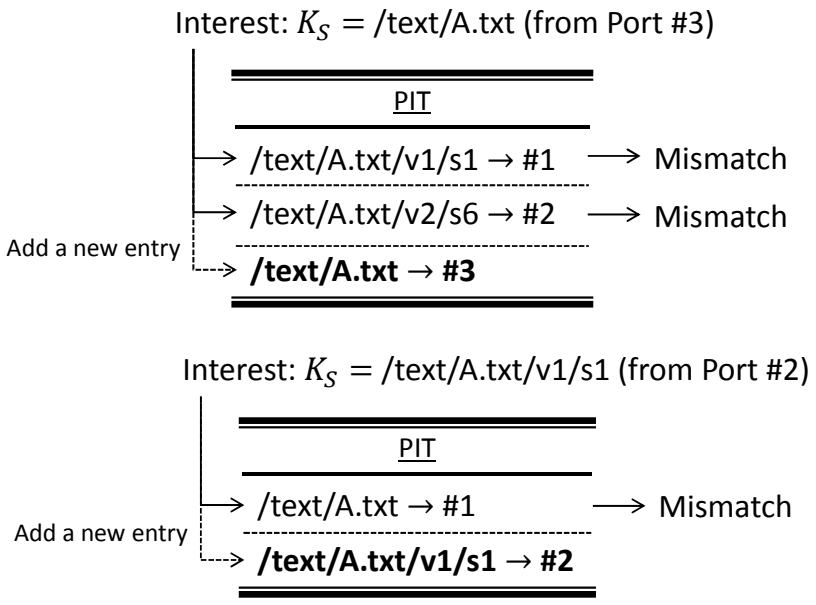


Figure 3.1: Example of Using EM (upper: $Case(1)$, lower: $Case(2)$)

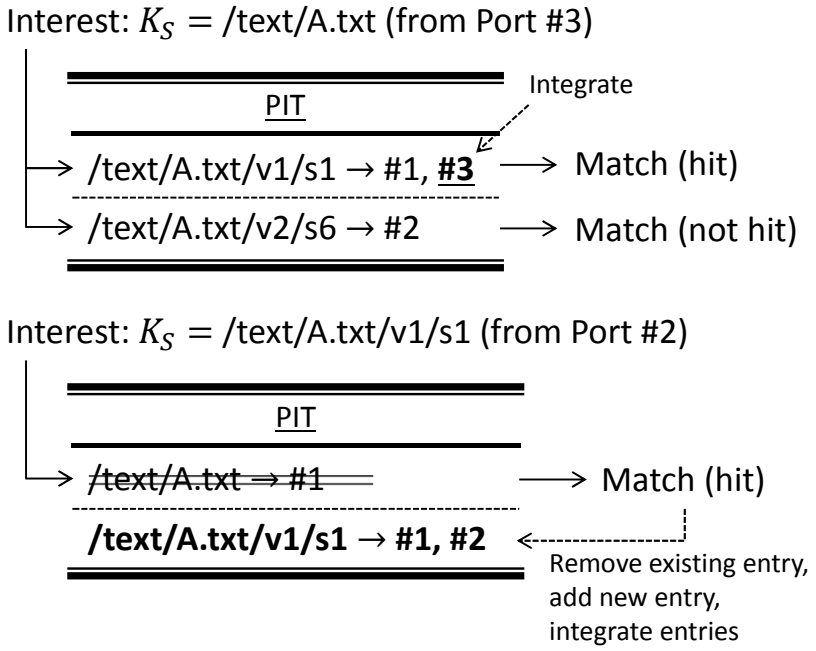


Figure 3.2: Example of Using BPM (upper: $Case(1)$, lower: $Case(2)$)

How to Realize CCN Capabilities

We note first that the specification of CCN is still in progress and may be changed significantly in the future. One possible way to implement a CCN router at this stage is to enumerate its possible capabilities by considering use cases of CCN, and use this enumeration to figure out what kind of functionalities would be needed in a CCN router. Active naming and name siblings are capabilities that would make CCN more flexible and useful [1, 3], although the current CCNx v1.0 specification does not include them [43]. One of the aims of this chapter is to represent the capabilities needed for CCN by a combination of algorithms and data structures implemented in CCN routers. We focus on active naming and name siblings in this chapter because of their influence on efficiency, management cost, and constraints of the CCN architecture.

Active naming, which is supported by all combinations shown in Table 3.1, allows a consumer to request content by specifying incomplete names. Suppose there is a consumer who does not know the name of the latest version of the A.txt file; however, the name of the Data packets that the user wants to retrieve requires specifying a version of the file and the order of the segments (e.g. the Mth segment of the Nth version of A.txt can be named "/text/A.txt/vN/sM"). By using active naming, the consumer can send an Interest request for "/text/A.txt" and receive the Data named "/text/A.txt/v2/s1", whose complete name indicates that it is the first chunk of version 2 of the file. In addition to its usefulness for retrieving content from a partial name, active naming can be used for applications that dynamically generate content.

In terms of the CCN router behavior, the complexity and efficiency of the operations to handle active naming vary according to the combinations of matching and selecting algorithms. The combinations (II)–(IV) shown in Table 3.1 require complex selecting algorithms for selecting the optimal entry to increase the cache hit rate of active naming.

For example, consider what happens when an Interest request for "/text/A.txt" arrives at a router containing cached copies of "/text/A.txt/v1/s1" and "/text/A.txt/v1/s2". If the router employs the combination (III) or (IV), then the Interest can be satisfied by the router instead of the content producer although the Interest matches both entries, and so the router needs to know how to decide which cached Data to choose. In this case, "/text/A.txt" may be taken as a request for "/text/A.txt/

v1/s1", which is the first chunk of A.txt. The decision may require all CCN routers to be familiar with the application-specific naming conventions and priorities, or require Interest requests that rely on active naming to contain additional information to identify the requested Data, such as by using a selector [1]. However, in the more complex case, where the Data for "/text/A.txt/v1/s1" are not cached in the router, the router will reply with undesired Data, such as "/text/A.txt/v1/s2", because the decision is based on the limited information available to the router. Thus, methods to avoid such errors are necessary (see Exclude [43] for an example).

There are similar problems with Interest aggregation in PIT. For instance, if a consumer uses active naming to request the latest version of A.txt and an Interest request for "/text/A.txt" arrives at a certain router simultaneously with another Interest request for "/text/A.txt/v1/s1", then the latest version of A.txt will be v2, but these two requests must be distinguished. However, both will be treated as the same request, for "/text/A.txt/v1/s1", by a router employing the combination (II) or (IV) unless information about versions is known in advance. To prevent problems resulting from not knowing the consumer's intention, a CCN architecture based on (II) or (IV) needs the additional mechanisms discussed above.

In contrast to the combinations (II), (III), and (IV), the combination (I) makes the selecting algorithms simple. In (I), it is unnecessary to avoid the conflicts between complete names and the ambiguous names produced by active naming, although this comes at a cost: Interests using active naming cannot be aggregated nor satisfied by intervening routers. In the above examples, an Interest request for "/text/A.txt" will fail to match the cache entries in CS and the unsatisfied requests in PIT; after that, the producer will directly satisfy the Interest according to the latest state of the content or the application.

Another important point to consider is the management of name siblings. The use of name siblings will cause undesired aggregation of requests and cache hits in the combinations (II)–(IV). To illustrate the problem, consider content named "/text/A.txt" that is different from content whose name is a strict extension of the original name, such as "/text/A.txt/v1/s1". If an Interest request for "/text/A.txt" matches an entry whose key is "/text/A.txt/v1/s1" in PIT or CS, then the consumer who sent the Interest request will ultimately receive the Data for "/text/A.txt/v1/s1", which is not the requested content. Therefore, name siblings cannot be permitted in the CCN architecture if any

Table 3.1: Summary of Matching and Selecting Algorithms

	CS-Interest	PIT-Data	PIT–Interest	Active naming	Name siblings
(I)	EM/-	SPM/LPM, FIFO[1], or all-hits	EM/-	Available	Available
(II)	EM/-	SPM/-	BPM/optimal	Available	Unavailable
(III)	EPM/optimal	SPM/LPM, FIFO, or all-hits	EM/-	Available	Unavailable
(IV)	EPM/optimal	SPM/-	BPM/optimal	Available	Unavailable

[1] FIFO: first in, first out

combination other than (I) is adopted. Furthermore, to prevent name siblings from being created by application errors or malicious attacks, dynamic mechanisms to eliminate inappropriate name siblings must be incorporated into the functions of CCN. In contrast, the combination (I) is free from cost of such management and still leaves the ability to creatively apply name siblings; for example, a shorter name can be used for content that contains the information needed to know the longer name and request its Data, such as the latest version and the number of segments.

In summary, the combination (I) should be adopted when avoiding the use of complex selecting algorithms for active naming and the costly management of name siblings is desired. In contrast, any of the combinations (II)–(IV) should be selected when efficient handling of packets using active naming is desired.

3.3 Architecture

Here, we describe the design principles of our proposed router architecture. In comparison with fixed-length IP addresses, variable-length name addresses impose a very high load for lookup. To handle variable-length name addresses at line speed, we introduce CAM and a distributed-and-load-balancing Bloom filter (DLB-BF) [42] into the prefix table; an associated element is the name lookup entity (NLE). An NLE provides a map between name addresses and entries in each of the three tables so that only one lookup is required to deterministically retrieve the most specific entry from among the three tables . DLB-BF, which allows name lookups to be performed in parallel, reduces the workload on the CAM. We also present the interest count entity (ICE), which is a new mechanism for identifying content worth caching.

Table 3.2: Name Lookup Algorithm Applied to Our Implementation

Packet	Storage	Matching Algorithm	Selecting Algorithm
Interest	FIB	SPM	Longest Match
	PIT	EM	-
	CS	EM [1]	- [2]
Data	PIT	SPM	Longest Match
	CS	EM [1]	- [2]

[1] This algorithm can be implemented as SPM.
[2] This algorithm can be implemented as LPM.

The most suitable combination of matching and selecting algorithms for the lookup mechanism using the CAM and Bloom Filter is combination (I) in Table 3.1. Table 3.2 shows the specific algorithms adopted in our implementation. If name siblings are disallowed (and therefore it is possible to use SPM as the matching algorithm), the combination (I) makes the lookup mechanism simple by choosing SPM for all matching algorithms except for the algorithm for the lookup of Interests in PIT, as shown in Table 3.2. For this reason, we assume that name siblings are not used. Additionally, Binary-CAM (BCAM) can be used instead of Ternary-CAM (TCAM); BCAM is more reasonable than TCAM with respect to cost and power.

Figure 3.3 illustrates the basic architecture of the proposed CCN router. First, an input packet is received by a Face element. After the packet is processed by the parser, its name and content are sent to an NLE and an ICE, respectively. NLE performs a lookup on the name and retrieves a pointer to a location in random access memory (RAM). ICE is used to avoid caching rarely requested data by counting how many Interest requests were made for the data. According to the results, an appropriate process, such as forwarding or caching, is determined. Finally, if the packet is to be forwarded, it is passed to an appropriate output face.

3.3.1 Name Lookup Entity

We propose NLE, which implements a fast lookup operation for name addresses. Almost all existing architectures that use a hash table sometimes yield a false positive, which results in a failure to forward packets. Preventing false positives in a hash table incurs a long delay to check that no

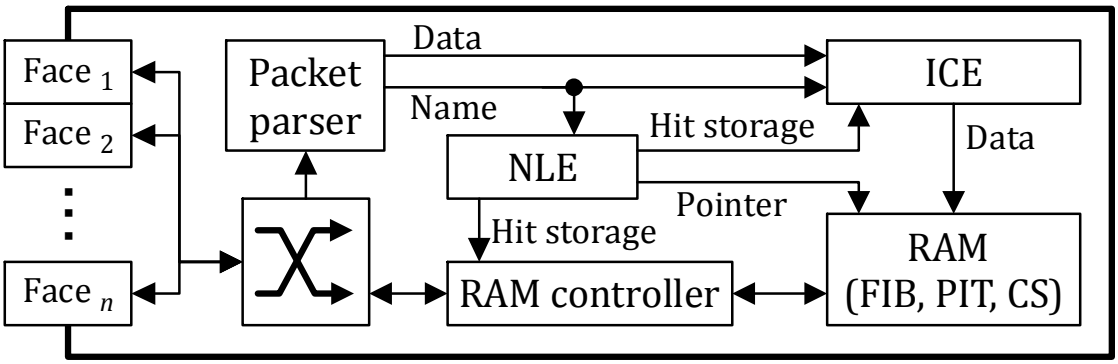


Figure 3.3: CCN Router Architecture

component of the searched name is falsely matched. Our approach avoids this issue by using CAM instead of a hash table. CAM can search its entire memory in a single lookup, but the cost and power requirements have been assumed to be prohibitive. We therefore propose a solution that splits the CAM into many small parts; this is expected to be less expensive than a single large memory solution. In addition, DLB-BF can dramatically reduce the load on CAM without sacrificing speed.

Because CAM stores fixed-length data words and name addresses are variable length, we must decide what to do when a name address is longer than the data word size. We divide such a name address into partial names and then simulate a hierarchical tree structure. We define three types of node: short name (SN), partitioned name (PN), and partitioned prefix (PP). An SN is used whenever the name is short enough to be store in a single data word; PN and PP are used otherwise. In terms of a tree structure, PN represents a leaf node and PP represents an internal node.

Figure 3.4 illustrates the definitions of fields of the node in the tree structure (i.e., the entry stored in CAM). W[bit] is the bitlength of CAM entries, and L[bit] is the bitlength of CAM addresses. "Address Flag" is set to 'TRUE(T)' in PN and PP, which use the "Address" field to store a link to the parent node. If "Prefix Flag" is true, this entry is PP, which is not a terminal node.

An example of several entries stored in CAM and RAM is shown in Figure 3.5. There are three names in NLE: N^A ="/aaa/.../bbb", N^B ="/aaa/.../ccc/.../ddd", and N^C ="/aaa/.../ccc/.../eee/.../fff". N^A is short enough to store in CAM as SN, while N^B and N^C are divided into two entries (N_1^B, N_2^B) and three entries (N_1^C, N_2^C, N_3^C), respectively. These divided entries are all of the same

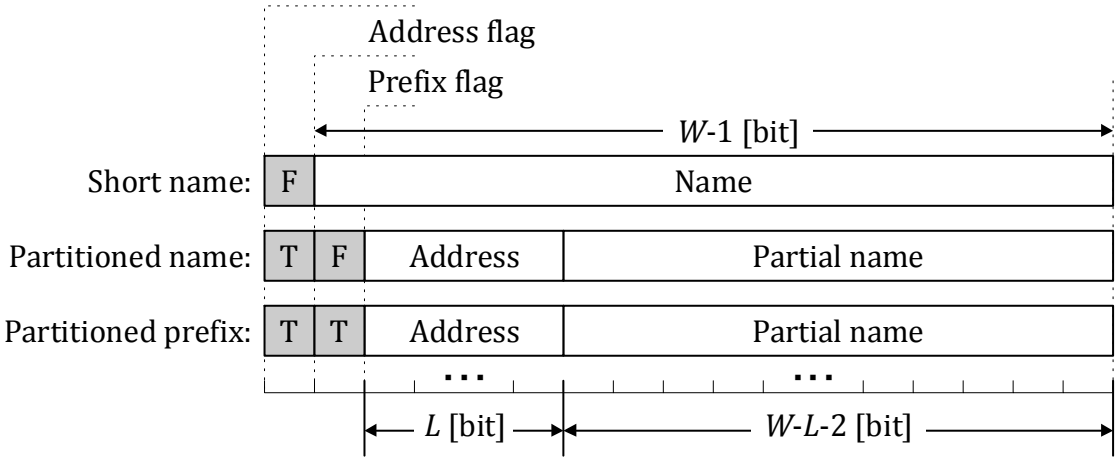


Figure 3.4: Definition of CAM Entry

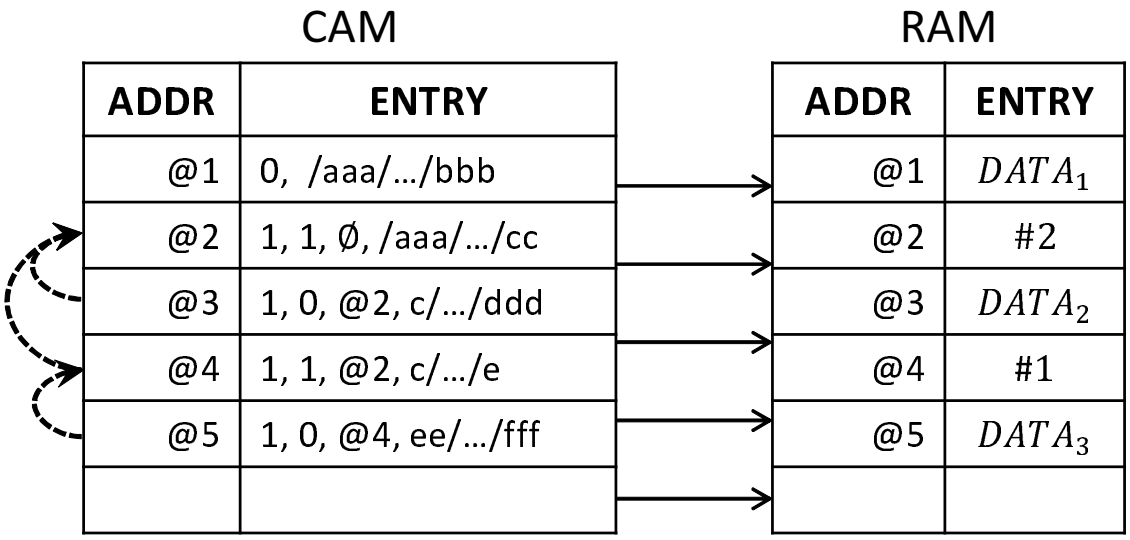


Figure 3.5: Example of CAM and RAM Entries in NLE

length: $W - L - 2[\text{bit}]$, as shown in Figure 3.4. The values of an entry in CAM correspond to the definition in Figure 3.4.

N^A is stored in CAM and RAM as an SN (a single entry), and so we need only the name address to retrieve the data from RAM. The process to retrieve the data corresponding to N^B, which is too

long to pack into SN, is as follows: a) divide N^B into N_1^B"/aaa/.../cc" and N_2^B ="c/.../ddd", where N_1^B and N_2^B = are used to search PP and PN, respectively; b) perform a lookup for N_1^B as PP with the Address field set to 0 (because the tree structure starts at this PP); c) create a search key as a PN from the name N_2^B and the address of the parent node N_1^B; and d) retrieve the data from RAM located at the address '@3', which was specified by searching the PN. Note that a lookup for a PN or PP entry requires the address of the parent PP. The lookup for N^C is performed in a similar way, although it requires one more PP lookup.

Since N^B and N^C share the prefix of name "/aaa/.../cc", the two entries for N_2^B and N_2^C, which are located at '@3' and '@4', respectively, assign the same value '@2' to the PP. The number of child nodes that have a references to this PP is held at the entry located at '@2' in RAM; we find the value '#2' there. This value is incremented whenever a new child node is registered and decremented whenever a child node is removed, allowing us to remove the PP entry when the count becomes zero.

It is worth noting that it is rare to perform multiple lookups for a long name like N^B and N^C, and doing so causes high latency. In accordance with the fact that 99% of domain names are no longer than 40 bytes [14], almost all name addresses can be stored in SN by setting $W = 320$. Entries of this length are supported in an existing CAM implementation.

3.3.2 Interest Count Entity

ICE is an entity used to avoid caching rarely requested data by counting Interest requests for the data. In general, the popularity of Internet traffic approximately follows Zipf's law. This means that a small amount of popular content accounts for the majority of requests. To exploit this characteristic, we propose ICE as a means of caching based on the number of requests for each piece of content. ICE counts the requests for each name, and only those Data whose frequency exceeds a certain threshold are cached. Thus, ICE prevents content that is requested once or just a few times from occupying limited cache space. Algorithms 1 and 2 show pseudocode of these ICE processes.

Algorithm 1 Interest Process in ICE

```
1:  procedure ITERESTPROCESSINICE(hitCS, name)
2:      if hitCS = True then
3:          return
4:      end if
5:      entry ← ICETable[H(name)]
6:      c ← entry.count
7:      n ← entry.name
8:      if n = name & c > 0 then
9:          entry.count++                    ▷ The existing counter is incremented.
10:     else
11:         entry.name ← name                ▷ A new entry is created (or overwritten).
12:         entry.count ← 1
13:     end if
14: end procedure
```

Algorithm 2 Data Process in ICE

```
1:  procedure DATAPROCESSINICE(hitCS, name, data)
2:      if hitCS = True then
3:          return
4:      end if
5:      entry ← ICETable[H(name)]
6:      c ← entry.count
7:      n ← entry.name
8:      if c > THRESHOLD & n = name then
9:          AddCS(name, data)                ▷ The Data item is cached.
10:     end if
11: end procedure
```

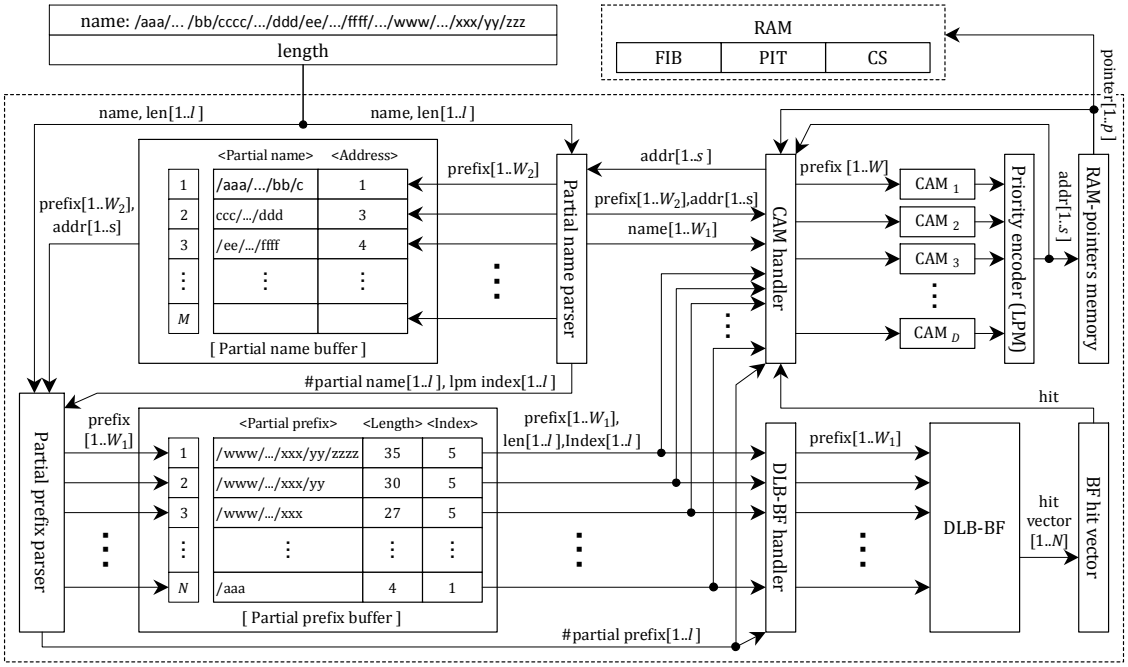


Figure 3.6: Hardware Design of NLE

3.4 Hardware Design

3.4.1 Name Lookup Entity

We now describe a detailed implementation of NLE. Figure 3.6 shows the hardware design of NLE. Roughly, NLE consists of four components: the unit to process *partial names* (upper left), the unit to process *partial prefixes* (lower left), DLB-BF (lower right), and CAM (upper right).

Name lookup is performed as follows. First, the input name is partitioned into fixed-length partial names if necessary. A partial name is a W_2-bits-wide segment of a name that is too long to store in a single entry (as SN in Figure 3.4). Since looking up a child node requires the address of its parent node, as discussed in 3.3.1, a buffer for partial names contains not only its string but also an address for the partial name. Second, a partial name and SN are further split into partial prefixes delimited by the character '/'. A buffer for partial prefixes both stores the prefixes and remembers the indexes of the partial names to which the partial prefixes correspond. Third, queries in DLB-BF

for the partial prefixes are executed in parallel; the CAMs then search the partial prefixes for which a membership query to the DLB-BF yields true. Finally, a pointer is obtained from the resulting address and used to retrieve the data from RAM.

To reduce the cost and power requirements of the system, we split the monolithic CAM into D smaller CAMs. In general, the price of large memory is higher than the price of the same amount of memory in smaller pieces. The power required to search CAM is proportional to the size of the CAM. Thus, many small CAMs will have lower power requirements than a single large CAM. Additionally, the distributed CAMs make it easier to perform a lookup operations in parallel, which improves throughput significantly.

W is the length of a CAM entry, and W_1 and W_2 are determined according to W: W_1 is the maximum length of "Name" defined in Figure 3.4, and W_2 is the maximum length of "Partial Name" defined in the same place. Two conflicting characteristics are desirable for W. It should be large enough to avoid CAM lookups by PP and achieve a single CAM lookup; however, large values of W cause wasted space from storing short variable-length names into fixed-length CAM entries. We are going to investigate and optimize W in light of this tradeoff.

3.4.2 Interest Count Entity

The hardware design of ICE is illustrated in Figure 3.7. ICE is implemented as a simple hash table with name addresses as key is a name address and the counts of Interest as values. Because hash collisions may occur, ICE also holds the complete name address.

We now present the caching algorithm with ICE. When receiving an Interest, an entry containing a count of requests for content with the specified name is retrieved by using the name of the input Interest. If the count is zero or the name stored in the counter is different from the input name, the existing entry is overwritten with the new name, and the count value is reset to one. Otherwise, the count value is incremented. When Data that have not yet been cached arrive at the router, the data are cached in CS when the count value is larger than a certain threshold.

ICE makes it possible to cache only content for which caching will improve performance. Since most content is rarely requested, the limited resources of CS and CAM would be exhausted by

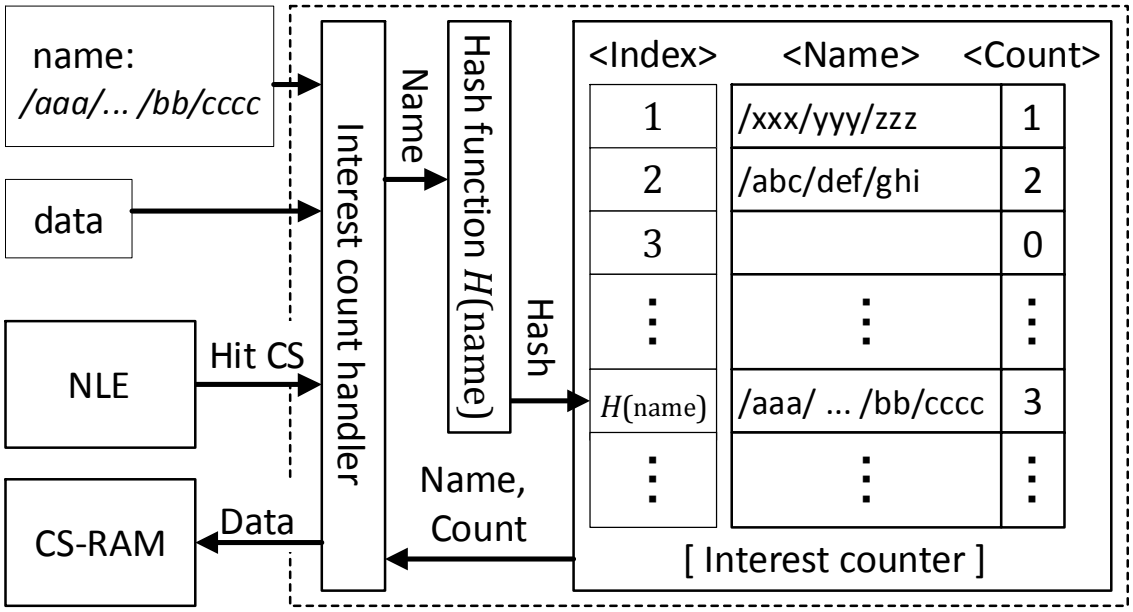


Figure 3.7: Hardware Design of ICE

simple caching. In contrast, the method of caching content that has a number of requests more than a certain threshold can be much more memory efficient. According to [37], ICE needs approximately one tenth the capacity of a universal cache.

Additionally, we can dynamically adjust the threshold according to network traffic volume. The number of requests for even unpopular content can be greater than a few if heavy traffic is handled. Furthermore, network traffic can vary hourly and daily. For these reasons, a fixed threshold is not ideal; however, a variable threshold can be used to maintain a desired cache hit ratio by adjusting the threshold in response to volume or characteristics of network traffic. The adjustment process is challenging because the first few times that content is requested, the returned data will not be cached but only counted. A method to determine suitable thresholds is left to future work.

3.4.3 Summary

Packet processes in our proposed CCN router processes are summarized in Figure 3.8. All packets received by the router are transmitted, integrated, cached, or satisfied with cached Data, with the

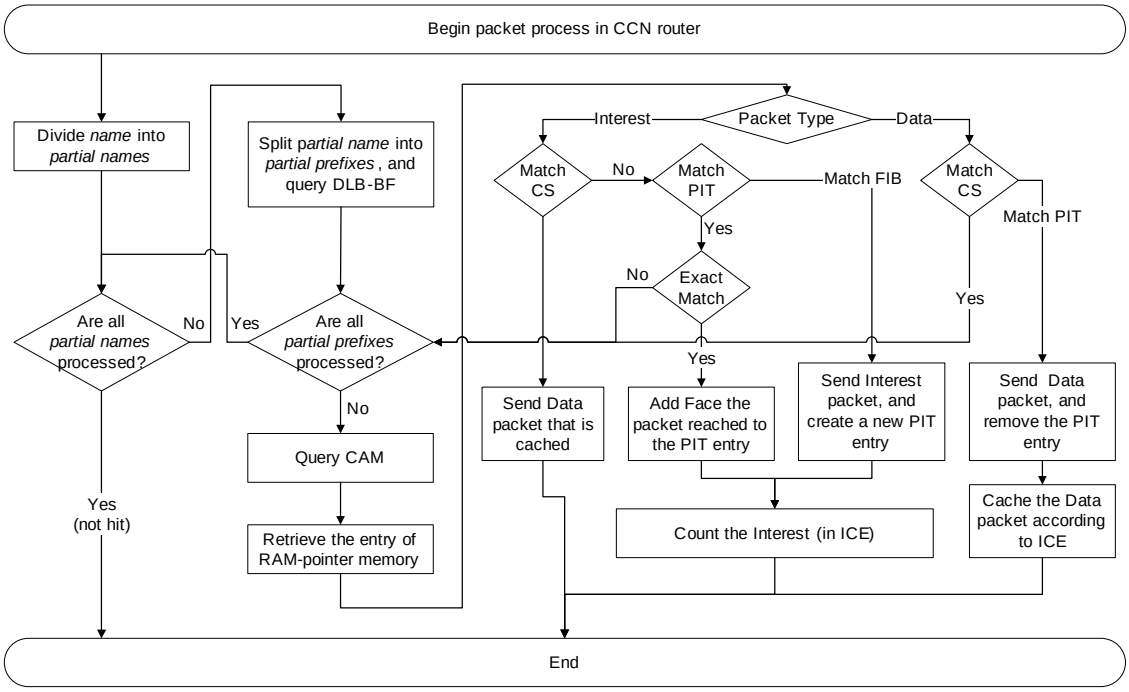


Figure 3.8: Packet Processing in the Proposed CCN Router

action decided according to the flow chart.

NLE consisting of DLB-BF and CAM implements a lookup system based on LPM. The case where Interest matches PIT, however, requires checking whether the match is an EM since matching and selecting algorithms follow the strategy illustrated in Table 3.2. The check can be simply implemented in the obvious way: we need know only the indexes of both buffers (i.e., a buffer for partial names and a buffer for partial prefixes) to know where the matching prefix is stored. If the prefix's indexes in both buffers are 1, the matching partial prefix is essentially identical to its complete name, and the match is identified as an EM. Otherwise, the match is a non-exact match; therefore, the processes for partial prefixes should be repeated to continue.

When a Data item matches an entry in CS, the process does not stop but instead continues to run, as shown in Figure 3.8. Although the match appears to be proof that the Data item is cached in CS and that the Interest requesting the Data has been satisfied by the cache, active naming requires additional verification processes. For example, if CS contains a cache of Data named

"/video/a.mpg/v1/s1", an Interest named "/video/a.mpg" will not match the cache entry because the Data's name is not identical to any prefixes of the Interest's name, and the Interest is expected to be satisfied with the returned Data. If the returned Data is named "/video/a.mpg/v1/s1", which is identical to the name of the cached content, the Data matches the cache entry. Without continuing the process in this case, the Interest named "/video/a.mpg" cannot be satisfied.

3.5 Evaluation

In this section, we analyze the performance of our CAM-based CCN router and discuss its feasibility and challenges to widespread adoption. We calculate the required memory size, cost of the memory, and throughput on the assumption of a table with 10 million entries, average packet size of 256 bytes, Interest packets of 40 bytes, and Data packets of 1500 bytes; these values are the same as in [30, 16]. In addition, we assume that 99% of existing domain names are no longer than 40 bytes and have no more than six components [14]. We also experimentally evaluated the performance, testing the hardware design implemented in Quartus II using Verilog HDL.

Existing lookup mechanisms that exclude false positives and our proposed mechanism are compared on searches per second, throughput, and size of static random access memory (SRAM); the results are shown in Table 3.3. MATA-NW seems to be fast enough for our purposes, but its throughput is achieved by employing a pipeline to a GPU. Both the pipeline process and the GPU make it hard to reduce the latency of each packet process, although the pipelined process outputs the packets at high-speed. In this section, we demonstrate that NLE, which is our proposed lookup mechanism, can achieve higher throughput and much lower latency than the existing methods by using CAM and DLB-BF. The size of SRAM required for NLE is expected to be at most 576 MBytes of SRAM. Nevertheless, a large portion of the memory can be implemented with dynamic random access memory (DRAM) instead of SRAM. NLE is also expected to require 3.2 Gbits of CAM in addition to the SRAM. The approaches to cope with these challenges rely on splitting the CAM, and are discussed later.

Table 3.3: Performance of Lookup Mechanisms (Number of Entries: 10 Million.)

lookup mechanisms	Search (per s) [MSPS]	Throughput [Gbps]	SRAM size [MB]
Character Trie[13]	3.172	6.344	1,026.34
NCE [14]	4.017	8.034	718.44
ENPT [15]	20.67	41.34	116.02
NameFilter [13]	37.003	74.006	234.27
ATA (200μs latency) [16]	6.56	13.12	682.55
MATA (100μs latency) [16]	29.75	60.50	490.28
MATA-NW (100μs latency) [16]	63.52	127.04	490.28
NLE (proposed method)	81.6	167.116	576.0

3.5.1 Memory Size and Cost

Scalability is limited by the scalability of the CAM, and so the necessary amount of RAM is determined according to the anticipated number of CAM entries. We therefore discuss how much memory is required to implement NLE. For this determination, it is important that NLE consists of two buffers, DLB-BF, and CAM.

The required sizes of the partial name buffer S_N[bit] and the partial prefix buffer S_P[bit] can be calculated as follows:

$$S_N = M(W_2 + s), \tag{3.1}$$

$$S_P = N(W_1 + 2l), \tag{3.2}$$

where M is the number of entries in the partial name buffer; N is the number of entries in the partial prefix buffer; W_1 and W_2 are the bit-lengths of the buffers, as shown in Figure 3.6; s is the bit-length of the $< Address >$ field; and l is the bit-length of the $< Length >$ and $Index$ fields in Figure 3.6. According to the size of domain names mentioned above, we define $W = 40$[Bytes], as the size of a CAM entry and the upper size limit of an entry in the buffers shown in Figure 3.4. The buffer for partial names, whose capacity should be large enough to store complete name addresses, needs more than 37 entries to store a name whose length is the maximum transmission

unit; therefore, we set $M = 32$ (cf. Figure 3.6). Since it is desirable to store all components into the buffer for partial prefixes, we set $N = 64$ according to the fact that the longest URL has roughly 70 components [30]. To achieve M and N, the partial name buffer needs 10 Kbits and the partial prefix buffer needs 22 Kbits; these buffer sizes are reasonable, although we must still investigate how parallel processing scales in terms of wiring cost.

The size of DLB-BF depends on the probability of a false positive. If m is the number of bits in the array, k is the number of hash functions, and n is the number of elements inserted into DLB-BF, the false positive probability α can be calculated as follows [42]:

$$\alpha = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k \simeq \left(1 - e^{\frac{kn}{m}}\right)^k. \tag{3.3}$$

Since $k = \frac{m}{n} \log 2$ minimizes the probability α, the equation (3.3) results in the following expression:

$$\alpha = \left(\frac{1}{2}\right)^{\frac{m}{n} \log 2}, \tag{3.4}$$

which can be simplified to

$$\frac{m}{n} = -\frac{\log \alpha}{(\log(2))^2}. \tag{3.5}$$

By substituting $\alpha = 10^{-x}$ into Equation (3.5), we finally obtain the following equation:

$$\frac{m}{n} = \frac{\log 10}{(\log(2))^2} x \simeq 4.7925 \times x \tag{3.6}$$

This means that extending the length of each entry by about 4.8 bits decreases the probability of a false positive 10-fold.

The size of DLB-BF is shown in Figure 3.9. When $\alpha = 10^{-6}$ and $n = 10M$, the amount of memory required for BF is 288 Mbits, and this grows to 4.6 Gbits upon assigning 16 bits to each entry for implementing counting filters (denoted "CBF" in Figure 3.9), which allow deletion of entries. For practical purposes, at most 2.3 Gbits of SRAM (8 $\times$ 288 Mbits of SRAM) is currently available. DLB-BF can be simply scaled down by increasing the probability of a false positive

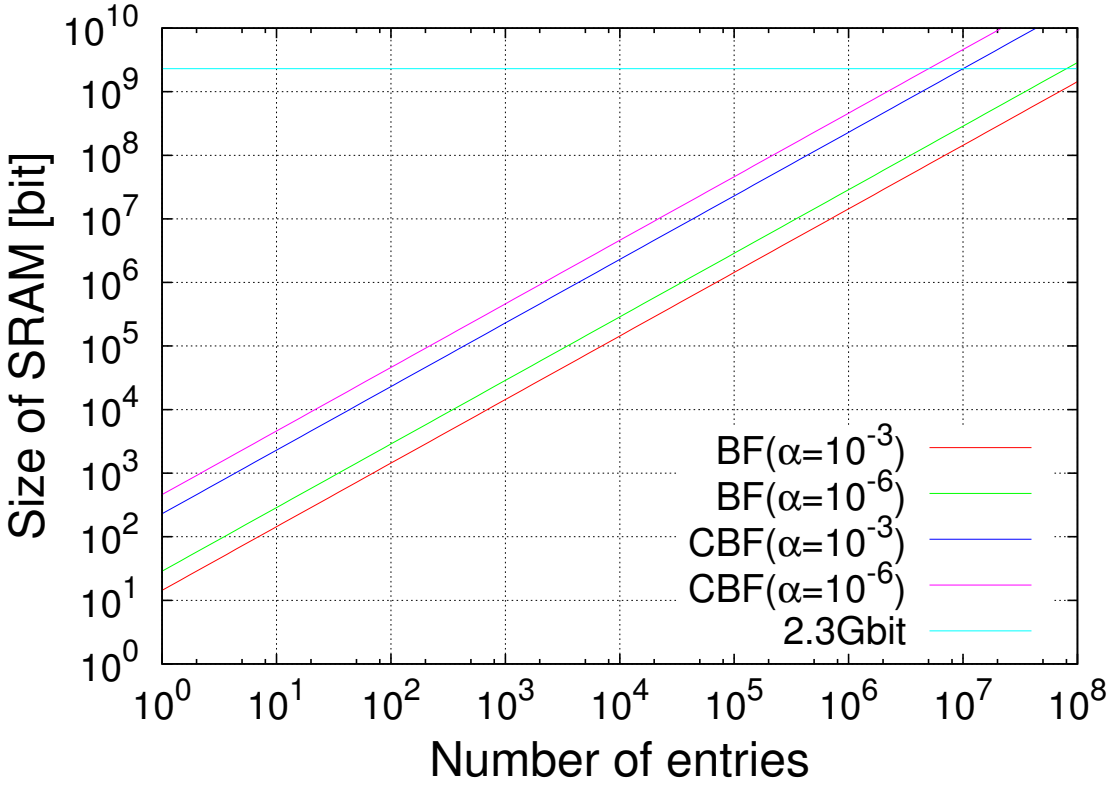


Figure 3.9: The Size of SRAM Required for NLE

from $\alpha = 10^{-6}$ to $\alpha = 10^{-3}$. We can also conserve SRAM by using DRAM or RLDRAM for implementing counting filters. Then, the required amount of SRAM can be 288 Mbits, even when $\alpha = 10^{-6}$. If DLB-BF is implemented on SRAM, whose cost is, at present, approximately 1 USD/ MB [44], then 4.6 Gbits will cost 576 USD, and 288 Mbits will cost 36 USD. On the other hand, a 4.6 Gbits DLB-BF implemented on DRAM will cost only 4.50 USD (using a present value for DRAM of 1 USD/ 128 MB).

The memory required for CAM is the most serious problem because of the limited size available. When $W = 40$[Bytes], CAM requires 3.2 Gbits to hold 10 million entries. Although a single CAM with capacity on the order of gigabits does not exist, it is easier and more efficient to arrange many small CAMs. Since 1 Mbit of CAM currently costs about 1 USD, we can estimate the cost of the CAM to be 3200 USD. As a result, the total memory cost can be estimated at 3776 USD.

Table 3.4: Number of Read/Write Accesses in the Name Lookup Process, by Memory Type

		PNB[1]		PPB[2]		DLB-BF		CAM	
R(read)/W(write)		R	W	R	W	R	W	R	W
Lookup	min	0	0	1	1	1	0	1	0
	ave	$\bar{m}$	$\bar{m}$	$T(\bar{m})$	$S(\bar{m})$	$T(\bar{m})$	0	$\bar{m} + \alpha S(\bar{m})$	0
	max	$m_{\max}$	$m_{\max}$	$T(m_{\max})$	$S(m_{\max})$	$T(m_{\max})$	0	$m_{\max} + S(m_{\max})$	0
Add (in CAM)	min	0	0	0	0	0	0	1	0
	ave	$\bar{m}$	$\bar{m}$	0	0	0	0	$\bar{m}$	0
	max	$m_{\max}$	$m_{\max}$	0	0	0	0	$m_{\max}$	0
Add (not in CAM)	min	0	0	0	0	0	1	1	1
	ave	$\bar{m}$	$\bar{m}$	0	0	0	1	$\bar{m}$	1
	max	$m_{\max}$	$m_{\max}$	0	0	0	1	$m_{\max}$	1

[1] Partial Name Buffer
[2] Partial Prefix Buffer

3.5.2 Throughput

The throughput of the CCN router strongly depends on the access time of NLE. The lookup operation in NLE requires accesses to two buffers, DLB-BF (implemented in SRAM), and CAM. Table 3.4 describes the minimum/average/maximum number of read/write access to the memory pools. In Table 3.4, α is the probability of a false positive from DLB-BF, $\bar{m}$ is the average number of partial names, $m_{\max}$ is the maximum number of partial names obtained from a name, and $S(m)$ and $T(m)$ are defined as follows:

$$S(m) = \sum_{i=1}^{m} n_i, \;\; T(m) = \left\lceil \frac{S(m)}{N} \right\rceil,$$

where n_i is the number of partial prefixes obtained from the ith partial name and N is the number of entries in the partial prefix buffer. Table 3.4 covers three operations: the lookup process, the add process used when the added entry exists in CAM, and the add process used when the added entry is absent from CAM.

Although NLE is designed to handle names too long to store into a single entry, in practice, almost all names can be stored as a single entry and processed in a single buffer access (i.e., $\bar{m} \leq 1$) by setting $W = 40$. In addition, we can reduce the number of write accesses to the partial prefix buffer from $S(m)$ to m by parallelizing the process of writing partial prefixes. As a consequence,

the access times required for the lookup and add operations are approximated as follows:

$$\begin{aligned} T_{\text{lookup}} &= (1.0 + 0.45) \times (2\bar{m} + 2T(\bar{m}) + S(\bar{m})) \\ &\quad +(1.0 + 4.0) \times (\bar{m} + \alpha S(\bar{m})) \\ &= 1.45 \times (2 + 2 + 1) + 5.0 \times (1 + 0) \\ &= 12.25[\text{ns}] \\ T_{\text{add}} &= (1.0 + 0.45) \times (2\bar{m} + 1) \\ &\quad +(1.0 + 4.0) \times (\bar{m} + 1) \\ &= 1.45 * (2 + 1) + 5.0 * (1 + 1) \\ &= 14.35[\text{ns}]. \end{aligned}$$

If the SRAM access time is 0.45 ns, then the CAM access time will be 4.0 ns, and the buffer access time will be 1.0 ns [30]. This access time results in a throughput for lookups of 81.6 million searches per second (MSPS) and a throughput for the add operation of 69.7 MSPS. With an average packet size of 256 bytes, these throughputs are roughly equivalent to 163 Gbits/s and 139 Gbits/s, respectively.

We also need to evaluate the environment in practical use, where there will be not only lookups but also updates. Unlike with IP packets, almost all CCN packets arriving at a router are looked up and result in updates to the tables in the router (i.e., an Interest will add a new PIT entry or update an existing PIT entry, and sending Data will remove the corresponding PIT entry). NLE can perform the lookup and update concurrently; therefore, the lookup and following update requires only an additional process to add/remove the matching entry to/from CAM and DLB-BF. The average access time for this operation is approximated as follows:

$$\begin{aligned} T_{\text{update}} &= T_{\text{lookup}} + (1.0 + 0.45) \times 1 + (1.0 + 4.0) \times 1 \\ &= 12.25 + 6.45 \\ &= 18.70[\text{ns}]. \end{aligned}$$

The throughput is 53.5 MSPS (109 Gbps), assuming the same parameters as above. Thus, we can realize CCN router processing at line speed. These throughputs could be greatly improved by designing pipeline mechanisms for concurrent lookups.

In addition to the above analysis, we designed an experimental implementation of NLE in Quartus II using Verilog HDL. The implementation is simplified in terms of the bit-widths and memory accesses. Specifically, the bit-width of each name is set to be 1/16th of that in the design discussed above, and a simple equivalent circuit replaces the SRAM and CAM needed for DLB-BF and CAM. By using the timing analyzer in Quartus II, we evaluated the maximum clock speed in Cyclone-V GT (5CGTFD9E5F35C7) and Stratix V (5SGSMD3H1F35C1); these gave achieved 177.84 MHz and 517.6 MHz, respectively. In the future, we plan to improve the hardware design and evaluate the feasibility in terms of the required power and number of logic elements.

3.6 Summary

This chapter contributes evidence for the feasibility of CCN by designing concrete CCN router hardware and evaluating its performance. Needless to say, it is a requirement for implementation of CCN that CCN routers be feasible. In addition, accurate estimates of actual performance are essential to all sorts of network-level simulations. We addressed these problems by proposing CAM-based CCN router architecture. We proposed NLE, which consists of many small CAMs and DLB-BF and allows reasonable costs, and ICE, which assists in adaptive caching; thus, we have shown the entire design of a CCN router. We also performed a basic theoretical analysis of the expected throughput and cost of the CCN router.

As discussed in Section 4.1, we split the monolithic CAM into smaller parts to make NLE feasible. A significant challenge for our architecture is to scale the memory capacity and the number of entries. There are no existing TCAMs with more than 100 Mbits of memory capacity. In addition, the power cost of a TCAM can be approximated as 1 kW/Mbit; the power requirements of our router, which needs at least 3.2 Gbits of CAM, can rise to more than 3 kW; however, even a 1 kW power requirement is beyond the capacity of any existing implementation by several orders of magnitude. These challenges can be addressed by using multiple small CAMs. A sufficient number

of small CAMs can dramatically reduce the costs for capacity and power. Employing 32 units of small CAM instead of 1 large monolithic CAM, for example, allows NLE to be implemented on an existing 100-Mbit TCAM chip. As a rough estimate, the distributed CAMs require only $1/32$ of the power of a large single CAM. We can adjust the number of CAMs running in parallel to achieve the desired trade-off between the power and the lookup performance.

FIB is required to handle websites, the number of which is approaching 1 billion according to a survey in [45]. The line-speed (40 Gbps) traffic, whose average round-trip delay time (RTT) is $\mathrm{RTT} = 100\mathrm{ms}$, imposes 2 million entries per port on PIT. Even if the effect of ICE is maximized, the number of chunks that would be stored in CS for 10 million entries is equivalent to the amount of files accessed per day in terms of city-scale traffic [37].

In the future, it seems likely that such numbers increase rapidly along with the number of content items. Our architecture can easily scale with this increase in the number of names by installing additional distributed CAMs according to the number of entries required for NLE. A simple solution—ignoring lookup time—is to partially replace CAM with a hash table; however, the limits of the system can be relaxed without sacrificing speed because we can use not only 16 T / cell TCAMs but also 10 T / cell BCAMs, and we expect exponential growth in the capacity and availability of feasible memory.

Chapter 4

Compact CAR: Low-overhead cache replacement policy for an ICN router

4.1 Introduction

In this chapter, we argue that ICN requires a novel cache replacement algorithm to fulfill the requirements in the design of a high performance ICN router. Then, we propose a novel cache replacement algorithm to satisfy the requirements named Compact CLOCK with Adaptive Replacement (Compact CAR), which can reduce the consumption of cache memory to one-tenth compared to conventional approaches.

In ICN, individual routers can be turned into caching devices by simply providing physical cache memory for them. This feature of ICN that all network devices have caching capability is called in-network caching function, and several ICN architectures, CCNx[1], NDN[3], SAIL[5] and PURSUIT[4], have already suggested utilizing the function to take several advantages of caching system such as reducing network access latency, alleviating network traffic, balancing network load, and achieving robustness against a single failure scenario. In this sense, ICN can be considered as a largely distributed caching architecture whose performance depends on mainly two factors: where to cache and how to cache contents. The former and the latter are known as content placement and cache replacement problems, respectively.

While the problem of content placement has attracted much attention in ICN research communities, that of cache replacement has been relatively ignored since many people believe that the problem has already been investigated intensively in the context of web-caching and a content delivery network (CDN). However, it is unclear that the conventional cache replacement approaches are suitable for ICN due to following two reasons. First, core ICN routers are expected to meet the speeds required for line-rate operation, especially by exploiting limited memory and computational resources. However, the conventional cache replacement approaches are designed for end-device operation rather than for core-device operation, which should be carried out in parallel with forwarding operation. Second, the fine granularity of chunks in ICN, namely chunks or segments, changes the traffic access patterns of request messages, which dramatically govern the performance of a cache replacement algorithm.

In the light of the observation above, this chapter studies the cache replacement problem in the core ICN routers. First, we discuss the access patterns of contents to understand its relation to cache replacement algorithms. Second, we focus on CLOCK, which is a classical cache replacement policy to achieve low-complexity LRU approximation, to support the line-rate operation in core ICN routers. Then, we propose a novel cache replacement algorithm named Compact CLOCK with Adaptive Replacement (Compact CAR) to fulfill the requirements. The numerical simulation shows that the proposed cache replacement algorithm can reduce the consumption of cache memory to one-tenth compared to conventional approaches.

This chapter is organized as follows. In Section 4.2, we review related research works. In Section 4.3, we describe the design considerations of a cache replacement algorithm for a core router of ICN. This is followed by a detail description of our proposed method Compact CAR in Section 4.4. In Section 4.5, we evaluate our protocol through extensive simulations. Then, we discuss on some implementation issues of our proposal, especially for the design of high performance of ICN core routers in Section 4.6. Finally, we conclude this article in Section 4.7.

4.2 Related works

There are a considerable number of cache replacement algorithms, ranging from those available in a computer system (e.g., CPU and I/O buffers) to those used in communication networks (e.g., web-proxies and CDNs). Thus, there are various requirements and methods suitable for individual environments. To understand the requirements of in-network caching in ICN, we review several cache replacement algorithms that have been carried out in the different context.

Replacement algorithms are developed originally for the purpose of paging in the computer system [46, 38]. The bottleneck of the systems is the latency of fetching pages from slow auxiliary memory to fast cache memory. On the one hand, the hardware cache such as CPU commonly used First-in, first-out (FIFO) and Not Recently Used (NRU) to reduce the memory and computational cost because of the hardly limited resources. On the other hand, the software cache such as virtual memory in OS commonly adopts LRU and LFU, which increase cache hit rate with cost maintaining a data structure or/and statistical information (i.e., the number of references to a page).

As researchers uncover problematic access patterns that degrade the cache hit rate of the algorithms, many variants of LRU and LFU are devised to overcome the problems. 2Q [47], ARC [46] and LIRS [48] improve the hit rate by exploiting the advantages of LRU and LFU while their time and space complexity are comparable to those of LRU. In contrast to them, CLOCK [49] reduces the complexity of LRU by approximating its behavior with a fixed circular buffer while keeping the hit rate. The complexity of CLOCK is comparable to that of NRU which has a low computational cost. CAR [38] combines CLOCK with ARC to achieve both hit rate improvement and cost reduction.

Since web services became explosively popular, web-cache and CDN-cache were researched intensively to improve the performance of them in terms of bottleneck, latency, overload and robustness [50, 51, 52]. Because the resource constraints of them are more moderate than that of computer systems, the cache replacement algorithms in a web and a CDN utilize statistical information including not only recency and frequency but also several others including size, latency, and URI [51]. However, the improvement was trivial or specific to particular environments in spite of an abundance of caching algorithms [52].

In recent years, ICN has revived research on caching algorithms because ICN provides inherent in-network caching feature. Unlike web-cache and CDN-cache employed in the application-layer, all network devices in ICN have caching capability. Thus, caching related researches in ICN have been carried out intensively, especially concerning the locations of content placements. Also, a few of cache replacement algorithms were introduced in [20, 21].

Previous chapters on caching use only LRU [17, 18] or claim that the effect on performance of cache replacement is minimal [19]. However, the chapters use only blunt cache replacement policies and ignore the suitability for network traffic. In fact, there are studies that exhibit the capability to improve the performance of a network [20, 21]. Cache replcement policy based on content popularity (CCP) [20] can significantly decrease the server load and increase cache hit rate compared to that of LRU and LFU. The work in [21] analyzed the effects of chunking and proposed Highest cost item caching (HECTIC), which uses a utility-based replacement algorithm and outperforms existing polices including LRU. Their statistical approaches are too expensive to be employed in an ICN core router due to computational and memory costs. However, we propose a low-overhead cache replacement policy that outperforms LRU-based and simple replacement policies by coping with access patterns specific to ICN.

To realize ICN, especially an ICN core router, it is also required to implement a cache replacement algorithm that can be operated with severe resource constraints instead of the statistical caching algorithms for a web and a CDN with rich resources. The implementation cost of commonly used approaches such as LRU and LFU is also prohibitive for router hardware, as pointed out in [22, 19]. Looking back at the history of cache replacement algorithms, ICN core routers need a hardware implementable approach whose complexity is comparable to that of FIFO or CLOCK.

4.3 Design Considerations of Cache Replacement Algorithm for ICN

4.3.1 Access Patterns of Traffic in the Network

To understand cache replacement strategy suitable for network traffic without expensive mechanisms (e.g, LFU and statistical approaches), we focus on an access pattern. An access pattern is

the important factor to govern the performance of cache replacement algorithms [46, 48, 47, 53]. It is well known that the popularity of contents follows a Zipf-like distribution: a large number of contents are requested only once or just a few times [54]. Although the exact access pattern of the network level traffic in ICN is not known due to the lack of available ICN traffic trace, such one-time used contents occupy 60% or more in the network level traffic in IP networks [37].

A sequence of requests to such one-time used contents forms an access pattern called SCAN [46, 38]. SCAN makes the performance of an LRU-based approach much poor because such unpopular contents occupy the whole cache. In particular, ICN is able to identify a chunk (its default size is 4K bytes in CCNx), which enables the chunk level caching in an ICN router. Thus, we conjecture that the distribution of the "chunk popularity" would have heavier tail than Zipf-like distributions, which renders the effect of SCAN more serious.

In addition, the distribution of content popularity changes frequently. The volatile popularity is also a problematic access pattern because it hinders the strategies depending on statistical information (including LFU) from replacing the out-of-date chunks that were accessed frequently. We conjecture that these access patterns would be frequently observed in ICN due to the volatile popularity observed in social networks that share user-generated contents as well as real-time applications such as video chatting.

For the reason above, the cache replacement algorithm for ICN should be able to deal with the access patterns described above. Among the conventional cache replacement algorithms, CAR is able to efficiently deal with the access pattern [38]. CAR is resistant to SCAN traffic access pattern due to its dual lists which enable to distinguish popular and non-popular contents. CAR is also resistant to the volatile popularity because of strategy based on limited-frequency. Our proposal is based on CAR to inherit these features.

4.3.2 Computational Power and Memory Limitations

In the design of the cache replacement algorithm, two of the performance metrics should be considered. One is the cost that updates the table holding the information of cached items in an ICN router. The other is the cost that holds the table in the memory according to a cache replacement

algorithm, e.g., prioritizing cached items. We call the former and the latter as a computational cost and a memory cost, respectively.

The computational cost includes insertion of a new caching item into the table, deletion of an existing cached item from the table, moving the location of cached items in the memory, and updating relevant information in the caching table. The operations listed above should be taken into account in the design of a cache replacement algorithm, especially when it is applied for a high speed ICN core router.

The memory cost increases as the number of cached items increases due to the increase of control information for the maintenance of the table [39, 38, 22, 19]. For example, LFU has much higher overhead to keep statistics of each cached item. In LRU using a doubly-linked list, this cost is prohibitive due to the maintenance of double pointers to other cached items.

To reduce the computational and memory costs in conventional approaches, CLOCK was introduced. CLOCK is a low-overhead approximation of LRU as mentioned in Section 4.2. We briefly describe its operation although the detail operation will be explained in Section 4.4.2. CLOCK has a circular buffer having a shape of a clock. It assigns each entry in a clock list with one-bit, which is called a reference-bit and is set whenever the entry is accessed. CLOCK searches for a cached item that needs to be replaced following a clockwise. While searching for a candidate for replacement, it refers to the reference-bit. When the bit is unset, the cached item is discarded. Otherwise, the searching process keeps on going because an entry with the bit is recently accessed. All bits skipped over during the searching process are unset. Thus, CLOCK requires only a single bit per chunk and a few repetitions of the searching process. Our proposed mechanism also adopts this mechanism of CLOCK to reduce the computational and memory costs.

4.3.3 Adaptable Parameter Tuning

Some cache replacement algorithms need to tune parameters statically or dynamically according to the access patterns of workloads. For instance, the parameters include the interval to obtain statistics of request arrivals in LFU, the ratio between the number of popular and that of non-popular cached items in LIRS, and the variable sizes of the lists used in ARC and CAR.

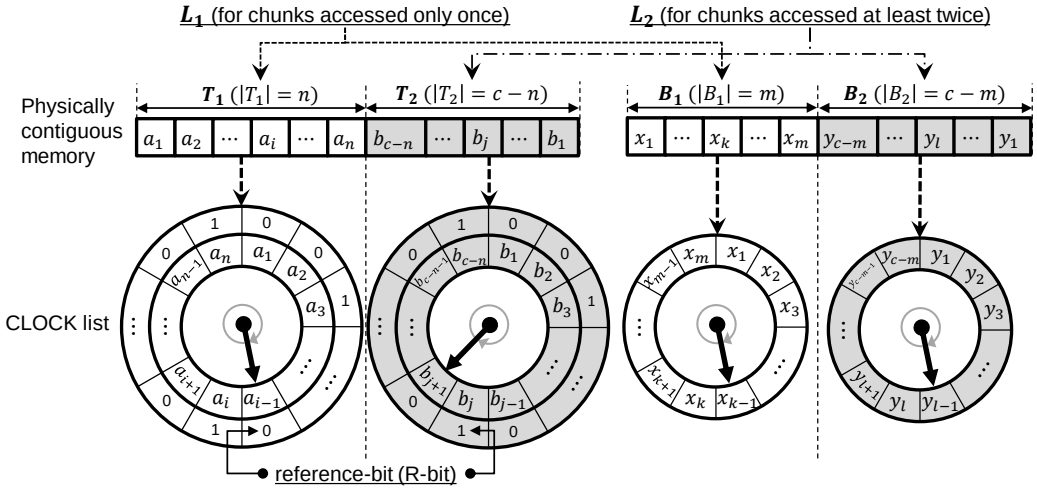


Figure 4.1: Data Structure of Compact CAR

While some parameters in ARC and CAR can be tuned adaptively to the change of access patterns, other parameters in LFU, LIRS and 2Q need to be defined in advance. However, the static parameters are unfavorable due to 1) difficulty of finding the optimal value, 2) invalidity of the optimal parameters in the change of access patterns, which causes performance fluctuation. For this reason, we conjecture that a cache replacement algorithm that adaptively changes the system parameters is preferable in the design of a cache replacement algorithm for ICN.

4.4 Compact CLOCK with Adaptive Replacement (Compact CAR)

4.4.1 Data Structure of Compact CAR

Compact CAR has two stacks, denoted by L_1 (unshaded) and L_2 (shaded) as shown in Figure 4.1. Each stack L_i is partitioned into two lists: a left list and a right list in the figure, denoted by T_i and B_i, respectively. Each list is implemented as a CLOCK list, which is known as the low-overhead LRU. T_1 consists of the entries of chunks that are initially cached. T_2 consists of the entries of chunks that have at least one cache hit as well as the entries of chunks are from B_1 and B_2. B_1 and B_2 act as "the losers bracket" providing an opportunity for discarded chunks to be re-cached. Entries in T_i hold information to point to cached chunks, and entries in B_i hold information to keep

only the record of chunks discarded from T_i (i.e., the chunks do not exist in the cache memory). The records in B_* are essential to adapt to the variety and dynamism of access patterns as explained later.

T_1 and T_2 are arranged in physically contiguous memory. B_1 and B_2 are arranged in the same manner (upper rectangles in Figure 4.1). Let T_* denote $T_1 \cup T_2$ and B_* denote $B_1 \cup B_2$ for explanation. T_* has the fixed size of c, where c denotes the number of cacheable chunks in the memory. T_1 and T_2 have n entries and $(c - n)$ entries, respectively. In the same manner, B_1 and B_2 have m entries and $(c - m)$ entries, respectively. Thus, the size of $L_1 \cup L_2$ is $2c$. In addition, we define the maximum size of L_1 as c (i.e., $n + m \leq c$). a_i, b_j, x_k, and y_l represent the entries in T_1, T_2, B_1 and B_2, respectively.

T_1, T_2, B_1 and B_2 are implemented as variable-sized CLOCK lists (lower circles in Figure 4.1). CLOCK lists T_1 and T_2 have reference-bits (R-bits). Each R-bit indicates whether the entry has been accessed, for example, R-bit is set to "1" when the chunk has been accessed, otherwise, "0". However, B_1 and B_2 do not have R-bits because they only contain the records of discarded information. Each CLOCK list has a hand which points to the first entry that the CLOCK list attempts to discard. The hand follows clockwise and moves only when it searches for a victim entry that needs to be replaced. In Figure 4.1, for example, the hand of T_1 points to a_{i-1}; therefore, T_1 attempts to discard a_{i-1} when replacement is required.

4.4.2 Operation of Compact CAR

Here, we explain the operation of Compact CAR. There are two cases when a new request arrives: (1) the requested chunk is in the cache memory or (2) the chunk is not in the cache memory. In the case (1), the entry of the requested chunk is in T_1 or T_2. If the R-bit of the corresponding entry is "0", it changes to "1"; otherwise, it remains "1". In the case (2), Compact CAR firstly retrieves the requested chunk from the source of the chunk. Then, Compact CAR verifies whether the requested chunk has been cached previously or not by checking the entries in B_1 or B_2. If the entry exists, it means it has been cached previsously but the chunk does not exist in the cache. Thus, the entry in B_1 (B_2) is discarded and added to T_2. If the entry is not found in B_1 and B_2, it means the requested

chunk has not been cached recently. Thus, a new entry for the chunk is added to T_1.

Then, we describe how the new entry is added to T_1 by discarding an existing entry in T_1. Compact CAR searches for an entry to be discarded with a hand operation. Suppose that the hand currently points to the entry a_{i-1} as shown in Figure 4.1. Because the R-bit of a_{i-1} is "0" (hereafter abbreviated to $R = 0$), which indicates an evictable entry that is not recently requested, the hand discards the entry a_{i-1}. The discard event triggers following processes: First, the chunk corresponding to the entry a_{i-1} is discarded, and then the entry itself moves to B_1. If B_1 is full, the entry x_{k-1}, which is currently pointed to by the hand in B_1, is discarded for the entry a_{i-1}. Then, the hands moves to the next entry (i.e., the hand in T_1 moves to a_i and the hand in B_1 moves to x_k).

Now, let us consider the other case when the entry initially pointed to by the hand in T_1 has a R-bit of "1", which indicates a recently re-requested chunk. The entry with $R = 1$ is removed from T_1 and inserted to T_2. In this manner, the size of T_1 is reduced by one and at the same time that of T_2 increases by one. This is why the sizes of the CLOCK lists are variable. We will elaborate this operation in detail in the following section. At this point, we have not found an entry with $R = 0$, which needs to be discarded. Thus, the hand keep moving to search for an entry with $R = 0$. Once the entry is found, all necessary procedures described in the previous paragraph are carried out.

Until this point, we explain the case where an entry in T_1 is discarded. However, the entry to be discarded can be selected from T_2 as well. This also changes the sizes of T_1 and T_2, which enables Compact CAR to adapt to traffic access patterns dynamically. The sizes of T_1 and T_2 influence the caching behavior of Compact CAR: When the size of T_1 increases, the number of chunks that have been accessed only once increases. In other words, the operation behavior of Compact CAR becomes suitable for the case where recently accessed content are important. This behavior enables Compact CAR to deal with the volatile popularity by removing outdated chunks quickly. On the other hand, when the size of T_2 increases, the number of chunks that have been accessed at least twice increases as well. It means Compact CAR becomes suitable for the case where frequently requested content are important. This behavior enables Compact CAR to deal with SCAN by removing one-time used chunks quickly.

To adaptively control the sizes of T_1 and T_2, Compact CAR defines the target size for T_1. The target size for T_1 is represented as p ($0 < p \leq c$). When the current size of T_1, which is n, is

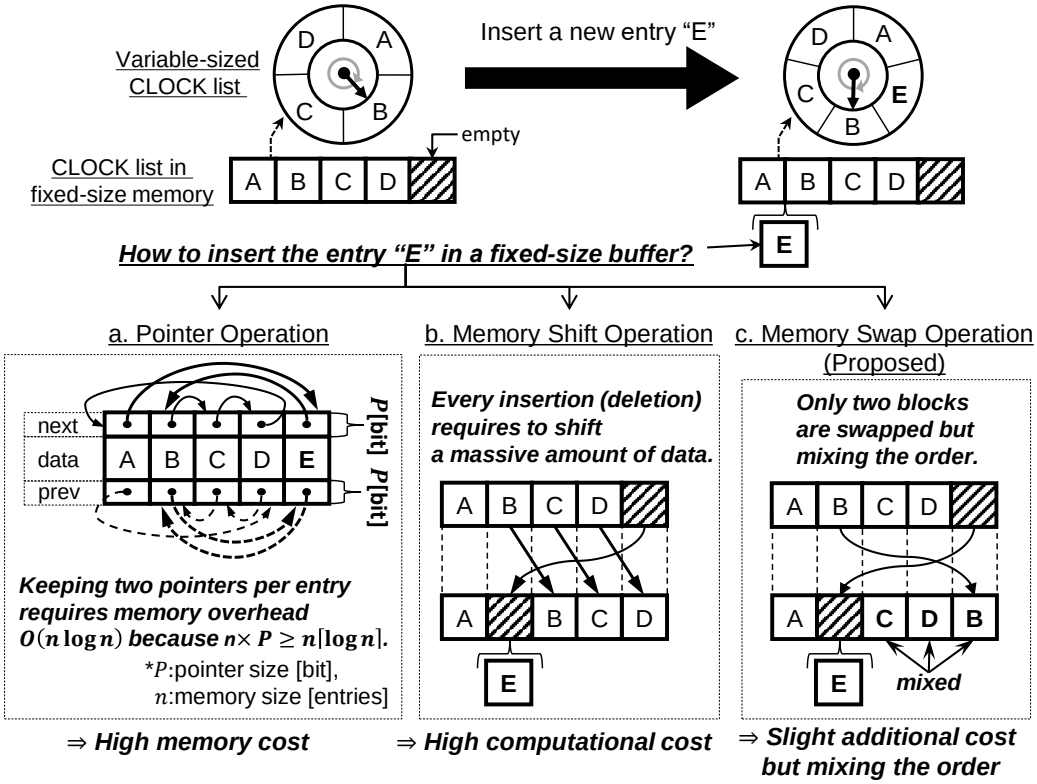


Figure 4.2: Illustration of Computational and Memory Costs in the Inserting Operation in the Different Data Structures

larger or equal to the target size p ($n \geq p$), an entry in T_1 is discarded; otherwise, an entry in T_2 is discarded. By adjusting p, the sizes of the CLOCK lists of T_1 and T_2 vary adaptively. In summary, the target size p governs the behavior of Compact CAR. We will explain how the target size p is dynamically adapted according to traffic access patterns in Section 4.4.4.

4.4.3 Design of Low-overhead Variable-sized CLOCK List

As mentioned in the previous section, Compact CAR varies each size of CLOCK list to adapt the change in traffic access patterns. A typical CLOCK list, whose size is fixed, is known as low-overhead because it simply replaces an old entry with new one, which does not change its size. However, the variable-sized CLOCK list in Compact CAR is costly because it changes its size to insert or delete an entry. For instance, when an entry of a new chunk needs to be inserted in T_1,

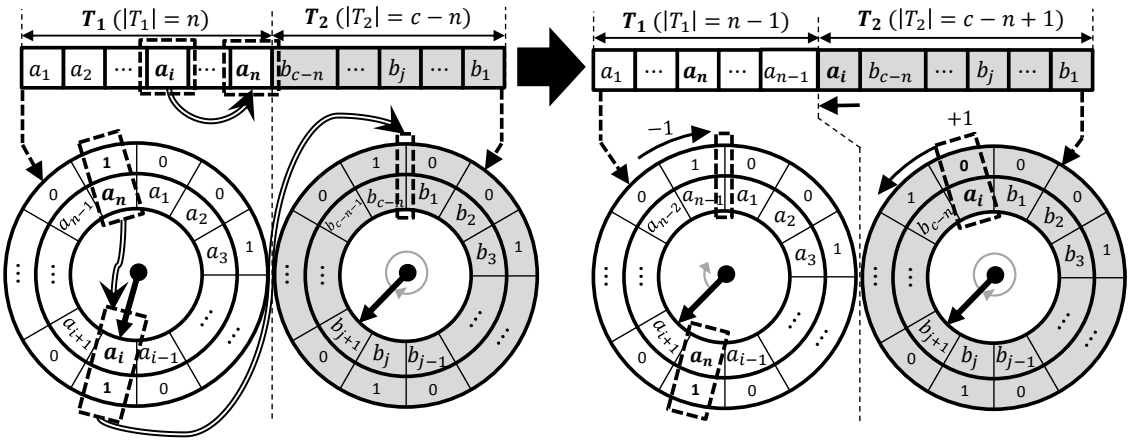


Figure 4.3: Example of Moving a Chunk a_i from T_1 to T_2 by Replacing the Edge Chunk a_n

the hand keeps moving to search for an entry with $R = 0$, which will be replaced with the new entry. When the hand encounters an entry with $R = 1$, the entry is removed from T_1 and inserted to T_2. This operation is expensive from the view point of computational and memory costs because it varies the size of CLOCK lists.

To understand the cost of a variable-sized CLOCK list, Figure 4.2 illustrates the operation of entry insertion in a CLOCK list. Initially, as an example, the CLOCK list is capable of holding five entries and only four entries are currently occupied, which are A, B, C, and D. These entries are stored in fixed-size memory as shown bottom of the CLOCK list in the figure. The hand points to the entry B.

Consider what happens when a new entry E is inserted to the CLOCK list. The entry E is supposed to be inserted between A and B because the position is farthest from the hand. However, we must make a space for E in the physically contiguous memory, which currently four entries occupy. In the bottom of Figure 4.2, we show three different approaches to insert the entry into the memory from the viewpoint of computational and memory costs.

First, the most left case (a) illustrates a pointer operation with a doubly-linked list. The doubly-linked list introduces additional pointers which manage the order of entries in a CLOCK list. When a chunk is inserted in the middle of memory space, the chunk is inserted physically at the end

of the memory space. Then, the order of the chunks in the memory is arranged virtually using the doubly-linked list. It involves two operational costs: computational cost which involves the rearrangement of pointers in the doubly-linked list, and memory cost which involves the memory space accommodating the doubly-linked list. Computational cost is not that expensive. However, it consumes a decent amount of memory space to maintain the order by keeping two pointers per entry (see also Section 4.6.1 for detail). The original CAR algorithm [38] adopts the pointer operation to insert a new entry.

Second, the case (b) illustrates a case of a memory shift operation. In this case, a doubly-linked list is not used but memory blocks are shifted when an entry is inserted. It does not require high memory cost to maintain pointers because the order of entries in a CLOCK list is realized in physical memory directly without a doubly-linked list in the first scenario. However, this scenario introduces high computational cost caused by the shift of memory blocks (see also Section 4.5.7 for more detail).

Third, the case (c) illustrates the operation of entry insertion in Compact CAR. To insert a new entry at the position of the entry 'B', Compact CAR moves the entry 'B' to the end. Then, the new entry 'E' is inserted to the location. The difference between our proposal and the second operation is that the new entry 'E' and the old entry 'B' are swapped rather than shifting all entries. In this manner, the computational cost can be reduced. Furthermore, it does not use a doubly-linked list to create virtual order of entries in the memory space and so the memory cost can be reduced as well.

It may be concerned that the operation changes the order of recency, which may degrade the cache performance of the proposal: the operation shortens a lifetime of a popular entry or vise versa. If "B" is unpopular in this example, the unpopular entry will undesirably occupy the space for a longer time. Thus, the operation mixing the order of a CLOCK lists may make its behavior close to random replacement. However, the influence is not that serious: we will address the issue in Section 4.5.

Figure 4.3 gives an example of moving an entry a_i within T_* to realize the swap operation illustrated in Fig. 4.2(c). Remember the example explained in Section 4.4.2, where an entry a_i with $R = 1$ is moved from T_1 to T_2. We realize the memory swap operation by exploiting the constancy of the size of T_* when the sizes of T_1 and T_2 change. Compact CAR swaps a_i with the entry a_n at

the boundary between T_1 and T_2 in the physically contiguous memory. Then, Compact CAR just shifts the boundary leftward to make a_i be in T_2. This simple swap operation enables the movement of an entry between T_1 and T_2 without computational and memory costs.

4.4.4 Replacement Algorithm of Compact CAR

Algorithms 1, 2, 3, and 4 show pseudocode of the cache replacement algorithm of Compact CAR. The replacement process starts with Algorithm 1. There are three cases when a new request arrives for a chunk whose entry is represented as x: (1) x is in T_*, (2) x is in B_*, (3) x is neither in T_* nor in B_*. The case (1) occurs on a cache hit. The cases (2) and (3) occurs on a cache miss. In the case (1), the process sets the R-bit of x to "1" and terminates (lines 2–4).

In the case (2), first, the process discards x from B_* and updates the parameter p, which represents the target size for T_1 as mentioned in Section 4.4.2 (lines 7–13). At line 6, i stands for the index of the list T_i into which x is to be cached; therefore, i is set to 2 as explained in Section 4.4.2. The process of tuning p is important to make Compact CAR adaptive to changes in access patterns as discussed in [46, 38]. This process needs to determine (a) whether to increase or decrease p, and (b) the amount of increase or decrease in p.

In respect to (a), when x is in B_1, p increases; otherwise, p decreases. As described in Section 4.4.2, T_1 and T_2 are devised to effectively cache recently accessed chunks and frequently accessed chunks, respectively. Since B_1 and B_2 correspond to T_1 and T_2, respectively, we can adaptively control the behavior of Compact CAR by changing p according to accesses to B_1 and B_2. The tuning process increases p when x is in B_1 because the access indicates recently accessed chunks are becoming important. On the other hand, when x is in B_2, p decreases to place importance on frequency.

In respect to (b), we determine the amount of increase (decrease) δ according to the ratio of $|B_1|$ to $|B_2|$, where $|B_i|$ represents the size of B_i, to adapt the changes of access patterns rapidly. To explain the reason, let us consider the case where there is an access to B_1 when $|B_2|$ is larger than $|B_1|$. Intuitively, the fact that $|B_2| > |B_1|$ indicates frequently accessed chunks were more important than recently accessed chunks until now; however, the current access to B_1 indicates

recently accessed chunks are becoming important. Thus, when $|B_2| > |B_1|$, the tuning process sets $\delta = |B_2|/|B_1|$ to rapidly adapt the access patterns where recency is more important; otherwise, $\delta = 1$.

In the case (3), line 15 sets i to 1 to cache the new entry x in T_1. Lines 16–18 ensure that there is a room in B_* because an existing entry in T_1 (T_2) is replaced by x and is moved to B_1 (B_2) at lines 21–23. If the lists in Compact CAR are not full (i.e., $|L_1| < c$ and $|L_1 \cup L_2| < 2c$ as defined in Section 4.4.1), the entry is simply inserted into B_*. If L_1 is full (i.e., $|L_1| = c$), the replacement process discards an entry in B_1 to insert the entry. If L_1 is not full and $L_1 \cup L_2$ is full (i.e., $|L_1 \cup L_2| = 2c$), the process discards an entry in B_2.

After that, lines 20–27 cache the new entry x in T_i. If T_* is not full, x is simply inserted to T_i (line 25). Otherwise, the process replaces an entry in T_* with x (lines 20–23). The victim entry is selected from T_1 if the size of T_1 is not less than the target size p; otherwise, the entry in T_2 is replaced. The discarded entry can move to B_* because we have ensured that there is room in B_* (lines 16–18). Finally, x is cached at a position s_t in T_i, which has been ensured to be available (line 27).

Algorithms 2, 3, and 4 describe how to make a room for a new entry. The location pointed to by the hand of T_i is represented as Hand_{T_i}, and B_i is analogous. The `DiscardBottom` procedure (shown in Algorithm 2) discards the entry x in B_i. The `ReplaceBottom` procedure (shown in Algorithm 3) discards an entry pointed to by the hand of B_i in the case (3) above. The `ReplaceTop` procedure (shown in Algorithm 4) removes an entry pointed to by the hand of T_i when the cache is full. If this procedure finds an entry with $R = 1$ in T_1, it moves the entry to T_2 in the manner described in Fig. 4.3. As shown in the figure, when the entry a_i needs to be moved to the other list, the operation swaps a_i with the entry a_n at the boundary between the lists. $EdgeEntry$ denotes the entry located at the boundary and $EdgeAddres$ denotes its address in Algorithms 2, 3, and 4.

4.5 Performance Evaluation

In this section, we evaluate the performance of Compact CAR compared to OPT, FIFO, CLOCK, and CAR in various scenarios to demonstrate the fulfillment of the design considerations discussed

Algorithm 3 Compact CAR Replacement Algorithm

procedure CACHEREPLACEMENT(x) ▷ x is an accessed entry.
  if $x \in T_*$ **then** ▷ Cache hit
    x.R-bit$\leftarrow 1$
    return
  else if $x \in B_*$ **then** ▷ Record of discard remains
    $i \leftarrow 2$ ▷ To cache x in T_2
    if $x \in B_1$ **then**
      $\delta \leftarrow \max(1, \frac{|B_2|}{|B_1|})$; $p \leftarrow \min(c, p + \delta)$
      DiscardBottom(1,x)
    else ▷ $x \in B_2$
      $\delta \leftarrow \max(1, \frac{|B_1|}{|B_2|})$; $p \leftarrow \max(0, p - \delta)$
      DiscardBottom(2,x)
    end if
  else ▷ Cache miss
    $i \leftarrow 1$ ▷ To cache x in T_1
    if $\mathrm{Full}(L_1)$ & $|B_1| > 0$ **then** $\mathrm{ReplaceBottom}(1)$
    else if $\mathrm{Full}(L_1 \cup L_2)$ & $|B_2| > 0$ **then** $\mathrm{ReplaceBottom}(2)$
    end if
  end if
  if $\mathrm{Full}(T_*)$ **then**
    if $|T_1| \geq \max(p, 1)$ **then** $s_t \leftarrow \mathrm{ReplaceTop}(1)$
    else $s_t \leftarrow \mathrm{ReplaceTop}(2)$
    end if
  else ▷ T_* is not full.
    $s_t \leftarrow$ an available address in T_i
  end if
  $T_i[s_t] \leftarrow x$ ▷ x is cached as an entry in T_i.
end procedure

Algorithm 4 DiscardBottom() for Compact CAR

procedure DISCARDBOTTOM(i, x)
  Swap(x, B_i.EdgeEntry)
  Discard x (at the edge of B_i) ▷ Ensuring that the address next to the edge of B_i is free
end procedure

previously. OPT is Belady's algorithm [55]: off-line optimal algorithm with a priori knowledge of the stream of requests. OPT shows absolute upper bound on the achievable cache hit rate.

Algorithm 5 ReplaceBottom() for Compact CAR

```
1: procedure REPLACEBOTTOM(i)
2:     Swap(B_i[Hand_{B_i}], B_i.EdgeEntry)
3:     Discard B_i.EdgeEntry
4:     Rotate Hand_{B_i}          ▷ Ensuring that the address next to the edge of B_i is free
5: end procedure
```

Algorithm 6 ReplaceTop() for Compact CAR

```
1: function REPLACETOP(i)
2:     while T_i[Hand_{T_i}].R-bit= 1 do
3:         T_i[Hand_{T_i}].R-bit← 0
4:         if i=1 then
5:             Swap(T_i[Hand_{T_i}], T_i.EdgeEntry)
6:                                    ▷ Shift the boundary between T_1 and T_2.
7:         end if
8:         Rotate Hand_{T_i}
9:     end while
10:    s_e ← an address next to the edge of B_i
11:    B_i[s_e] ← T_i[Hand_{T_i}]
12:    Swap(T_i[Hand_{T_i}], T_i.EdgeEntry)
13:    Discard T_i.EdgeEntry
14:    Rotate Hand_{T_i}
15:    return T_i.EdgeAddr
16: end function
```

First, the performance of the proposed algorithm is evaluated with various access patterns including synthetic traffic as well as real traffic trace. We investigate the cache performance of ideal-cooperative caching and non-cooperative caching by using two topologies: a one-node topology and a line topology. Then, the adaptability of our proposal to changing access traffic patterns is demonstrated by comparing to the same approach without tuning a parameter. Finally, the computational and memory costs of our proposal are theoretically analyzed to present its efficient memory usage which is critical in the design of a high performance ICN core router.

4.5.1 Simulation Setup and Configuration

Two types of workloads are used in this simulation study: artificial workloads that follow a Zipf distribution and real traffic traces of Video-on-Demand (VoD), e.g., YouTube, DailyMotion, and

NicoVideo, which are collected from a network gateway at Osaka University campus. The former and the latter are denoted by $A_{Zipf(\alpha)}$ and by A_{Real}, respectively. In addition, their superscript C and P, e.g., $A^{C}_{Zipf(a)}$ and $A^{P}_{Zipf(\alpha)}$ represent the workloads in units of content and chunks, respectively.

The popularity of Internet content (e.g., VoD, web pages, file sharing, and user generated traffic) has been reported to follow the Zipf distribution with $0.6 \leq \alpha \leq 1.2$ [40, 37]. Thus, we use these values to generate synthetic traffic requests from the Zipf distribution for this simulation study.

To justify the results using synthetic traffic, we also use the real traffic traces. The traces are gathered from July 26th 2013 to February 26th 2015. The number of unique contents is 2,428,880; the number of contents requested at least twice is 918,545; and the number of total accesses is 13,004,868. The popularity distribution of the real traffic trace follows the Zipf-like distribution, as depicted in Fig. 4.4. We also show the statistics of the real traffic traces in units of chunks in Table 4.2.

As stated in Section 4.3.1, the fine granularity of chunks in ICN, namely chunks or segments, changes the access patterns of request message, which dramatically governs the performance of cache replacement algorithm. Unfortunately, ICN traffic traces are not available yet. Thus, we generate synthetic requests for chunks, which simulates the access pattern of ICN in the following manner. We determines the inter-arrival time between requests to content according to our observed real trace. On the other hand, the inter-arrival time for chunks is constant according to Table 4.1, which is determined by the statistics of our observed real traffic. The generated requests are superimposed to simulate the aggregation of request messages in the network.

This simulation studies the cache performance of ideal-cooperative caching and non-cooperative caching[1] by using two topologies: the one-node topology and the line topology. One is the topology in which there is only one ICN router between clients and a server. The other is the line topology that includes ten ICN routers between them. As mentioned previously, the cache hit rate is governed by two factors: one is a cache replacement algorithm (how to cache), and the other is a cooperative

[1] A cooperative caching algorithm distributes chunks in the network to improve cache hits as well as to reduce the usage of network resources.

Table 4.1: Number of Chunks Per Second $[\mathrm{pck/s}]$

Chunk size	1.5 KB	15 KB	60 KB
Standard Definition (600kbps)	50	5	1.25
High Definition (1.2Mbps)	100	10	2.50

caching algorithm (where to cache). Because the purpose of our simulation is to evaluate the performance of the former, we do not investigate the performance of the latter in detail but rather we consider its best and worst case performance to make the analysis complete.

For explanation, we start to discuss the worst case. The line topology represents the worst case without any cooperation. This is because the cache capacity of a whole network is considerably wasted by redundant caches, especially when every nodes in the line topology attempt to cache contents being downloaded from one end to the other. Thus, the simulation using the line topology clarifies the lower bound of the performance caused by cooperative caching mechanisms. On the other hand, we can assume that the one-node topology represents the best case, where a cooperative caching algorithm works ideally. If the caching capacity of one node is equivalent to the total n nodes, the one-node topology can be considered as the n-node topology that has an ideal cooperative caching mechanism. In other words, the result with the one-node topology shows the upper bound of the performance where cooperative caching works ideally. Thus, the results of our simulation using the two types of topologies clarify the upper and lower bound of the performance caused by cooperative caching mechanisms.

Each cache at ICN router has same capacity c ranging from 10^1 to 10^6 chunks which are adjusted according to the traffic trace we adopt. Also, the transmission delay of each chunk on links and the unnecessary computation in the protocol stacks are ignored to simplify the simulation.

4.5.2 Cache Hit Rate with a SCAN Access Pattern using Synthetic Traffic

Figure 4.5 depicts the cache hit rate of each cache replacement policy in the one-node topology with synthetic traffic described previously: $A^C_{\mathrm{Zipf}(\alpha)}$ changing α from 0.6 to 1.2. Our proposal, Compact CAR, achieves a hit rate comparable to that of CAR, which is contrary to our speculation. We conjectured that the operation mixing the order in the Compact CAR would degrade its performance

Table 4.2: Statistics of Workloads in Units of Chunks

Workload	# of total accesses	# of observed unique chunks	# of chunks requested at least twice
$A_{\text{Real}}^{P(1.5\text{KB})}$	17,955,409	5,465,044	440,254
$A_{\text{Real}}^{P(15\text{KB})}$	14,557,548	5,321,617	552,631
$A_{\text{Real}}^{P(60\text{KB})}$	16,606,810	8,006,084	1,769,759

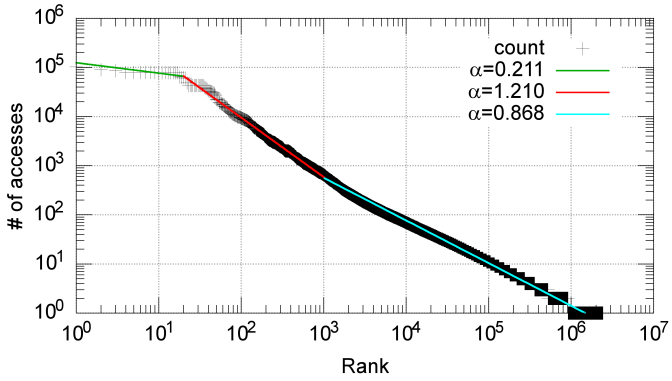
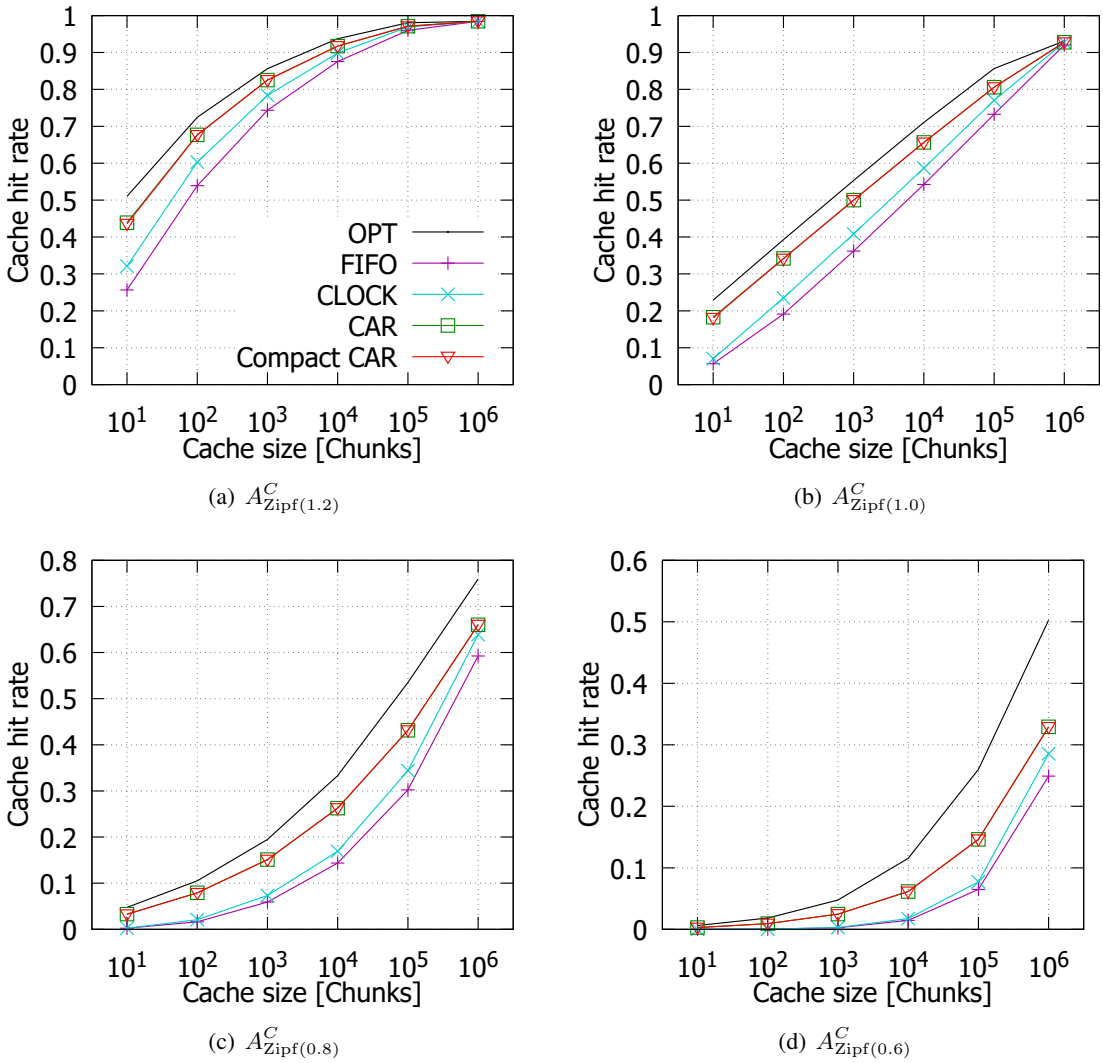


Figure 4.4: Popularity Distribution of Real Trace

because the operation makes its behavior close to random replacement. The result is promising because we can achieve the performance as good as CAR even with much less memory cost. The memory cost of Compact CAR including several others is theoretically analyzed in Section 4.5.6 in detail. In addition, the results show that Compact CAR can achieve the same cache hit ratio with one-tenth of cache size compared to simple cache replacement algorithms such as FIFO and CLOCK in the best case. This is because the two-stack approach of Compact CAR prevent popular content from being removed from the cache by SCAN.

In addition to the simulation using traces in units of contents, Figure 4.6 shows the cases when the sizes of chunks change from 60 KB to 1.5 KB with the parameters of the Zipf distribution α at 1.0 and 1.2, which are denoted by, e.g., $A_{Zipf(0.6)}^{P(60KB)}$ to $A_{Zipf(0.6)}^{P(1.5KB)}$. As the value of α increases,

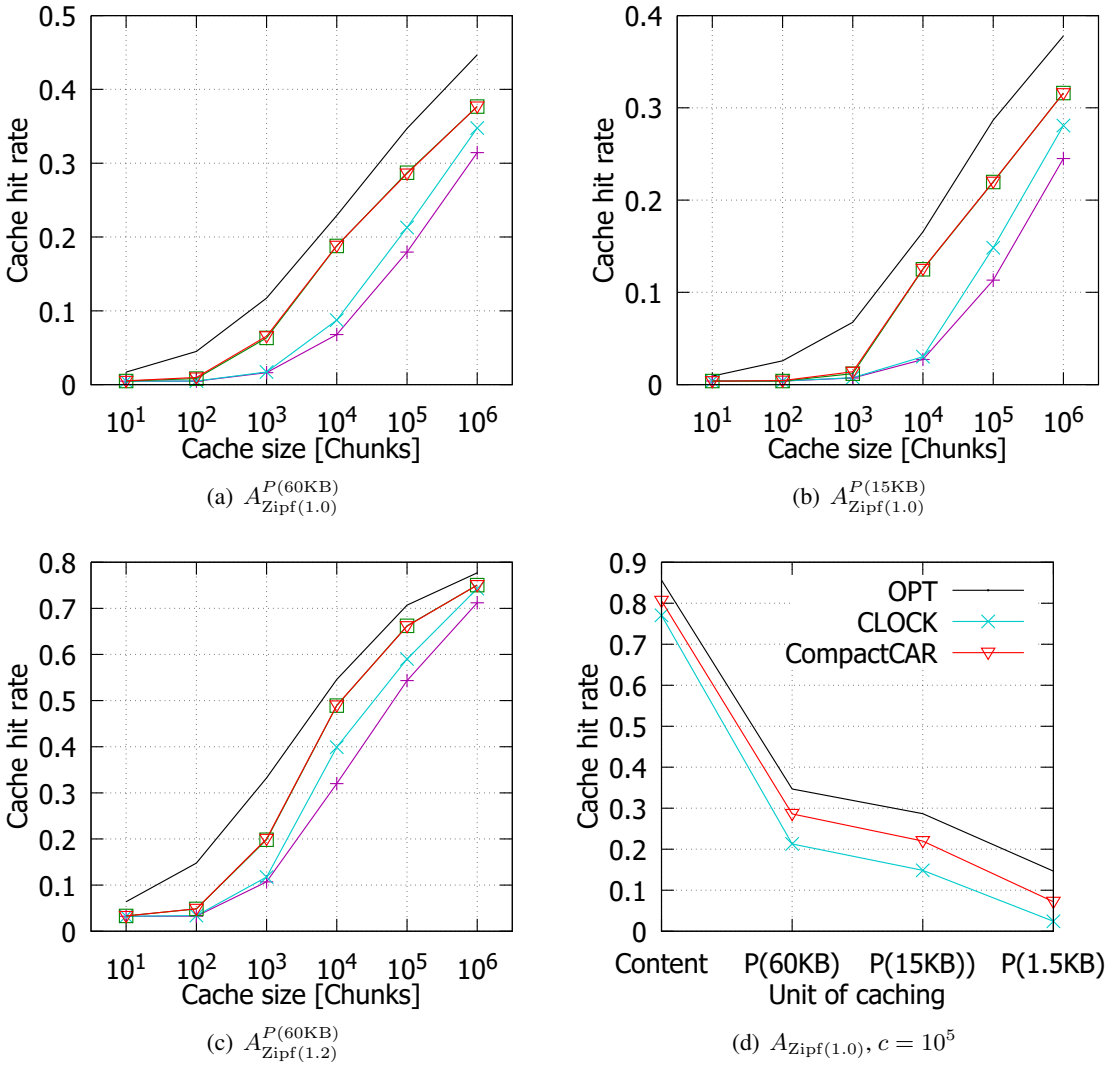


(a) $A^{C}_{\mathrm{Zipf}(1.2)}$ (b) $A^{C}_{\mathrm{Zipf}(1.0)}$

(c) $A^{C}_{\mathrm{Zipf}(0.8)}$ (d) $A^{C}_{\mathrm{Zipf}(0.6)}$

Figure 4.5: Results for Synthetic Traffic in Units of Content

the hit rate increases. This means that a high popularity bias results in a high hit rate as known in the previous studies. As depicted in Fig.4.6(d), we observe that the cache hit rate decreases substantially as the size of chunks becomes small, e.g., from a whole content to chunks.



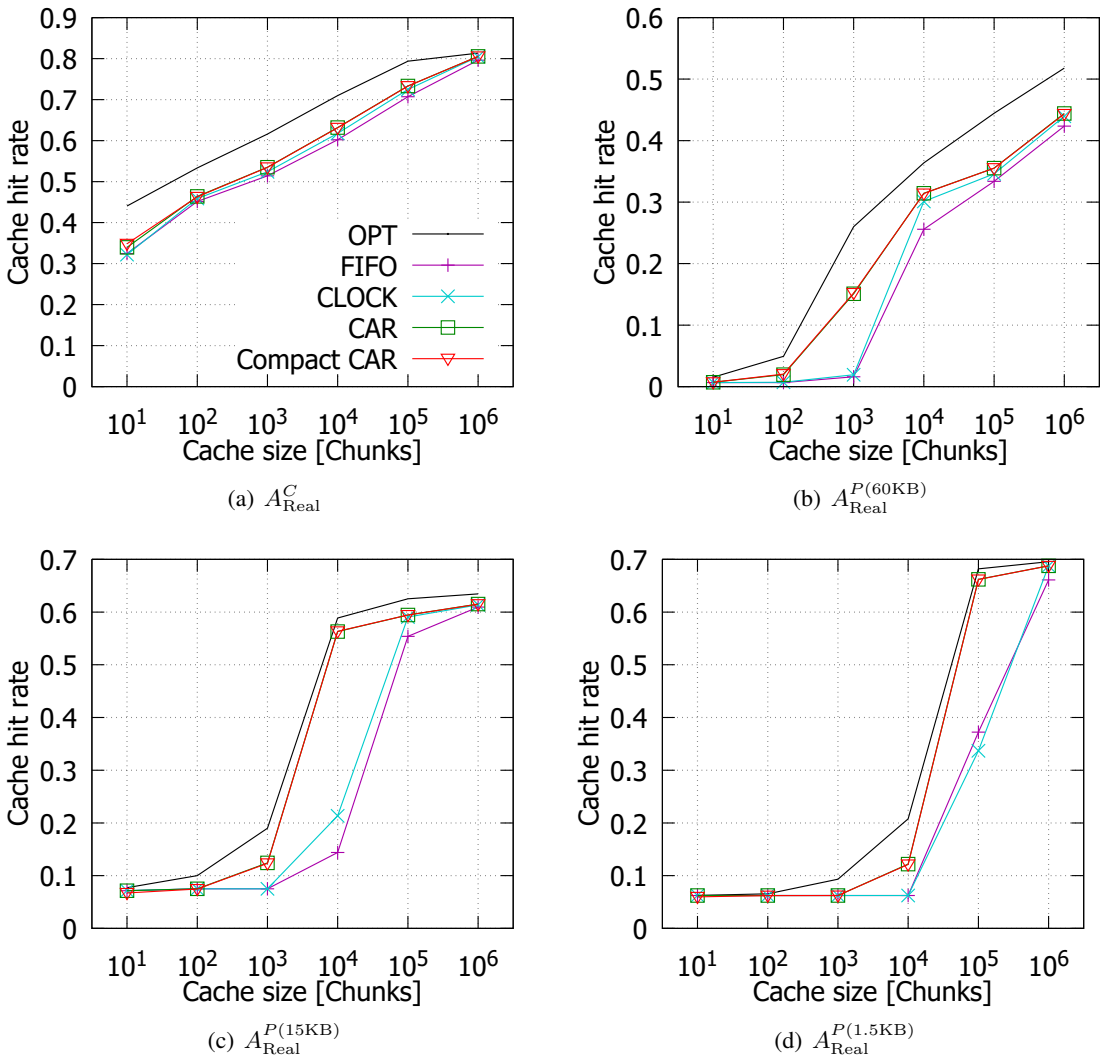
(a) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ (b) $A^{P(15\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$

(c) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.2)}$ (d) $A_{\mathrm{Zipf}(1.0)}$, $c = 10^5$

Figure 4.6: Results for Synthetic Traffic in Units of Chunks

4.5.3 Cache Hit Rate with a SCAN Access Pattern using Real Traffic Trace

Figure 4.7 presents the simulation results in the one-node topology with real Video-on-demand (VoD) traffic which was collected at Osaka University. Because the original traces are in units of content, we divide each content into small sized chunks to simulate the traces in units of chunks in ICN networks. The cache hit rate in Fig. 4.7 are similar to those in Fig. 4.5 and Fig. 4.6. In Fig. 4.7, one interesting observation is that the cache hit rate of our proposed algorithm suddenly soars,



(a) A^{C}_{Real} (b) $A^{P(60\mathrm{KB})}_{\mathrm{Real}}$

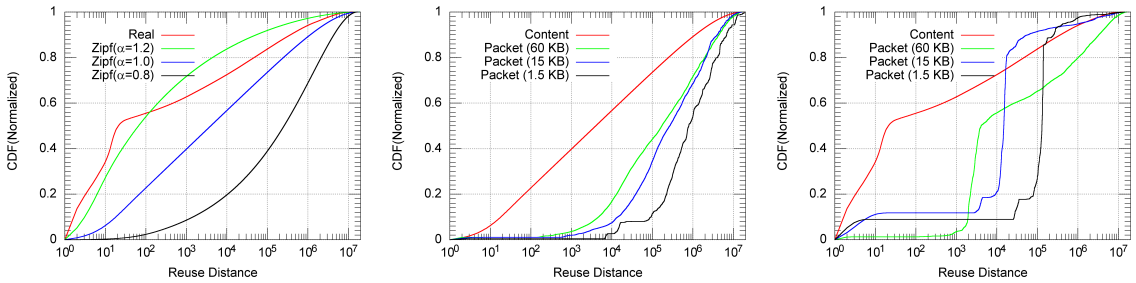
(c) $A^{P(15\mathrm{KB})}_{\mathrm{Real}}$ (d) $A^{P(1.5\mathrm{KB})}_{\mathrm{Real}}$

Figure 4.7: Results for Real Traffic Trace

e.g., when cache size is 10^4 in Fig. 4.7(c) compared to conventional cache replacement algorithms: the performance becomes outstanding. This phenomenon correlates to the Reuse Distance (RD); therefore, we discuss it below.

Figure 4.8 plots the cumulative distribution functions (CDFs) of RD. RD represents the number of requests between two consequtive requests for the same chunk. For example, consider what happens when the value of RD is larger than the size[2] of cache. The cache of the chunk requested

[2]Its unit is the number of chunks



(a) Different Types of Workloads in Units of Content

(b) Artificial Workloads ($\alpha = 1.0$) in Different Units

(c) Real Traces in Different Units

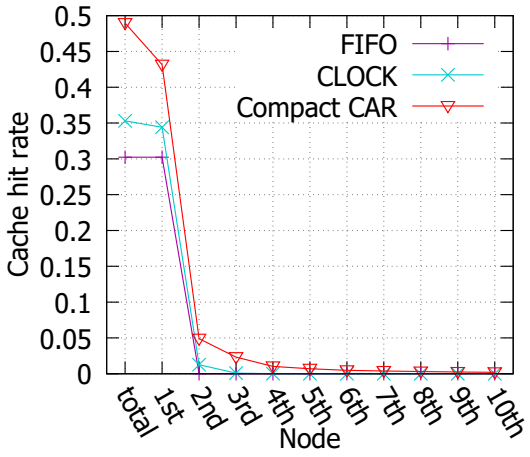
Figure 4.8: CDF of RD in Various Workloads

by the first request has a high probability to be discarded from the cache before the second request arrives. If this case keeps happening due to a large amount of one-time contents (e.g., SCAN), only non-popular contents remain in the cache. This situation is called cache pollution that non-popular contents occupy whole cache causing low-cache hit rate. Thus, a cache hit almost occurs when RD is smaller than cache size, and vice versa.

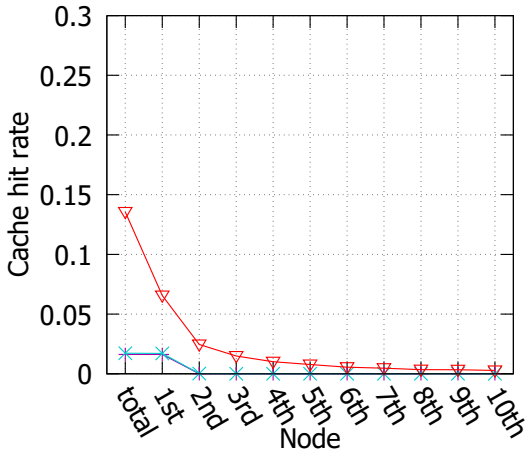
As mentioned in Section 4.4, Compact CAR maintains two link lists: one for non-popular contents, and the other for popular contents. Thus, the cache pollution only affects to the link list that maintains non-popular contents. In other words, Compact CAR is robust to the cache pollution scenario caused by a large amount of non-popular contents.

4.5.4 Simulation with the Line Topology

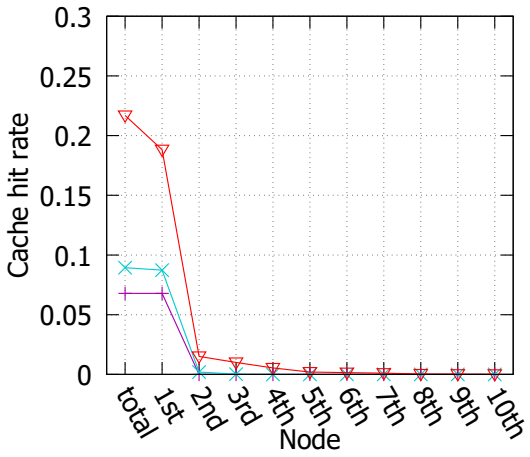
Figure 4.9 presents the cache hit rate of individual nodes on the line topology. We omit the graph of CAR because the hit rate of CAR are almost identical to Compact CAR. Compact CAR improves the hit rate in the second and succeeding routers, whereas the hit rate of FIFO and CLOCK decreased to approximately zero. Figure 4.10 shows the upper and lower bound of the performance achieved by cooperative caching. The performance of ideally cooperative caching is denoted by "ideal-coop", which specifies the upper bound. The result denoted by "non-coop" means the total cache hit rate of nodes in the line topology, which is the performance of non-cooperative caching and specifies the lower bound. We also show the hit rate of the only first node of the line topology as "1st-node" to understand how CLOCK is inappropriate for the environment without cooperative



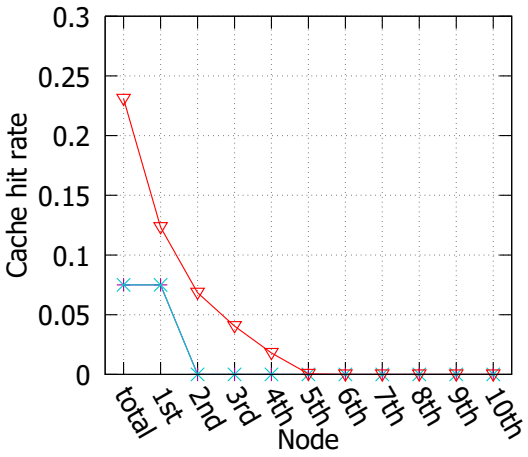
(a) $A^{C}_{\mathrm{Zipf}(0.8)}$ $(c = 10^5)$



(b) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ $(c = 10^3)$



(c) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ $(c = 10^4)$

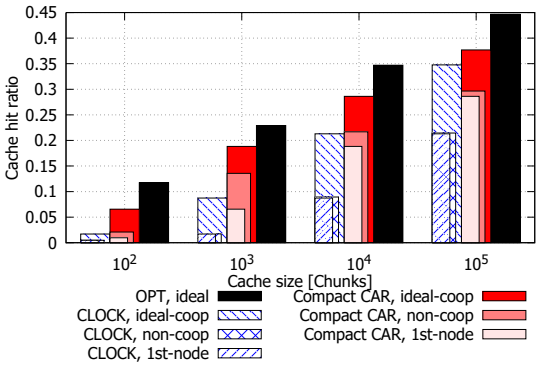


(d) $A^{P(15\mathrm{KB})}_{\mathrm{Real}}$ $(c = 10^3)$

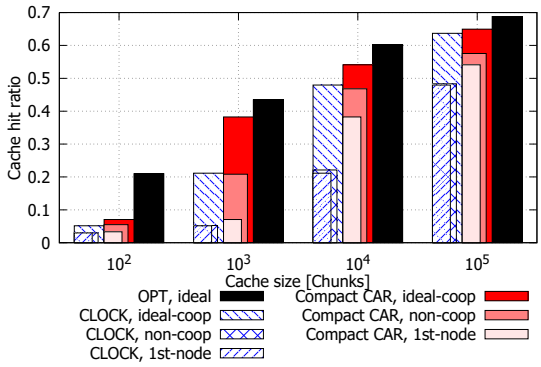
Figure 4.9: Results for Simulation with the Line Topology

caching. There is less difference between the upper bound and the lower bound of Compact CAR than that of CLOCK. This result indicates Compact CAR can exploit resources in a network by reducing redundant caches caused by the cooperation failure.

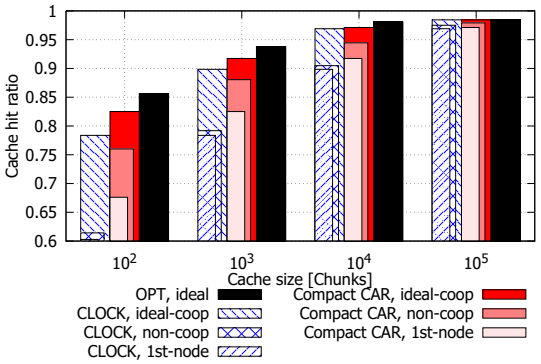
It is interesting to analyze the performance under an environment with a certain cooperation or a cache decision algorithm; however, we do not show the analysis because the main purpose of this chapter is proposing the cache replacement algorithm that is feasible and appropriate for an ICN



(a) $A^{P(15\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$



(b) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.2)}$



(c) A^{C}_{Real}

Figure 4.10: Comparison between Non-cooperative Caching and Ideally-cooperative Caching

router. In future, we will investigate the effects of various cache placement and decision algorithms on a network and communication quality.

4.5.5 Dynamic Parameter Tuning

As explained in Section 4.4.3, Compact CAR dynamically adapts to changing traffic access patterns by varying the parameter p. There is no one-size-fits-all parameter and it is necessary that the parameter should be tuned to maximize cache hit rate under any circumstances.

Here we evaluate the parameter tuning strategy for the proposed Compact CAR whose parameter p represents the target size for T_1. The parameter p ranges from zero to the cache size c. As the value of p increases, the operational behavior of Compact CAR becomes similar to the case where

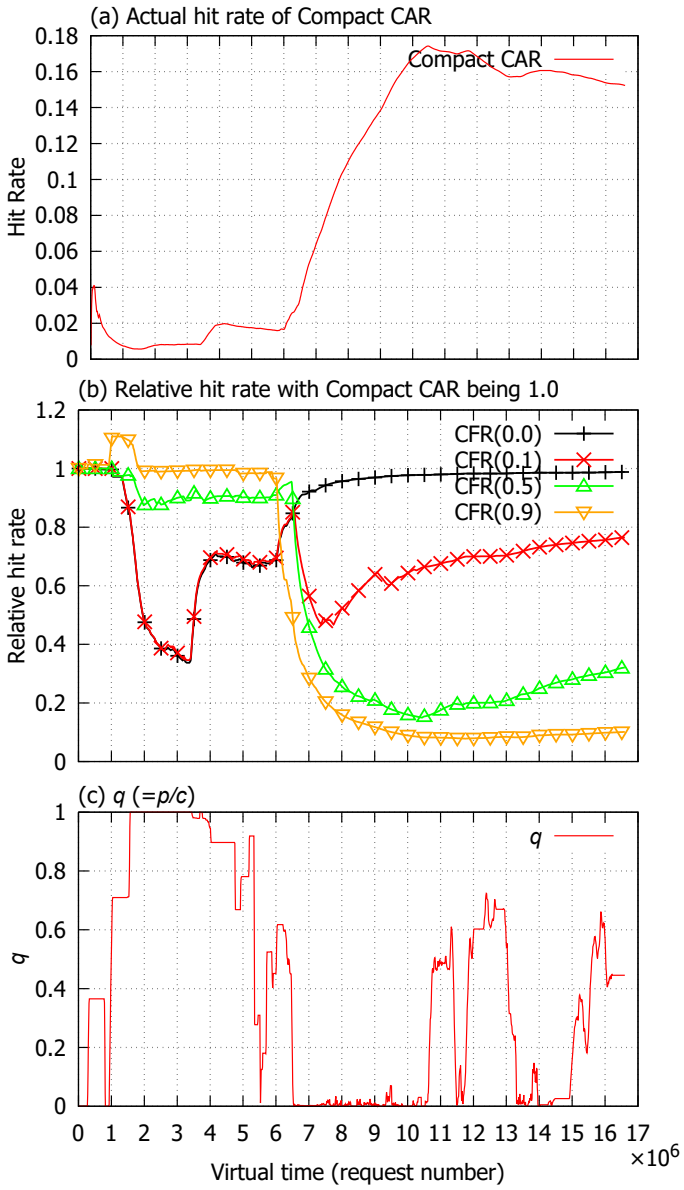


Figure 4.11: Dynamics of Hit Rate of CFR(q) and Adaptive Parameter q

recently accessed content becomes important. On the other hand, as p decreases, Compact CAR behaves similar to the case where frequently requested content becomes important.

Thus, depending on the variation of access patterns, the parameter p should be tuned. To compare the difference between dynamical tuning and statical tuning, we introduce Clock with Fixed

Replacement (CFR) algorithm which corresponds to our proposal Compact Clock with Adaptive Replacement (Compact CAR). CFR has the fixed value of $q = p/c$ ($0 \leq q \leq 1$) which is determined in advance.

Figure 4.11 shows that Compact CAR adaptively changes the parameter: the trends of q and the cache hit rate of CFR(q). The x-axis shows the virtual time t, which is equivalent to the total number of requests. The cache hit rate of CFR(q) are shown as relative value with that of Compact CAR being 1.0 in Fig. 4.11 (b). When $0 < t < 6 \times 10^6$, CFR(q) with high q achieves the high hit rate, and vice versa. When $t = 6 \times 10^6$, we can observe the rapid increase in the cache hit rate of Compact CAR. This increase is due to an arrival of many popular contents. Thus, the value of q decreases to adopt the access patterns, where frequently accessed content becomes important, and the corresponding hit rate of CFR(q) increases. The results show that Compact CAR can adaptively change the parameter. In addition, q of Compact CAR continues to follow the optimal value at any time as evidenced by the fact that the best relative hit rate among CFR(q) are at most nearly 1.0. By contrast, the relative hit rate of the parameter fixed algorithms become at worst nearly 0.1. Thus, we can confirm that the parameter tuning algorithm of Compact CAR are necessary and greatly adaptive.

4.5.6 Analysis on Space and Time Complexities of CAR and Compact CAR

We analyze the time and space complexity of Compact CAR. The complexity is analyzed from the viewpoint of an additional process or memory required for the algorithms. In the evaluation of time complexity, we calculate the number of memory access as a dominant factor when a cache hit or a miss occurs. Because the actual value is typically unsteady, we study the worst-case and average-case complexity in the two different cases (i.e., a cache hit and a cache miss). Space complexity depends on the amount of additional bits needed to maintain a data structure, and so we calculate the amount of bits. We also express them with big O notation. Our analysis does not calculate the amount of memory to keep records of discarded chunks since it should be compared with the amount of memory required for cache data rather than control information.

In this analysis, we define the following notations and variables. n is the number of cache

Table 4.3: Time Complexity of Cache Replacement Algorithm's Overhead

	worst case		average case	
policies	hit	miss	hit	miss
FIFO	δ	$t_r + t_w + \delta$	δ	$t_r + t_w + \delta$
LRU$_{DLL}$	$3t_r + 6t_w + \delta$	$3t_r + 6t_w + \delta$	$3t_r + 6t_w + \delta$	$3t_r + 6t_w + \delta$
LRU$_S$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
LRU$_C$	$O(1)$	$O(n)$	$O(1)$	$O(n)$
LFU$_H$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
ARC (with LRU$_{DLL}$)	$O(1)$	$O(1)$	$O(1)$	$O(1)$
LIRS (with LRU$_{DLL}$)	$O(m)$	$O(m)$	$O(\frac{1}{\beta})$	$O(\frac{1}{\beta})$
CLOCK	$t_w + \delta$	$O(n)$	$t_w + \delta$	$O(\frac{1}{1-\beta})$
CAR (with LRU$_{DLL}$)	$t_w + \delta$	$O(n)$	$t_w + \delta$	$O(\frac{1}{1-\beta})$
Compact CAR (our proposal)	$t_w + \delta$	$O(n)$	$t_w + \delta$	$O(\frac{1}{1-\beta})$

Table 4.4: Space Complexity of Cache Replacement Algorithm's Overhead

	Space Complexity		
policies	memory [bit]	order	number of history
FIFO	$\log n$	$O(\log n)$	-
LRU$_{DLL}$	$2n \log n + 2 \log n$	$O(n \log n)$	-
LRU$_S$	δ	$O(1)$	-
LRU$_C$	$n \log n + \log n$	$O(n \log n)$	-
LFU$_H$	$n \cdot C$	$O(n \cdot C)$	-
ARC (with DLL)	$4n \log n + 7 \log n$	$O(n \log n)$	n
LIRS (with DLL)	$4n \log n + 2n + 2m \log n + 4 \log n$	$O(m + n \log n)$	m
CLOCK	$n + \log n$	$O(n)$	-
CAR (with DLL)	$4n \log n + n + 9 \log n$	$O(n \log n)$	n
Compact CAR (our proposal)	$n + 9 \log n$	$O(n)$	n

entries. Some policies use P-bit pointers to cache entries. P requires at least $\lceil \log n \rceil$ [bit] to identify n individual entries. For the analysis of the time complexity of variants of CLOCK, let us assume h_i denotes the number of content accessed at least i times in a certain range, β and γ represent h_2/h_1 and h_3/h_1, respectively. Note that β and γ satisfies the inequality $0 \leq \gamma \leq \beta \leq 1$ since $h_{i+1} \leq h_i$. We basically express time complexity of an algorithm as order of the function of n or β. If the complexity of a algorithm is $O(1)$ and can be accurately calculated, we describe the complexity with read time t_r, write time t_w and negligibly small time δ, which is required for the other processes, instead of big O notation, because the memory access time is a dominant factor in caching algorithm execution time.

Although we analyze only two cache replacement algorithms: CAR and Compact CAR in this section, Table 4.3 and 4.4 summarize the analytical results of space and time complexity of not only the two algorithms but also other cache replacement algorithms including FIFO, LRU, CLOCK, ARC and LIRS for the purpose of comparison. The detail explanations on the complexity analysis for the other than CAR and Compact CAR are presented in Section 4.5.7.

Space Complexity

First, we analyze the space complexity of CAR and Compact CAR. Compact CAR maintains four CLOCK lists shown in Fig. 4.1. Our simple swapping renders Compact CAR free from the additional costs of memory or process for maintaining the order of sweeping the CLOCK list. Furthermore, B_1 and B_2 do not need R-bits and the total length of the other two CLOCK lists, T_1 and T_2, is n. Thus, Compact CAR consumes $(n + 9P)$ bits for the two normal CLOCK lists whose total length is n, the two CLOCK lists without a R-bit, four information of the size of the lists, and a parameter of a target size for T_1.

On the other hand, CAR has two variable-sized CLOCK lists and two LRU lists. The variable-sized CLOCK list must support insertion (deletion) of a chunk into (from) an arbitrary position in a list allocated in physically contiguous memory. The implementation of variable-sized CLOCK needs the same data structure as LRU to keep the order of sweeping the CLOCK list. CAR is implemented with a doubly-linked lists as illustrated in Fig. 4.2(a). The space complexity of two CLOCK lists and two LRU lists is comparable to that of four doubly-linked lists whose maximum total length is $2n$. In addition, total n R-bits are required for two CLOCK lists. CAR also uses an adaptively tuned parameter called a target size, which costs at least P bits. Thus, the memory overhead is $(4Pn + n + 9P)$ bits.

Time Complexity

Then, we elaborate the time complexity of CAR and Compact CAR. Since many of the analysis is overlapped, we first elaborate the time complexity of CAR, followed by that of Compact CAR. CAR as well as CLOCK incurs $t_w + \delta$ complexity at a cache hit since it requires only to update

R-bit. The worst-case complexity at a cache miss is $O(n)$ because the hand must move n times to go around the clock in the worst case where R-bit of all entries in CLOCK is set.

The average number of hand movements at a cache miss ω is represented as n/s, where s is the number of cache misses during n hand movements. Because we aim to calculate the order of ω, our analysis can be simplified by considering the extreme case where ω is maximized in the steady state. Therefore, we consider two cases where n is maximized, and where s is minimized. For brevity, we do not show how to maximize n and minimize s here, which is obtained by the same calculation as CLOCK discussed in Appendix 4.5.7. The difference between CLOCK and CAR is that we must count not only the first and second accesses to a chunk but also the third accesses should to maximize n since the accesses turn on R-bits of entries in T_2. According to the calculation, ω satisfies the following inequality:

$$\omega = \frac{n}{s} \leq \frac{h_1 + h_2 + h_3}{h_1 - h_2} = \frac{1 + \frac{h_2}{h_1} + \frac{h_3}{h_1}}{1 - \frac{h_2}{h_1}} = \frac{1 + \beta + \gamma}{1 - \beta}.$$

Thus, the average-case time complexity depends on the characteristics of accesses rather than the cache size n, and $O(\omega) = O(\frac{1+\beta+\gamma}{1-\beta}) = O(\frac{1}{1-\beta})$ because $0 \leq \gamma \leq \beta \leq 1$. The time complexity of Compact CAR can be calculated in the same way as CAR. The time complexity at a cache hit is $t_w + \delta$. The worst-case complexity at a cache miss is $O(n)$ and that of the average-case is $O(\frac{1}{1-\beta})$.

4.5.7 Time and Space Complexity of the Remaining Policies

We analyze the time and space complexity of policies which are skipped in Section 4.5.6.

We analyze them in the same manner as described in Section 4.5.6. In addition to the notations in Section 4.5.6, we define the following notations and variables. m is the number of records of discarded chunks in LIRS. Statistical policies assign C-bit information (as a counter used in LFU) to every entry.

Complexity of FIFO

In FIFO, only a P-bit pointer to remember the head of the queue is required. When a cache hit occurs, no additional operations are necessary (except for common operations such as reading the

accessed chunk). When a cache miss occurs, there are two additional operations: reading the pointer to discard the entry at the head of the queue and updating it. Thus, the space complexity is P bits. The time complexity at a cache hit and miss are δ and $(t_r + t_w + \delta)$, respectively.

Complexity of LRU

LRU$_{DLL}$(Pointer Operation with a Doubly-linked List) To implement LRU$_{DLL}$, it is necessary to maintain a sorted doubly-linked list, where each entry has two P-bit pointers and the most recently used (MRU) entry is at the front of the list. In addition, two pointers are needed to remember MRU and LRU entries. Thus, LRU$_{DLL}$ totally requires $(2Pn + 2P)$-bit memory overhead.

Let e_i denote the i-th most recently accessed entry in LRU$_{DLL}$ ($i = 1, 2, \cdots, n$), that is, smaller i means that the entry is more recent. In addition, p_i^{prev} and p_i^{next} denote the pointers that point the previous and next entry, respectively. If e_i is accessed, e_i is moved to the front of the list. This process updates six pointers: two pointers of e_i, p_{i-1}^{next}, p_{i+1}^{prev}, p_1^{prev} and a MRU pointer. To find e_{i-1}, e_{i+1} and e_1, it is necessary to read three pointers. On the other hand, if there is a cache miss, e_n is discarded and a new entry is cached as a previous entry of e_1. After reading the addresses of first, n-th and $(n-1)$-th entries, it is required to write a new entry and update p_{n-1}^{next} and p_1^{prev} and MRU and LRU pointers. Consequently, $(3t_r + 6t_w + \delta)$ gives an estimate of time complexity imposed by LRU$_{DLL}$ in the case of both a cache hit and a cache miss.

LRU$_S$(Memory Shift Operation) LRU$_S$ introduces no additional memory cost because its data structure maintains all control information needed to perform the algorithm. The LRU entry, which is discarded when a cache miss occurs, resides at the bottom of the stack. When a cache miss occurs, a new entry stored at the top of the stack.

However, LRU$_S$ requires shifting a large amount of entries to insert or move an entry just like the algorithm described in Section 4.4.3. If e_i is accessed, all entries from e_1 to e_{i-1} must be shifted. If there is a cache miss, it is required to shift entries from e_1 to e_{n-1} and write a new entry at the top of the stack. In the worst case, n entries are moved. On average, $n/2$ entries are moved at a cache hit if all entries are uniformly referenced. Thus, time complexity of LRU$_S$ is $O(n)$. This process in a small-scale computer system is typically supported by special hardware for the shifting operation;

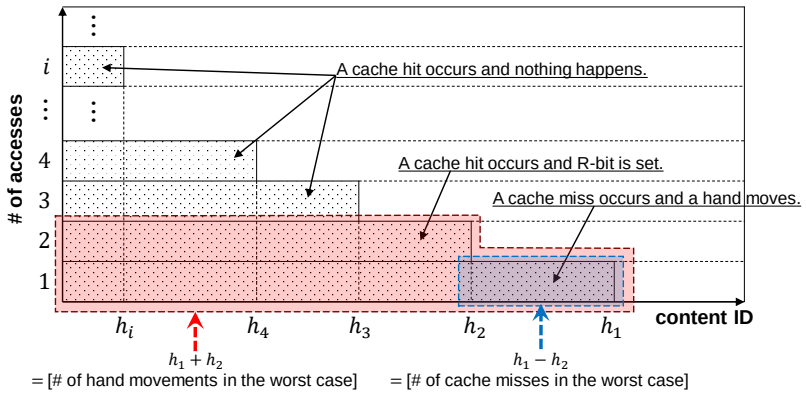


Figure 4.12: Description of calculating ω of CLOCK

however, it is infeasible for use in an ICN router because of an excessive amount of entries.

LRU$_C$ LRU$_C$ assigns each entry with a C-bit counter, which remembers the count of accesses and acts as time-stamp. In addition, a C-bit counter is necessary to remember the total number of accesses. Thus, LRU$_C$ imposes $(Cn + C)$-bit space complexity.

The time complexity at a cache hit is $O(1)$in accordance with processes updating a counter and writing the value at a new entry. The time complexity at a cache miss is $O(n)$ because of the look-up process to retrieve an entry with the minimum counter value from the unsorted list.

Complexity of CLOCK

To store n R-bits and a position located by a clock hand, the space complexity of CLOCK is $(n+P)$ bits. The time complexity at a cache hit is $(t_w + \delta)$ since it requires only to update R-bit. The worst-case time complexity at a cache miss is $O(n)$ because a hand must move n times to go around the clock in the worst case where R-bit of all entries in CLOCK is set. However, such a case rarely happens.

Let s denote the average number of cache misses during one cycle of a hand (i.e., n hand movements) to calculate the average-case time complexity $\omega = n/s$, which can be defined as the number of hand movements per cache miss on average. Fig. 4.12 gives an intuitive understanding of how to calculate n and s according to h_i defined in the time interval $[1, n]$ during n hand movements.

Because we aim to calculate the order of ω, our analysis can be simplified by considering the extreme case where ω is maximized in the steady state. Therefore, we consider two cases where n is maximized, and where s is minimized.

First, we discuss the case where n is maximized. It is obvious that the first access to a chunk causes a cache miss and rotation of a hand. A cache hit by the second access to a chunk set R-bit of the accessed entry. This entry whose R-bit is set causes a movement of a hand because the hand ignores the entry only resetting the R-bit. Even if a chunk is accessed three or more times per cycle, the accesses do not cause a hand movement. Therefore, the number of hand movements to go around CLOCK's circular list is at most $h_1 + h_2$ as illustrated in Fig. 4.12 (a red area).

Second, we determine the minimum number of cache misses s. It is clear that $s = 1$ at the minimum in the worst case where $(h_1 - 1)$ chunks have been already accessed and their R-bits are set before our considering time interval $[1, n]$. However, assuming the steady state where the popularity distribution of chunks (i.e. the distribution of h_i) is stable, there is at most h_2 chunks that is accessed before the beginning of the interval. Therefore, the number of cache misses is at least $h_1 - h_2$ as illustrated in Fig. 4.12 (a blue area).

According to the above discussion, ω satisfies the following inequality:

$$\omega = \frac{n}{s} \leq \frac{h_1 + h_2}{h_1 - h_2} = \frac{1 + \beta}{1 - \beta}.$$

Thus, the average-case time complexity depends on the characteristics of accesses rather than the cache size n, and $O(\omega) = O(\frac{1+\beta}{1-\beta}) = O(\frac{1}{1-\beta})$ because $0 \leq \beta \leq 1$.

Complexity of LFU$_H$

Because LFU$_H$ is implemented with a heap, the complexity of LFU$_H$ accords with that of a heap. If a heap is arranged in an array, (Cn)-bit space complexity is necessary because each entry holds a C-bit counter. The operation performed at a cache hit is moving an accessed entry, which is less expensive than adding a new entry. The operation performed at a cache miss is comparable to the cost of adding and deleting an entry. Both of the operations require $O(\log n)$ time complexity.

Complexity of ARC

ARC has two LRU lists and each LRU list contains n entries, therefore, the space complexity of ARC implemented with LRU$_{DLL}$ is twice as much as that of LRU$_{DLL}$. In addition, the LRU list is partitioned into two portions. To remember the partitioned location, each LRU list must maintain a P-bit pointer. ARC as well as CAR has the P-bit parameter. Thus, memory overhead of ARC grows $4Pn + 7P$ bits. The time complexity is $O(1)$ as well as LRU$_{DLL}$ because there is no repetition in ARC's algorithm.

Complexity of LIRS

LIRS uses two LRU lists which are called LRU stack S and Q. The maximum size of LRU S and Q is $(n + m)$ and n, respectively. In addition, two bits are assigned to each entry to mark a hot chunk[3] and a record of a discarded chunk. Thus, the space complexity is $(4Pn+2n+2Pm+4P)$. m is practically smaller than $4n$ [48] although the length of m, which is determined by the length of a sequence of one-time content such as SCAN, is theoretically unlimited.

Time complexity can grow significantly since there is an operation called stack pruning in LIRS. In the worst case, m records of discarded caches are removed by only a single stack pruning operation, therefore, worst-case complexity is $O(m)$. Especially, if there is a long SCAN, this overhead becomes extraordinarily large according to the length of the access pattern.

The average-case time complexity of stack pruning can be calculated in accordance with the average number of deleted entries by stack pruning, ω. Assuming n entries (i.e., the same amount of entries as the cache size) are removed by stack pruning while stack pruning is conducted s times, ω can be defined as n/s. Specifying the time interval of h_i accordingly, h_1 accesses causes cache misses, h_2 accesses render the accessed entry hot switching the LRU hot chunk into a cold chunk and trigger stack pruning. Because the other $\sum_{i\geq 3} h_i$ accesses treated as accesses to hot entries, stack pruning is not conducted by the accesses. According to the above calculations, the average-case time complexity is $O(\omega) = O(h_1/h_2) = O(1/\beta)$. The more one-time accesses occupy the traffic, the larger this complexity becomes.

[3] LIRS classifies chunks into two types: a hot chunk and a cold chunk. Briefly, 'hot' means to be popular and 'cold' means to be unpopular. They have similar features to the two lists T_1 and T_2 in Compact CAR.

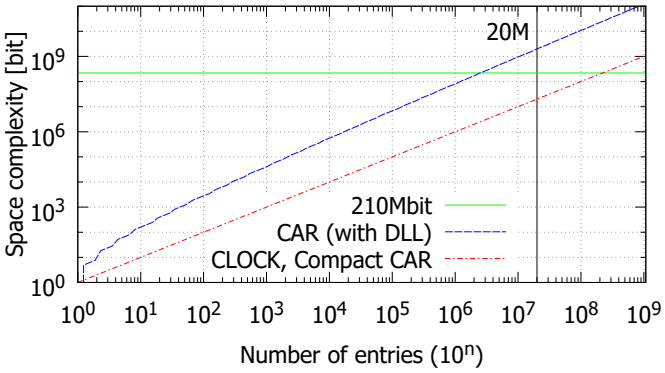


Figure 4.13: Space Complexities of CAR and Our Proposal(Compact CAR)

4.6 Discussion on the Implementation of Compact CAR for High Performance ICN Core Router

4.6.1 Feasibility of Hardware Implementation

The throughput and capacity of a cache are the most serious obstacles to realize an ICN core router. Assuming 10 Gbps of traffic and with 64-byte data packets, a single-line card has the throughput of approximately 20 million accesses per second at maximum (equivalently, 50 ns access time at a minimum). Since routers typically contain many line cards, a cache mechanism in a router must realize a level of throughput in linear proportion to the number of line cards. In practice, the existence of interest packets, data packets larger than 64 bytes, and skipping cache accesses by cache hit may ease the required access time several-fold.

Figure 4.13 shows a memory overhead of CAR and Compact CAR. As explained in Section 4.5.6, CAR using a doubly-linked list consumes $(4Pn + n + 9P)$ bits. Assuming a router holds 20 million cache entries, the memory cost of CAR becomes 2 Gbit to hold 20 million entries because $P \geq \lceil \log n \rceil$. This cost is prohibitive according to the constraint of SRAM, whose available size is 210 Mbit [30]. On the other hand, Compact CAR requires a memory overhead of one bit per entry. Compact CAR consumes 20 Mbit; therefore the memory cost of Compact CAR is feasible.

4.6.2 Computational Overhead of Variants of CLOCK

In Section 4.5.6, we analyzed the computational cost of Compact CAR, which provides the complexity of $O(1/(1-\beta))$ in terms of β. It may be arguable that the complexity could be extremely large as the parameter β becomes close to 1.0. In fact, the β values of content-level and packet-level workloads used in our simulation ranges from 0.38 to 0.71 and from 0.08 to 0.22, respectively. $\frac{1+2\beta}{1-\beta}$ showing average-case time complexity of Compact CAR is less than only 2.0 when $\beta < 0.2$. $\frac{1+2\beta}{1-\beta}$ grows 6.0, which is the computational cost of LRU, when β becomes 0.625. $\frac{1+2\beta}{1-\beta} < 8.0$ even if $\beta < 0.7$. Although the space complexity of CAR can be reduced by using a memory shift operation instead of a doubly-linked list, the memory shift operation makes the time complexity prohibitive as illustrated in Fig. 4.2(b).

If SRAM access time is 0.45 ns [30], the router can handle about 278 million accesses per second even if eight hand movements per access are required as discussed above. Assuming 64-byte data packets, this throughput is 142 Gbps. In conclusion, the computational cost of Compact CAR is acceptable in the design of high performance ICN core router.

However, the data of the chunks must be kept in a scalable memory, such as dynamic RAM or a solid-state disk. Since such memory is slow, we plan to consider a hierarchically structured cache memory and a pipelined process to ensure a high average speed for read/write accesses. We will eventually evaluate the router performance in a hardware implementation of the router, combining Compact CAR and a name lookup entity [29], to demonstrate the feasibility of the router.

4.7 Summary

A few researches have been done for cache replacement algorithms in the context of ICN because they have been intensively researched in the fields of web-caching and a CDN previously. This chapter argued that the conventional cache replacement algorithms cannot be directly applied to the design of a high performance ICN core router.

For this reason, we proposed a novel cache replacement algorithm named Compact CAR which would be an important component in the design of a high performance ICN core router. Compact CAR outperforms compared to conventional cache replacement algorithms in terms of cache hit

rate and memory usage in the design of ICN router. In detail, the proposed algorithm can achieve the same cache hit rate with only one-tenth of memory usages that simple conventional algorithms consume. In addition, the cache hit rate by the proposed algorithm is only 10% less than the optimal case over the various simulation scenarios. In particular, the difference becomes negligible when we use real traffic traces whose RD values are similar to the cache size. This result provides a clue that a high cache hit rate can be achieved if the cache size adaptively changes according to the distribution of RD value in real traffic. Furthermore, Compact CAR can dynamically adapt itself to the network environment whose traffic access patterns change dynamically, which is important to deal with various traffics in ICN.

ICN has been researched nearly 10 years and it may be the time to consider its deployment issue in Internet-scale where the design of a high performance ICN core router becomes critical. We believe that the proposed cache replacement algorithm plays a key role in the design of such a high performance ICN core router in near future.

Chapter 5

Scalable Cache Component in ICN Adaptable to Various Network Traffic Access Patterns

5.1 Introduction

In this chapter, we study more about the cache replacement algorithm. In Chapter 4, we proposed Compact CAR to deal with the SCAN access pattern caused by a large amount of one-time contents. This feature significantly improve the performance of content store because the popularity of Internet traffic approximately follows a Zipf distribution, where a large amount of packets carrying redundant data are transferred. According to [37], content requested more than once occupy 50% or more of the volume of network traffic, with the value depending on the observation period. However, we designed it under the conditions without the constraints of the memory cost of a lookup table. We also did not cover all access patterns that appear in network traffic.

We propose a cache replacement algorithm that is suitable for network traffic, addresses more access patterns of network traffic than Compact CAR, and is scalable enough to address the challenge of implementing a cache mechanism in resource-restricted hardware. Our proposed algorithm, called CLOCK-Pro Using Switching Hash-tables (CUSH), is inspired by CLOCK-Pro [39].

CUSH satisfies the performance requirements by employing two strategies: two types of chunks and ghost caches. Classifying chunks enables a cache to detect and hold popular content. Ghost caches are the records of content discarded from the cache. By storing the historical information of accesses rather than the data of content, CUSH can adapt to the variety and dynamism of network traffic access patterns while avoiding an excessive memory cost.

CUSH also satisfies the cost requirements by adopting the low-overhead data structures: a CLOCK list and hash tables accepting hash collisions for ghost caches. The CLOCK list is a low-overhead variant of LRU, which requires only one-bit per entry and achieves as good performance as LRU. The hash-tables significantly reduce the cost of ghost caches and the lookup table, which are expensive in an ICN router because of a long name used in the ICN communication. Although these low-overhead data structures cannot directly adapt to our high-performance strategies, we devise the algorithm that can be performed on the low-overhead data structures and exploits the benefits of our strategies.

This chapter is organized as follows. We summarize various cache mechanisms ranging from computer systems to in-network caching in Section II and the following Section III uses the knowledge of them to reveal the design considerations of cache replacement algorithm for a network environment. In Section IV, we give the description of CUSH. The simulation results in Section V show that our proposed approach, which can be assumed as a low-overhead variant of CLOCK-Pro, is comparable to CLOCK-Pro and outperforms simple policies. In addition, our proposal can achieve cache hits against the traffic traces where simple conventional algorithms cannot cause any hits. The section also clearly shows that CUSH is low-overhead by comparing time and space complexity of cache replacement algorithms. Finally, we conclude this article in Section 5.6.

5.2 Related Works

Because cache resources are limited, practical cache replacement algorithms are necessary to keep popular content and remove rarely used one. There are a considerable number of cache replacement algorithms, ranging from those used in software (e.g., database and web applications) to those

available in hardware (e.g., CPU and I/O buffers). In this section, we review several cache replacement algorithms that have been carried out in the different context to understand the requirements of in-network caching.

Replacement algorithms are developed originally for the purpose of paging in the computer system [46, 38]. The bottleneck of the systems is the latency of fetching pages from slow auxiliary memory to fast cache memory. On the one hand, the hardware cache such as CPU commonly used First-in, first-out (FIFO) and Not Recently Used (NRU) to reduce the memory and computational costs because of the hardly limited resources. On the other hand, the software cache such as virtual memory of OS commonly adopts LRU and LFU, which are costly to maintain a data structure or/and statistical information (i.e., the number of references to a page).

Then, researchers have uncovered access patterns that degrade the performance of the algorithms. To overcome the problematic access patterns, many variants of LRU and LFU are developed. 2Q [47], ARC [46] and LIRS [48] improve the performance by exploiting the advantages of LRU and LFU while their time and space complexities are comparable to that of LRU. They also hold *ghost caches*, which are the records of evicted chunks rather than the data of the chunk to adapt to the variety and dynamism of access patterns. In contrast to them, CLOCK [49] reduces the complexity of LRU by approximating its behavior with a fixed circular buffer while keeping the performance. The complexity of CLOCK is comparable to that of NRU which has a low computational cost. CLOCK-Pro [39] combines CLOCK with LIRS to achieve both performance improvement and cost reduction.

After the emergence of the web services, web-cache and CDN-cache are researched intensively to improve the performance of them in terms of bottleneck, latency, overload and robustness [50, 51, 52]. Because the resource constraints of them are more moderate than those of computer systems, the cache replacement algorithms in a web and a CDN utilize statistical information including not only recency and frequency but also several others including size, latency, and URI [51]. However, the improvement was slight or specific to particular environments in spite of an abundance of caching algorithms [52].

In recent years, ICN has revived research on caching algorithms because ICN provides inherent in-network caching feature. Unlike web- and CDN-cache employed in the application-layer, all

devices in ICN have caching capability. Because one of the most interesting problems is improvement achieved by through cooperation among ICN routers in the network-layer, many researchers focus on cache placement algorithms [56, 57]. As a cache replacement algorithm taking advantage of ICN, there are also policies that make use of content popularity [20, 21]. Previous chapters on caching use only LRU [17, 18] or claim that the effect on performance of cache replacement is minimal [19]. However, the chapters use only blunt cache replacement policies and ignore the suitability for network traffic. In fact, there are studies that exhibit the capability to improve the performance of a network [20, 21]. Cache replcement policy based on content popularity (CCP) [20] can significantly decrease the server load and increase cache hit rate compared to that of LRU and LFU. The work in [21] analyzed the effects of chunking and proposed Highest cost item caching (HECTIC), which uses a utility-based replacement algorithm and outperforms existing polices including LRU. Their statistical approaches are too expensive to be employed in an ICN core router due to computational and memory costs. However, we propose a low-overhead cache replacement policy that outperforms LRU-based and simple replacement policies by coping with access patterns specific to ICN. Compact CAR is low-overhead and copes with a part of traffic access patterns; however, the policy is weak to LOOP and assumes that the memory cost of the lookup table for ghost caches is acceptable, which are discussed in the following section.

To realize ICN, especially an ICN core router, it is required to implement a cache replacement algorithm that can be operated with severe resource constraints instead of the statistical caching algorithms for web and CDN with rich resources. The implementation cost of commonly used approaches such as LRU and LFU are also prohibitive for router hardware, as pointed out by [22, 19]. Looking back at the history of cache replacement algorithms, ICN routers need a hardware implementable approach whose complexity is comparable to that of FIFO or CLOCK. In addition to the cost, this approach should cope with access patterns specific to ICN, where the unit of caching is a fine-grained chunk rather than whole content data. To understand how to satisfy these requirements of cost and performance, we examine the knowledge of caching in computer systems and apply it to in-network caching in the following section.

5.3 Design Considerations of Cache Replacement Algorithm for ICN

5.3.1 Access Patterns of Traffic in the Network

An access pattern is the important factor to govern the performance of cache replacement algorithm. It is well known that the popularity of content follows a Zipf-like distribution: a large number of content requested only once or just a few times [54]. In addition, many requests generate asynchronous requests for content, and so the temporal locality of network traffic becomes relatively low.

In particular, ICN is able to identify a chunk (its default size is 4K bytes in CCNx), which enables the chunk level caching in an ICN router. Thus, we conjecture that the distribution of the "chunk popularity" would be more biased than Zipf-like distributions. In this chapter, to design an appropriate cache replacement algorithm for ICN under different types of the distributions, we classify access patterns of traffic, which governs the distribution [47, 46, 53, 48], into four categories: SCAN, LOOP, COOREALTED-REFERENCES, and FICKLE-INTEREST as follows:

- SCAN: a sequence of requests to different chunks, and so each chunk is requested only once
- LOOP: a repetition of a scan
- CORRELATED-REFERENCES: a short-term intensified requests to a few chunks
- FICKLE-INTEREST: rapidly changing sets of requested chunks

First, although the exact access pattern of the chunk level (i.e., network level) traffic in ICN is not known due to the lack of available ICN traffic trace, such one-time used items occupy 60% or more in the network level traffic in IP networks [37]. We conjecture that the highly aggregated network level traffic in ICN would have a large number of one-time used chunks, which correspond to SCAN access pattern. SCAN makes the performance of an LRU-based approach much poor because such unpopular content occupies the whole cache.

Second, ICN is originally designed to efficiently disseminate multimedia traffic which generally occupies high network bandwidth and is requested repeatedly. Thus, we also conjecture that the chunk level traffic in ICN will have LOOP access pattern. As mentioned previously, LOOP is

highly correlated to SCAN: SCAN and LOOP are generated by unpopular and popular content, respectively. Each chunk in LOOP is removed from a cache before being accessed again. LOOP has adverse effect on a LFU-based approach as well as LRU because all chunk in LOOP have the same recency and frequency.

Third, CORRELATED-REFERENCES and FICKLE-INTEREST access patterns are observed in the requests to user-generated content and real-time content, respectively. We conjecture that these access patterns would be frequently observed in ICN due to the growth of social networks that share user-generated content as well as real-time application such as video chatting. The volatile traffic hinders the strategies depending on statistical information (including LFU) from replacing the out-of-date chunks that were accessed frequently.

For the reason above, the cache replacement algorithm for ICN should be able to deal with the access patterns described above. We here focus on SCAN and LOOP in the design of the cache replacement algorithm for ICN since it is the major traffic that occupies the network bandwidth. Among the conventional cache replacement algorithms, CLOCK-Pro is able to efficiently deal with the traffic access patterns [39] due to its strategy based on inter-reference recency (IRR) which enables to cope with SCAN by distinguishing popular and non-popular content and furthermore keep the appropriate number of popular content for the traffics with LOOP. Our proposal is based on CLOCK-Pro to inherit this feature.

5.3.2 Computational and Memory Cost

The cache replacement algorithm in an ICN router must satisfy the requirements for consider computational and memory costs: the former is the cost that updates the table holding the information of cached items in the ICN router, and the other is the cost that manages the table in the memory according to a cache replacement algorithm, e.g., prioritizing cached items. The computational cost includes insertion of a new caching item into the table, deletion of an existing cached item from the table, moving the location of cached items in the memory, and updating relevant information in the caching table. The memory cost increases as the number of cached items increases due to the increase of control information for the maintenance of the table [39, 38, 22, 19]. For example, LFU

has much higher overhead to keep statistics of each cached item. In LRU using double-linked-list, this cost is prohibitive due to the maintenance of double pointers to other cached items.

CLOCK can satisfy the requirements. CLOCK has a memory link list having a shape of a clock and assigns one-bit to each entry in the list. CLOCK searches for an entry containing a cached item that needs to be replaced following a clockwise. CLOCK can keep recently accessed content because CLOCK judges whether a found entry should be removed by the assigned bit, which is set to on whenever the cache is accesses. When the bit is on, CLOCK unsets the bit and continues the search process; otherwise, CLOCK removes the cache. Thus, CLOCK requires only a single bit per chunk and few repetitions of the searching process. Our proposed mechanism also adopts this mechanism in CLOCK to reduce the computational and memory costs.

5.3.3 Overhead on Lookup Table caused by Ghost Caches

Some cache replacement algorithms such as LIRS and CLOCK-Pro hold ghost caches, which are historical record of cached content items rather than cached data. Although the ghost caches are essential to distinguish the chunks worth caching from chunks in SCAN and LOOP, the ghost caches impose high memory overhead on a cache lookup table. For example, consider the lookup table for a cache replacement algorithm using ghost caches as depicted in Fig. 5.1. In the cache in ICN, the lookup table maps the name of a chunk to the address of the entry for the chunk in the data structure of a cache policy. In the figure, we assume CLOCK-Pro as the cache replacement policy, which can cache n chunks and remember n ghost caches. Entries for the cached chunks (unshaded) and the ghost caches (shaded) are stored in the CLOCK lists shown in the bottom of the figure. In this example, the chunk "A.mpg/s1" is cached and is mapped to the entry at the address 3 in the CLOCK list. The chunk "/B.mpg/s1" is not in the cache memory but only its historical record is stored in the entry at the address $2n$ as a ghost cache.

Then, let us discuss the memory cost of a lookup table to keep ghost caches. Assuming the lookup table is implemented as a simple key-value store, it needs to store a name as a key and an address as a value. Assuming the maximum length of a name is L_{max}[bit], the lookup table totally needs $2n(L_{max} + \log 2n)$[bit], where $n(L_{max} + \log 2n)$[bit] are consumed to keep n ghost caches.

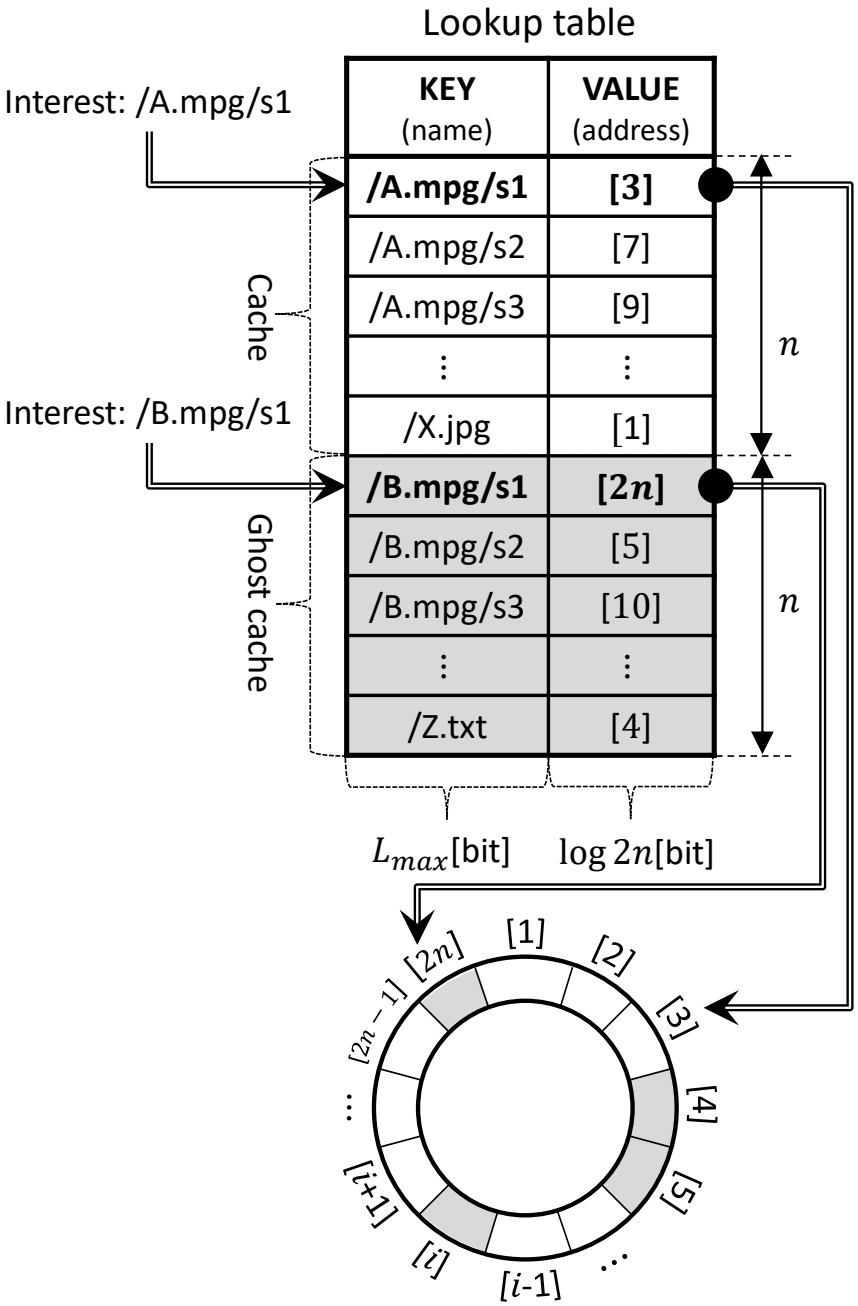


Figure 5.1: Ideal Lookup Table for the Cache Replacement Policy using Ghost Caches (e.g., CLOCK-Pro)

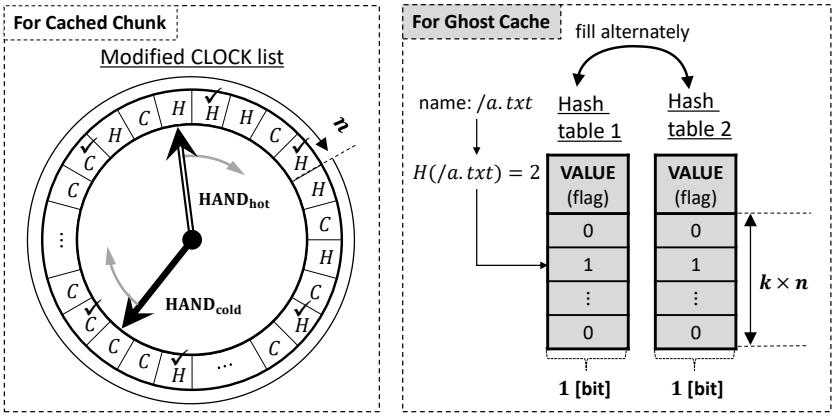


Figure 5.2: Data Structure of CUSH

Since L_{max} can be at most 512 Kbit according to the specification of CCNx, the memory cost easily becomes prohibitive. Of course, previous literature on ICN reduces the cost by proposing a low-overhead lookup table such as a tree-based implementation [16]; however, it is expensive to double the number of entries in a lookup table for ghost caches. To solve this challenge retaining benefits of ghost caches, our proposal addresses the challenge of minimizing the additional cost of ghost caches as discussed in Section 5.4.3.

5.4 CLOCK-Pro Using Switching Hash-table (CUSH)

5.4.1 Data Structure of CUSH

CUSH has two component parts as shown in Fig. 5.2. One is the circular buffer which works as a modified CLOCK list, whose entry holds information to point to an actually cached chunk. The CLOCK list has two types of entries: *hot* entries and *cold* entries. H and C in the figure denote a hot entry and a cold entry, respectively. The hot entry points to a frequently accessed chunk, and the cold entry points to the rest. Precisely, they are classified based on IRR as explained later. The other component is the two hash-tables for ghost caches, which act as "the loser bracket" providing an opportunity for discarded chunks to be re-cached.

The CLOCK list assigns two bits to each entry: a bit that indicates whether it is hot or cold

(H-bit), and a reference bit (R-bit). The R-bit indicates whether the entry has been accessed. H-bit is set to "1" when the entry is hot; otherwise, "0". R-bit is set to "1" when the chunk is accessed. The entry with a check mark in Fig. 5.2 indicates its R-bit is set to "1". R-bit is set to "0" by using the two hands: $\mathrm{HAND}_{\mathrm{cold}}$ and $\mathrm{HAND}_{\mathrm{hot}}$. $\mathrm{HAND}_{\mathrm{cold}}$ is used to discard a cold entry when cache replacement occurs. $\mathrm{HAND}_{\mathrm{hot}}$ is used to change a hot entry to a cold entry. Both hands remember the position to start the process. When the process is needed, the hand rotate clockwise to search a entry whose R-bit is "0", and rotates ignoring it. When $\mathrm{HAND}_{\mathrm{cold}}$ ($\mathrm{HAND}_{\mathrm{hot}}$) encounters a cold (hot) entry whose R-bit is "1", the hand resets its R-bit to "0" and continues to rotate ignoring the entry. In the process of $\mathrm{HAND}_{\mathrm{cold}}$, $\mathrm{HAND}_{\mathrm{cold}}$ also sets its H-bit to "1". In addition, $\mathrm{HAND}_{\mathrm{cold}}$ ($\mathrm{HAND}_{\mathrm{hot}}$) ignores a hot (cold) chunk.

The two hash-tables store ghost caches, which are the information of chunks discarded from the CLOCK list recently. The information of a ghost cache is recorded as the flag bit in the hash-tables. In detail, when a chunk whose name is /a.txt has been removed from the CLOCK list, the flag bit of an entry at the address of $H(/a.txt)$ is set to "1", where $H(name)$ is the mapping function that maps a name in ICN to the address of a flag bit in hash-tables. When the chunk is accessed, the chunk is cached as hot if its corresponding flag bit is "1"; otherwise, it is cached as cold. CUSH has two hash-tables to fill (and clear) them alternately. It is undesirable to have only one hash-table and remove all ghost caches when needed because the chance of the access to a ghost cache (i.e., the chance to detect popular chunks) is lost shortly after the clear operation.

n denotes the number of chunks that a router can cache in the cache memory. The number of entries in the CLOCK list is also n because the CLOCK list is for actually cached chunks. The number of entries in hash-tables can be expanded to $k \times n$, where k is determined according to the characteristics of access patterns and the memory cost. k is important because the LOOP-resistant property of CUSH depends on the number of ghost caches. Although longer LOOP requires larger k, the memory cost is very small because the cost of a flag bit is 1-bit.

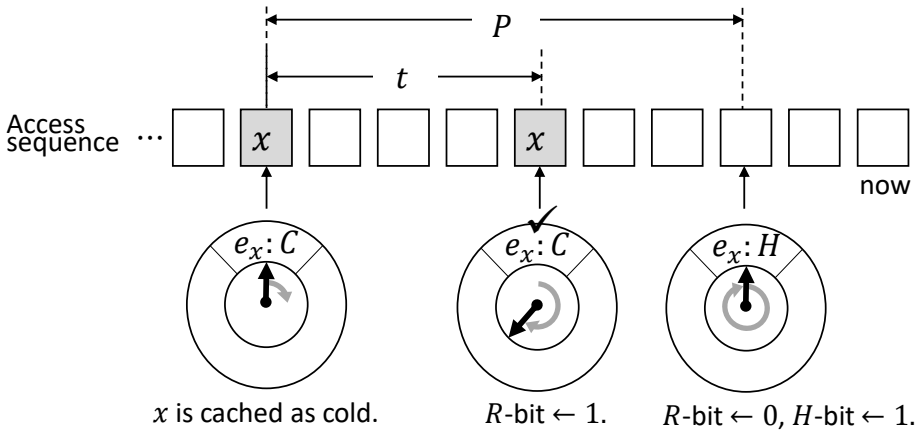


Figure 5.3: The Operation of CUSH in the Case where e_x Becomes Hot

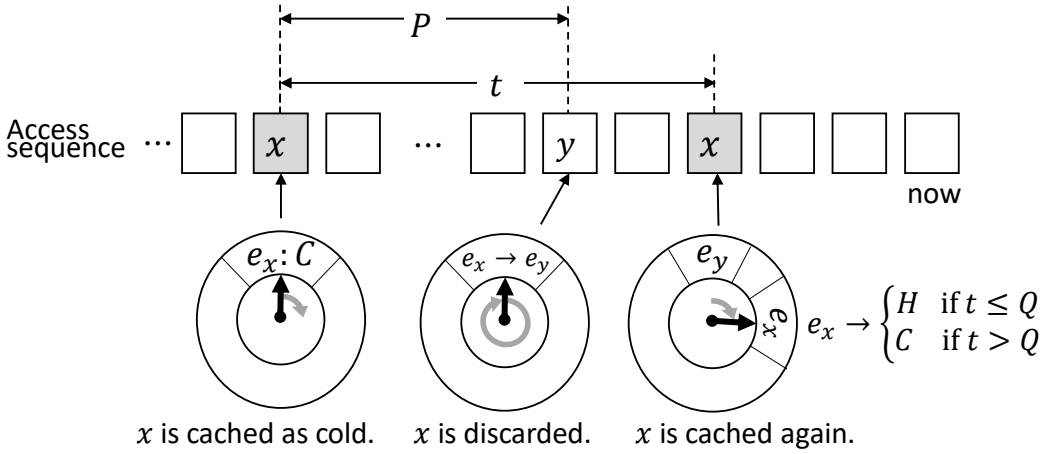


Figure 5.4: The Operation of CUSH in the Case where e_x Becomes Cold

5.4.2 Operation of CUSH

Here, we explain the operation of CUSH. There are two cases when a new request arrives: (1) the requested chunk is in the cache memory or (2) the chunk is not in the cache memory. In the case (1), the entry of the requested chunk is in the CLOCK list. If the R-bit of the corresponding entry is "0", it changes to "1"; otherwise, it remains "1". In the case (2), CUSH firstly retrieves the requested chunk from the source of the chunk. Then, CUSH verifies whether the requested chunk has been cached previously or not by checking the hash tables. If the flag at the address of $H(name)$ is "1",

it means a *ghost hit*, that is, the chunk with the name has been cached previously but the chunk does not exist in the cache memory. Thus, when the chunk is cached, its entry is added to the CLOCK lists as a hot entry. If the flag bits in the hash tables are "0", the requested chunk has not been cached recently. Thus, a new cold entry for the chunk is added to the CLOCK list. We elaborate these processes in detail in the following.

Figures 5.3 and 5.4 illustrates the operation classifying the entry into hot or cold. The figures show the recently accessed requests to a router until now, which are arranged in the order of the accessed time. In this example, we focus on a certain chunk "x" and illustrate how to classify its entry denoted by "e_x" in the CLOCK list. The figures show the case when two requests to the chunk x arrives with the time interval t. The time interval t is the number of chunks accessed from the penultimate access to the last access, which is the definition of IRR [48]. In addition, we introduce two more parameters: P and Q. P is the rotation period of $\mathrm{HAND}_{\mathrm{cold}}$. Q is the period clearing the hash-tables (we discuss the period in Section 5.4.4). For brevity, we illustrate only $\mathrm{HAND}_{\mathrm{cold}}$, which rotates whenever a new chunk arrives and a cache replacement process is performed.

Here, we can consider two cases: (a) the first chunk x has a hot entry and (b) the first chunk x has a cold entry. In the case (a), the hot entry e_x remains in the CLOCK list regardless of t. The case (b) is further divided into two cases: (i) $t \leq P$ and (ii) $t > P$. Figure 5.3 shows the case (i). The cold entry e_x becomes hot because e_x is accessed before $\mathrm{HAND}_{\mathrm{cold}}$ encounters e_x. Figure 5.4 shows the case (ii). The cold entry e_x is discarded from the CLOCK list and is registered in the hash-table to provide an opportunity for the discarded chunk x to be re-cached with a hot entry. In short, t affects the behavior of CUSH. Since t represents IRR, a chunk with low IRR is classified as hot and remains in the CLOCK list. On the other hand, a chunk with high IRR remains cold and is finally discarded when $\mathrm{HAND}_{\mathrm{cold}}$ encounters it.

In the case (ii), in addition, the parameter Q determines whether e_x moves back to the CLOCK list or is discarded. If $t < Q$, e_x moves back to the CLOCK list because x is requested before its flag bit is cleared; otherwise, x is completely discarded. When e_x moves back to the CLOCK list, e_x becomes hot because it is assumed to have low IRR.

CUSH can deal with SCAN and LOOP by using two types of chunks and ghost caches. The both access patterns pollute a cache, that is, all chunks in a cache is removed from a cache by the

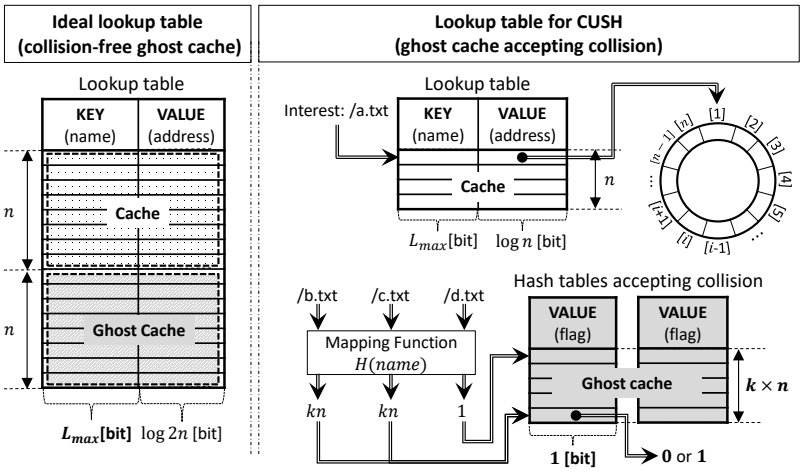


Figure 5.5: Variations of Lookup Table for Ghost Caches

access patterns as explained in Section 5.3.1. CUSH can avoid the adverse effect of SCAN because only cold chunks are replaced by one-time requested chunks and popular chunks remain as hot chunks. CUSH can also cope with LOOP as explained in Section 5.4.4.

To adapt the dynamics of access patterns, CUSH adjusts the target number of hot/cold chunks (denoted by m_h and m_c, respectively). According to the target number, CUSH manages the number of hot/cold chunks. When accesses tend to have low IRR, CUSH increases m_c and attempts to behave like CLOCK, which focuses on the recently accessed chunks. It is undesirable to increase m_h against low IRR because CUSH tries to hold out-of-date chunks as hot chunks and it becomes hard to keep up with the volatile popularity such as CORRELATED-REFERENCE and FICKLE-INTEREST. On the other hand, it is desirable to increase m_h against accesses with high IRR. CUSH copes with the accesses with high IRR such as SCAN and LOOP by holding chunks worth caching as hot chunks and ignoring wasteful chunks such as one-time requested chunks. Specifically, m_c increases whenever a cache hit or a ghost hit occurs. m_h increases whenever $\mathrm{HAND}_{\mathrm{hot}}$ encounters a cold chunk and hash-tables are switched. Because the parameters satisfy $m_h = m_c$, the one parameter decreases when the other parameter increases.

5.4.3 Collision-free Lookup Table vs Hash-tables Accepting Collision

Figure 5.5 compares the overhead of a lookup table for a cache replacement policy using ghost caches. As shown in the left of the figure, which is the same as Fig. 5.1, ghost caches impose a high memory cost on a lookup table although ghost caches are essential to deal with SCAN and LOOP. The memory cost becomes prohibitive due to a name used in ICN in contrast to the computer system as explained in Section 5.3.3.

CUSH can avoid the memory cost on a lookup table for ghost caches. As shown in the right of Figure 5.5, CUSH requires the lookup table only for the CLOCK list. The hash-tables of CUSH do not require the support of the lookup table because an entry in the hash-tables can be accessed by computing $H(name)$. Thus, we can reduce the memory cost of the lookup table for ghost caches.

To reduce the cost, CUSH must accept a hash collision on searching a ghost cache. Suppose, for example, hashes of /b.txt and /c.txt take the same value as shown in Figure 5.5. When an initial request for /c.txt arrives at a router after /b.txt was removed from the cache and has been stored as a ghost cache, CUSH stores /c.txt as a hot chunk because $H(/b.txt) = H(/c.txt)$ and a hash collision occurs. It is concerned that, if /c.txt is one-time requested content, the unpopular content wastes the cache capacity for a long time and degrades the cache hit rate of CUSH.

However, the collision problem is less serious compared to the benefits of dealing with the access patterns. In addition, the scalable data structure of the hash tables enables to reduce the probability of hash collisions. We can expand the hash space by increasing k. Another solution is to use multiple bits instead of a one-bit flag. We explore the influence of the difference in the implementations of ghost caches in Section 5.5.2.

5.4.4 Management of Hash-tables to deal with LOOP

CUSH can deal with LOOP by using two types of entries (i.e., hot and cold entries) and ghost caches. The existing LOOP-resistant policies (e.g., CLOCK-Pro) requires to maintain an ideal lookup table without a collision and manage the number of ghost caches appropriately. To reduce the memory cost of existing policies, we propose a method to deal with LOOP without such an expensive lookup table as described below.

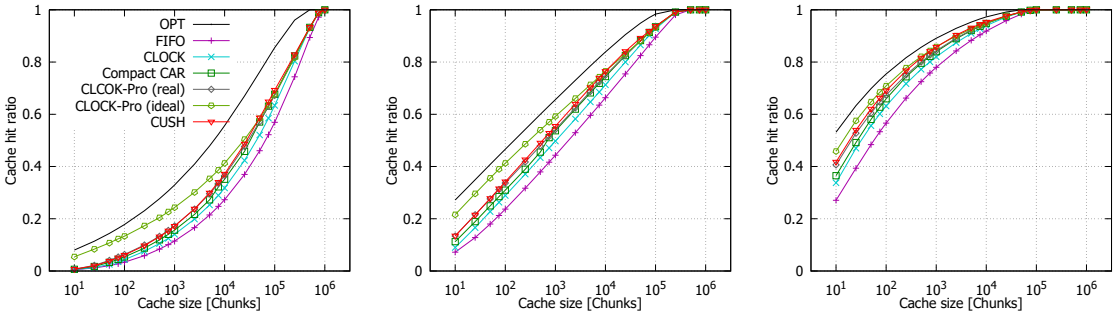
As mentioned in Section 5.3.1, each chunk in LOOP is removed from a cache before being accessed again; therefore, the all accesses to chunks in LOOP cause cache misses in spite of the repetition of accesses. Although LOOP is the repetition of the same SCAN accesses, the solution to SCAN, which classifies chunks into two types and keeps hot chunks, cannot be directly applied to LOOP. This is because all chunks in LOOP have the same recency and frequency; therefore, the number of chunks becoming hot is too large and the chunks overflow the capacity of hot chunks. Specifically, even if the ghost cache of a chunk in LOOP is accessed and the chunk becomes a hot chunk, too many chunks also becomes hot after that and the chunk is removed from the cache before being accessed again. Because this is applied to all chunks in LOOP, a cache hit never occurs.

Our solution to LOOP is not only to detect the LOOP access pattern but also to hold a cacheable portion of the chunks in the LOOP as hot chunks. The detection needs ghost caches, which is already introduced in CUSH, and the hold process needs to manage the number of ghost caches. To realize the latter process, CUSH clears the hash-tables at an appropriate time interval. If the interval is too long, CUSH tries to hold more hot chunks than the capacity and hot chunks overflow as described above. On the other hand, the too short interval reduces the opportunity to detect the repetition of LOOP because ghost caches are cleared before cacheable chunks are detected. We propose the criteria to determine the appropriate interval according to the number of hot chunks n_h. n_h is the upper bound that hot chunks do not overflow. Thus, CUSH clears hash-tables whenever the count of both a cache hit and a ghost hit becomes n_h.

Because CUSH has two hash-tables, in fact, CUSH clears the two hash-tables alternately whenever the count of hits becomes $n_h/2$. CUSH stores ghost caches into one of the two hash-tables until $n_h/2$ hits occur, and then the hash-table is switched and cleared. Although the lookup operation must be performed against two hash-tables, we can support the parallel lookup by hardware-implementation.

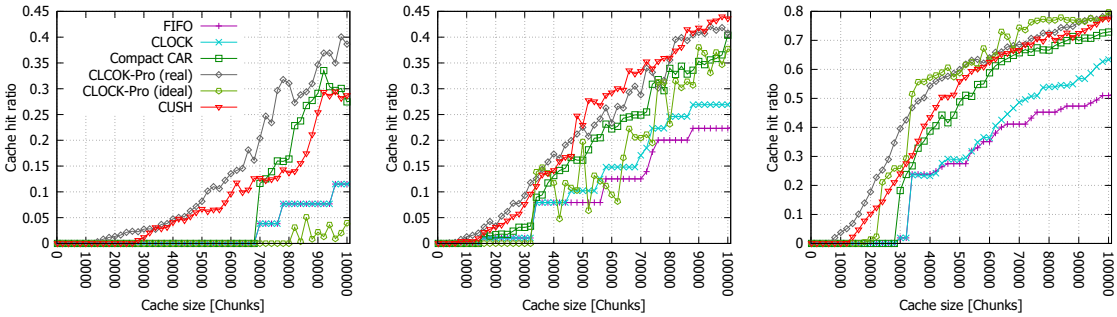
5.5 Performance Evaluation

This section explores the following three facts with simulation studies: (1) Our proposal improves the performance against the access patterns specific to network traffic compared to existing cache



(a) $\alpha = 0.8$ (b) $\alpha = 1.0$ (c) $\alpha = 1.2$

Figure 5.6: Evaluation for SCAN-resistant Property with Synthetic Workloads in Units of Content



(a) $\alpha = 1.0, L = 1.5\text{KB}$ (b) $\alpha = 1.2, L = 1.5\text{KB}$ (c) $\alpha = 1.4, L = 1.5\text{KB}$

Figure 5.7: Evaluation for LOOP-resistant Property with Synthetic Workloads in Units of Chunks

replacement policies policies. (2) Our proposed management scheme of ghost caches improves the LOOP-resistant property in terms of its capacity and its collision probability. (3) The memory and computational costs of our proposal are low enough to install it into an ICN router.

5.5.1 Evaluation of Property Resistant to Access patterns using Synthetic Traffic

First, we demonstrate the adaptability of CUSH to several traffic access patterns compared to some cache replacement policies. The access patterns on which we focused in this simulation are SCAN and LOOP, which are caused by a large number of one-time used content items and the fine granularity of items as stated in Section 5.3.1. Unfortunately, ICN traffic traces are not available yet. Thus, we generate two types of synthetic traffic workloads: requests for content items and requests

for chunks, which simulate the access patterns of ICN. The synthetic workloads follow a Zipf-like distribution because the popularity of Internet content (e.g., VoD, web pages, file sharing, and user generated content) has been reported to follow the Zipf-like distribution [40, 37]. According to the previous literature, we vary α, which is a constant parameter of the Zipf-like distribution, from 0.8 to 1.4. We also change the chunk size L from 1.5 KB to 60 KB.

The cache replacement policies used in our evaluation are OPT (here, an offline optimal algorithm with a priori knowledge of the stream of requests), FIFO, CLOCK, Compact CAR, CLOCK-Pro, and our proposal. OPT provides the absolute upper bound on the achievable hit rate. FIFO, CLOCK and Compact CAR are the policies consuming the low memory and computational costs which can be implemented in an ICN router. Compact CAR is SCAN-resistant but is weak to LOOP. Although Compact CAR is originally designed on the assumption that it has the collision-free ghost cache as depicted in the Fig. 5.1, it uses the ghost cache that allows hash collision in our evaluation. CLOCK-Pro also uses ghost caches but we evaluate the performance of both implementations to show the effect of hash collision. `CLOCK-Pro(ideal)` and `CLOCK-Pro(real)` denote CLOCK-Pro using the collision-free ghost cache and CLOCK-Pro using the ghost cache without collision resolution, respectively.

An ICN router manages n entries, where n ranges from 10^1 to 10^6 chunks which are adjusted according to the traffic trace we adopt. In this evaluation, we assume that CUSH consumes $4n$ [bit] for ghost caches $4n$. We did not use complicated topology because our purpose is demonstrating the adaptability of our proposal to network traffic. A topology we used in our simulation can be assumed to have only one ICN router between clients and a server. The transmission delay of each chunk on links and the unnecessary computation in the protocol stacks are also ignored to simplify the simulation.

Figure 5.6 depicts the cache hit rates for synthetic workloads in units of content. There is SCAN in these scenarios but LOOP does not appear. CUSH improves utilization efficiency by up to several-fold compared with simple policies that are weak to SCAN. We can find CUSH is superior to Compact CAR although we conjectured Compact CAR was better in a scenario without LOOP. This is because Compact CAR fails to detect and hold popular chunks caused by the hash collisions. Our low-overhead approximation shows the performance as good as original `CLOCK-Pro(real)`,

Table 5.1: Cache Policies Used to Analyze Effect of Ghost Cache

Name	Implementation of ghost cache
CUSH-ht1(xk)	Hash-table with kn 1-bit entries
CUSH-ht2(xk)	Hash-table with kn 2-bit entries
CUSH-ht4(xk)	Hash-table with kn 4-bit entries
CLOCK-Pro(real)	Hash-table without collision resolution
CLOCK-Pro(ideal)	Hash-table with collision resolution

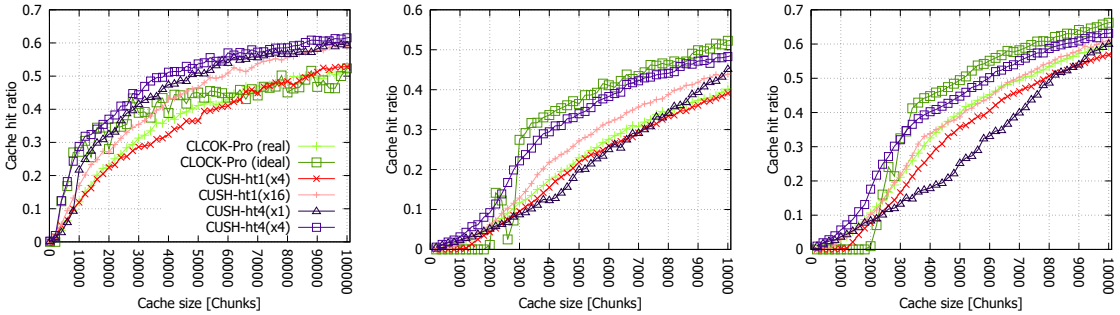
while the ideal policy with collision resolution achieves the best performance.

Figure 5.7 shows the cases when the size of cacheable chunks L is 1.5 KB. We can find the adverse effect of LOOP as some policies have no cache hit happen until the cache size exceeds a certain value. The policies that are weak to LOOP are disturbed by LOOP when the cache size is less than a certain value (e.g., 30,000 when $\alpha = 1.4$); however, CUSH achieves cache hits in the same condition. Thus, these results show CUSH can adapt to access patterns in network traffic. CUSH is also a good approximation of CLOCK-Pro. In fact, the performance of CUSH is comparable to that of CLOCK-Pro.

5.5.2 Evaluation of the Influence of Ghost Caches to LOOP-resistant Property

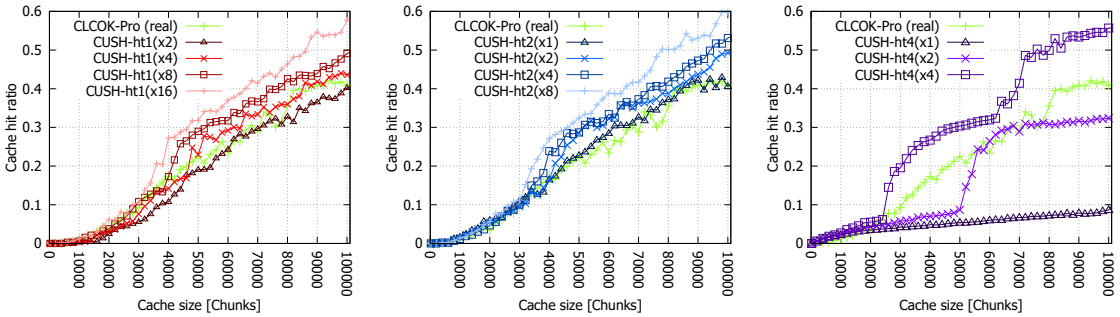
Then, we analyze the influence of the difference in the implementation of a ghost cache. The LOOP-resistant property is improved by increasing ghost caches and decreasing the hash-collision probability as mentioned in Section 5.4.3. Thus, we compare the policies shown in Table 5.1. CUSH-ht$b$(xk) denotes the implementation of CUSH with a hash-table that contains $k \times n$ entries and assigns b-bits per entry. For example, CUSH used in the previous section corresponds to CUSH-ht1(x4). Ghost caches increase as k becomes larger. Increasing b reduces the collision probability. `CLOCK-Pro(real)` and `CLOCK-Pro(ideal)` are the same as used in the previous section.

In the evaluation, we adopt the synthetic workloads in units of chunks described previously according to the purpose of analysis the LOOP-resistant properties. Figure 5.8 shows the hit rates for the synthetic workloads where $L = 15K$ and α changes from 1.0 to 1.4. Figure 5.9 compares the hit rates for the synthetic workloads with $L = 1.5K$ and $\alpha = 1.2$ among different implementations



(a) $\alpha = 1.0, L = 15\text{K}$ (b) $\alpha = 1.2, L = 15\text{K}$ (c) $\alpha = 1.4, L = 15\text{K}$

Figure 5.8: Effect of Implementation of Ghost Cache on LOOP-Resistant Property



(a) CUSH-ht1 (b) CUSH-ht2 (c) CUSH-ht4

Figure 5.9: Effect of the Amount of Ghost Caches on LOOP-Resistant Property (Using Synthetic Trace in Units of Chunks with$\alpha = 1.2$ and $L = 1.5\text{KB}$)

of a ghost cache. The larger k is, the better performance becomes typically. On the other hand, when $k = 1$ or 2, the hit rates with $b = 4$ is inferior to the hit rates with $b = 1$ or 2. This indicates a part of LOOP becomes accidentally a hot chunk when the number of ghost caches are too small to detect the repetition part of LOOP. This is because a high collision probability (i.e., small b) causes a cold chunk to turn into a hot chunk randomly. In fact, the hit rates of CUSH-ht4(x4) are best because this case fulfills both a low collision probability and many ghost caches enough to enable collision-free detection of a part of LOOP. Thus, we can enhance the LOOP-resistant properties by only adding bits to each entry without a significant change to the data structure of CUSH.

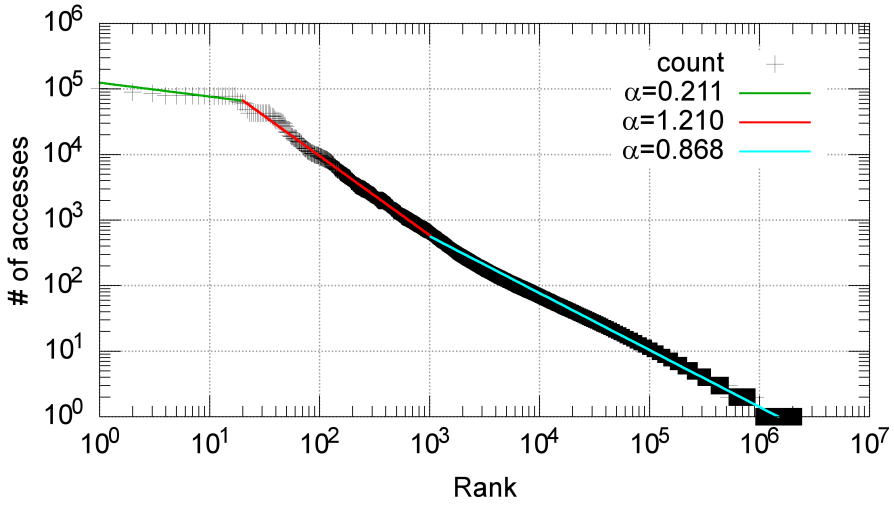


Figure 5.10: Popularity Distribution of a Real Trace

Table 5.2: Number of Chunks Per Second[pck/sec]

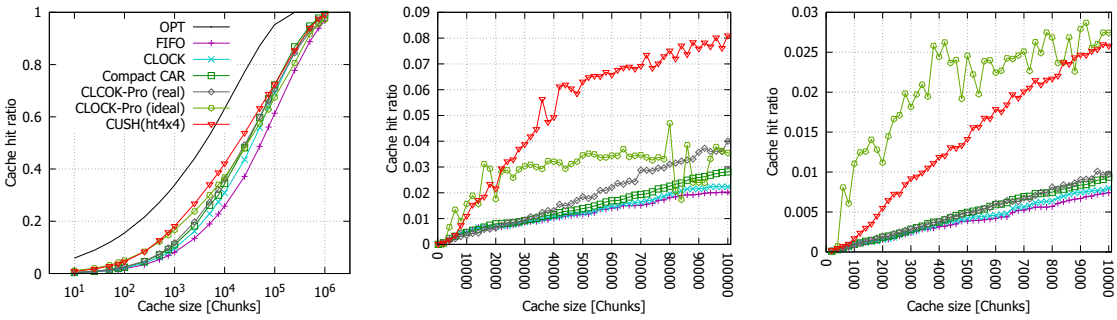
Chunk size	1.5KB	15KB	60KB
SD(600[kbps/min])	50	5	1.25
HD(1.2[Mbps/min])	100	10	2.50

5.5.3 Cache Hit Rate with Real Traffic Trace

To justify the results using synthetic traffic, we also use the real traffic traces. We collected traces of VoD (e.g, YouTube, DailyMotion, and NicoVideo) from a network gateway at Osaka University campus. The traces are gathered from July 26th 2013 to February 26th 2015. The number of unique content is 1,451,558; the number of content requested at least twice is 381,527; and the number of total accesses is 3,378,925. The popularity distribution of the real traffic trace follows the Zipf-like distribution, as depicted in Fig. 5.10. We also show the statistics of the real traffic traces in units of chunks in Table 5.3. The inter arrive time for chunks is assumed to be constant according to Table 5.2, which is determined by the statistics of our observed real traffic.

Table 5.3: Statistics of Workloads in Units of Chunks

L	# of total accesses	# of observed unique chunks	# of chunks requested at least twice
15 K	14,557,548	5,321,617	552,631
60 K	16,606,810	8,006,084	1,769,759



(a) Trace in Units of Content (b) Trace in Units of Chunks (L = 15KB) (c) Trace in Units of Chunks (L = 60KB)

Figure 5.11: Results for Real Traces

Figure 5.11 presents the simulation results. Here, we use CUSH-ht4(x4) to exhibit the benefits of an enhanced ghost cache. In the real environment, it is important to deal with access patterns caused by the volatility of popularity such as CORRELATED-REFERENCE and FICKLE-INTEREST. We can find that CUSH achieves higher hit rates than the others in the results for workloads in units of content shown in Fig. 5.11(a), which reveals the SCAN-resistant properties of CUSH. Figures 5.11(b) and 5.11(c) show the results for workloads in units of chunks, which contain LOOP. As `CLOCK-Pro(real)` is as bad as simple policies, CLOCK-Pro is difficult to cope with the changing popularity. CUSH outperforms the other policies except `CLOCK-Pro(Ideal)`. Thus, CUSH has the properties to cope with four access patterns discussed in Section 5.3.1, and therefore can adapt to real network traffic.

Table 5.4: Space Complexity of Cache Replacement Algorithm's Overhead

Policies	Cache management [bit]	Ghost cache [bit]	Lookup table for ghost cache [bit]
FIFO	$O(\log n)$	-	-
LRU_{DLL}	$O(n \log n)$	-	-
CLOCK	$O(n)$	-	-
CAR (with LRU_{DLL})	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Compact CAR	$O(n)$	$O(n)$	$O(n \log n)$
CLOCK-Pro	$O(n)$	$O(n)$	$O(n \log n)$
CUSH	$O(n)$	$O(kn)$	-

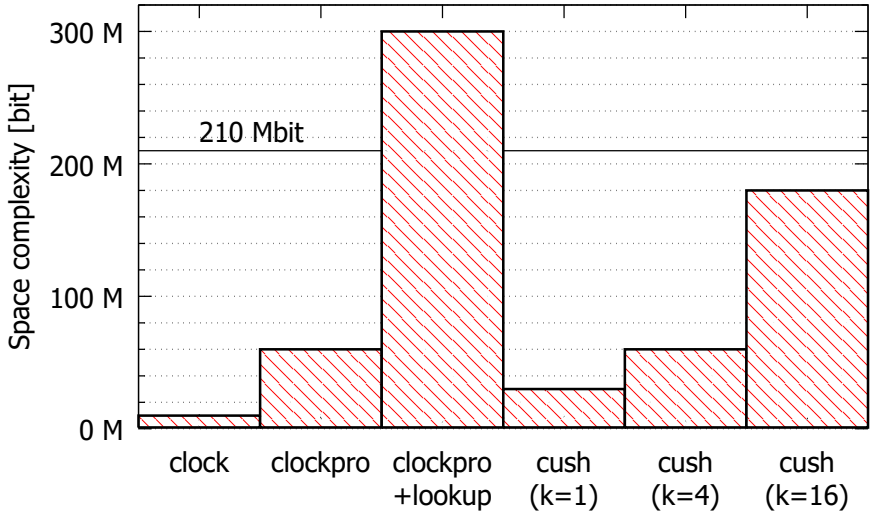


Figure 5.12: Space Complexities of CLOCK, CLOCK-Pro, and Our Proposal When $n = 10^7$

5.5.4 Analysis on Space and Time Complexities

Space Complexity

We analyze the space complexity of some cache replacement policies to elucidate CUSH is scalable. There are three types of memory costs: a control information to manage actually cached entries, a control information for a ghost cache, and the additional cost of lookup table for a ghost cache.

Table 5.4 summarizes the results of calculation with the big O notation. Our analysis compares seven algorithms: FIFO, LRU, CLOCK, CAR, Compact CAR, CLOCK-Pro, and CUSH. Because

Table 5.5: Average hand movement count of CLOCK-based policies

Cache hit rate	n	Average per access				Average per miss			
		CLOCK	CLOCK-Pro	CUSH	Compact CAR	CLOCK	CLOCK-Pro	CUSH	Compact CAR
0.0–0.1	10	0.98	4.33	1.17	2.91	1.06	3.79	1.35	3.25
0.1–0.2	32	0.92	8.05	1.09	2.78	1.11	6.29	1.41	3.46
0.2–0.3	100	0.84	13.82	0.99	2.52	1.14	9.52	1.45	3.52
0.3–0.4	317	0.74	22.44	0.86	2.21	1.17	13.33	1.47	3.56
0.4–0.5	1000	0.65	32.26	0.71	1.85	1.20	16.11	1.47	3.68
0.5–0.6	3163	0.55	62.34	0.58	1.52	1.25	25.20	1.51	3.73
0.6–0.7	10000	0.45	618.03	0.43	1.17	1.32	190.44	1.51	3.78
0.7–0.8	31623	0.34	3402.53	4.18	0.83	1.46	734.00	20.79	3.86
0.8–0.9	100000	0.23	32.68	0.65	0.40	1.73	4.20	5.14	3.20

there are several ways to implement LRU, we consider the method using a doubly-linked list (denoted by LRU$_{DLL}$) in this analysis. Although an additional cost of a lookup table depends on how to implement the lookup table, we assume the lookup table for ghost caches is implemented as a hash-table without collision resolution and its additional memory cost is $O(x \log x)$ when the number of ghost caches is x. We also define the following notations and variables. n is the number of cache entries. The number of ghost cache entries is k times of n in CUSH.

FIFO requires only a pointer to remember the head of the queue. The pointer requires at least $\lceil \log n \rceil$ [bit] to identify n individual entries; therefore, its space complexity is $O(\log n)$. To implement LRU$_{DLL}$, it is necessary to maintain a sorted doubly-linked list, where each entry has two pointers. Thus, LRU$_{DLL}$ requires $O(n \log n)$ memory overhead. The space complexity of CLOCK is $O(n)$ to store n R-bits.

The cache replacement policies using a cache history information (CAR, CLOCK-Pro, Compact CAR, and CUSH) add a memory cost of ghost caches in addition to a cost to manage actually cached entries. The implementation of CAR is based on doubly-linked lists and the number of ghost caches is n; therefore, the costs for cache management and ghost caches are $O(n \log n)$.

Compact CAR and CLOCK-Pro use two CLOCK lists with n entries; therefore, the memory costs for cache management and ghost caches are $O(n)$. In contrast to these policies that add the overhead cost to a lookup table, our proposal is free from additional cost of a lookup table. CUSH requires the cost of cache management that is equivalent to that of CLOCK. CUSH furthermore can

enlarge the capacity of ghost caches k times.

We also calculate the actual memory cost added by CLOCK, CLOCK-Pro, and our proposal when the number of cache entries n is 10 million according to the previous literature [29, 16]. Figure 5.12 shows the results of the calculation. A policy based on a CLOCK list typically requires a memory overhead of several bits per chunk. CLOCK consumes 10 Mbits because it assigns only R-bit to each chunk. CLOCK-Pro has a modified CLOCK list, which contains $2n$ entries and assigns three bits to each entry (R-bit, H-bit, and a bit called a test flag), and so consumes 60 Mbits. CUSH assigns two bits to n entries and has n-bits hash-tables; therefore it consumes 30 Mbits when $k = 1$.

When taking account of the costs for ghost caches, CLOCK-Pro additionally consumes 240 Mbit for looking-up ghost caches. Thus, the total cost becomes 300 Mbit (denoted by "`clockpro+lookup`" in Figure 5.12). This cost is prohibitive because of the severe constraints of fast memory enough to be employed in an ICN router such as SRAM, whose available size is 210 Mbit [30]. On the other hand, CUSH is free from additional cost of a lookup table. In addition, CUSH can conserve the cost when enhancing a hash-table for ghost caches. Even if $k = 16$, the total memory overhead cost of CUSH is 180 Mbit. Thus, CUSH is a low-overhead and scalable cache replacement policy that satisfies the memory requirements of an ICN router.

Time Complexity

We analyze the time complexity in this section. We focus on the policies based on a CLOCK list because of their low space and time complexities enough to be installed into an ICN router. The CLOCK-based policy decides whether the replacement process continues or terminates every hand movement; therefore, we can estimate its time complexity by counting the hand movement.

Table 5.5 shows the number of hand movement. We evaluated the four policies: CLOCK, CLOCK-Pro, CUSH and Compact CAR. The number of hand movement intuitively depends on a cache hit rate because the process continues when there are many entries with $R = 1$ caused by many cache hits. For clarity, we use the synthetic trace with $\alpha = 1.0$, where a hit rate increases in proportion to $\log n$. We show both the average counts for the all accesses and the average counts for

the cache misses because a cache miss begins the hand movement process. We omit an evaluation of the worst case because the worst-case complexity is obviously $O(n)$.

We find that the time complexities of CLOCK-based policies except for CLOCK-Pro are constantly low. The average hand movement count of CLOCK per miss is less than two. The count of CLOCK-Pro is at most hundreds or thousands times as many as that of CLOCK. This is because CLOCK-Pro has three hands and the hands pass the chunks unrelated to each of them (e.g., $\mathrm{HAND}_{\mathrm{cold}}$ and $\mathrm{HAND}_{\mathrm{hot}}$ ignore all ghost cache entries). Although Compact CAR also has two types of chunks, the average hand movement count of Compact CAR is constantly less than four because the two types of chunks are maintained separately.

CUSH can also achieve the hand movement count equivalent to that of CLOCK in almost all the cases. The data structure of CUSH is devised to reduce the extremely large overhead of CLOCK-Pro by maintaining ghost caches separately. However, when the hit rate is about 0.8, the count becomes relatively large as well as CLOCK-Pro. If this computational cost is prohibitive and appears in an actual environment, it may be required to maintain the rest two types of chunks separately just as Compact CAR does although the cost will increase under the other conditions.

5.6 Summary

We proposed a novel cache replacement algorithm named CUSH which would be an important component in the design of a resource-restricted ICN router. CUSH outperforms compared to conventional cache replacement algorithms in terms of cache hit rates and reduction of memory usage. In detail, the proposed algorithm achieves cache hits against the traffic traces with access patterns that simple conventional algorithms hardly cause any hits. CUSH can enhance the mechanism to cache a content item worth caching without expensive additional cost, which is important to deal with various traffics in ICN.

Chapter 6

Conclusion

With the growing recognition that the Internet is reaching its limits, ICN has been drawing attention around the world. This promising architecture natively supports various functions including multicasting, in-network caching, and security with respect to content rather than a connection to end devices.

Chapter 2 presented the architecture and implementation of an OpenFlow-based ICN. OpenFlow, which is gaining an increasing presence, is expected to become the intermediary for the deployment of ICN devices. Our examination demonstrated the feasibility and possibility for the deployment, and the ability to simultaneously study challenges such as routing and caching strategies by using hierarchically structured hashes of content names. The trade-off between hash collision probabilities and the number of components that can be handled in names will be investigated once products and a framework are released.

In Chapter 3, we demonstrated the feasibility of ICN by designing concrete ICN router hardware and evaluating its performance. Needless to say, the feasibility of an ICN router is essential to realize ICN. In addition, accurate estimates of actual performance of an ICN router helps to perform all sorts of network-level simulations of ICN. We addressed these problems by proposing a CAM-based ICN router architecture. We proposed NLE, which consists of many small CAMs and DLB-BF and reduces costs, and ICE, which supports adaptive caching: we have shown the entire design of a ICN router in this Chapter. We also performed a basic theoretical analysis of the expected

throughput and memory consumption of the ICN router.

We split the monolithic CAM into smaller parts to make NLE feasible. A significant challenge for our architecture is to scale the number of routing entries that highly affect the memory capacity in a router. There are no existing TCAMs with more than 100 Mbits of memory capacity. In addition, the power consumption of a TCAM can be approximated as 1 kW/Mbit; the power requirements of our router, which needs at least 3.2 Gbits of CAM, can rise to more than 3 kW; however, even a 1 kW power requirement is beyond the capacity of any existing implementation by several orders of magnitude. These challenges can be addressed by using multiple small CAMs. A sufficient number of small CAMs can dramatically reduce the cost for capacity and power. Employing 32 units of small CAMs instead of 1 large monolithic CAM, for example, allows NLE to be implemented on an existing 100-Mbit TCAM chip. As a rough estimate, the distributed CAMs require only $1/32$ of the power of a large single CAM. We can adjust the number of CAMs running in parallel to achieve the desired trade-off between the power consumption and the lookup performance.

FIB is required to handle websites, the number of which is approaching 1 billion according to a survey in [45]. The line-speed (40 Gbps) traffic, whose average round-trip delay time (RTT) is $\mathrm{RTT} = 100\mathrm{ms}$, imposes 2 million entries per port on PIT. Even if the effect of ICE is maximized, the number of chunks that would be stored in CS for 10 million entries is equivalent to the number of files accessed per day in terms of city-scale traffic [37].

In the future, it seems that such numbers increase rapidly along with the number of content items. Our architecture can easily scale with this increase in the number of names by installing additionally distributed CAMs according to the number of entries required for NLE. A simple solution — ignoring lookup time — is to partially replace CAM with a hash table; however, the limits of the system can be relaxed without sacrificing speed because we can use not only 16 T / cell TCAMs but also 10 T / cell BCAMs, and we expect exponential growth in the capacity and availability of feasible memory.

In Chapter 4 and Chapter 5, we argued that the conventional cache replacement algorithms cannot be directly applied to the design of a high performance ICN core router, and proposed two cache replacement algorithms. Chapter 4 proposed a novel cache replacement algorithm named Compact

CAR which would be an important component in the design of a high performance ICN core router. Compact CAR outperforms compared to conventional cache replacement algorithms in terms of cache hit rate and memory usage in the design of ICN router. In detail, the proposed algorithm can achieve the same cache hit rate with only one-tenth of memory usages that simple conventional algorithms consume. In addition, the cache hit rate by the proposed algorithm is only 10% less than the optimal case over the various simulation scenarios. In particular, the difference becomes negligible when we use real traffic traces whose RD values are similar to the cache size. This result provides a clue that a high cache hit rate can be achieved if the cache size adaptively changes according to the distribution of RD value in real traffic. Furthermore, Compact CAR can dynamically adapt itself to the network environment whose traffic access patterns change dynamically, which is important to deal with various types of traffic in ICN.

Chapter 5 provided a cache replacement algorithm named CUSH which would be an important component in the design of a resource-restricted ICN router. Unlike Compact CAR, which is inappropriate for LOOP traffic pattern and is designed without the constraints of the memory cost of a lookup table, CUSH outperforms compared to conventional cache replacement algorithms in terms of cache hit rate and reduction of memory usage. In detail, the proposed algorithm achieves cache hits against the traffic traces that simple conventional algorithms hardly cause any hits. CUSH can enhance a caching mechanism in a way that it identifies content items to be cached without expensive additional cost, which is important to deal with various traffics in ICN.

ICN has been researched nearly 10 years and it may be the time to consider its deployment issue in Internet-scale where the design of a high performance ICN core router becomes critical. We believe that our proposal plays a key role in the design of such a high performance ICN core router in the near future.

Bibliography

[1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, and R. L. Braynard, "Networking named content," in *Proceedings of the ACM CoNEXT 2009*, pp. 1–12, December 2009.

[2] 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," *IEEE Communications Surveys Tutorials*, vol. 16, no. 2, pp. 1024–1049, February 2014.

[3] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, K. Claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, and E. Yeh, "Named data networking (NDN) project," pp. 1–24, October 2010. [Online]. Available: http://named-data.net/techreport/TR001ndn-proj.pdf

[4] N. Fotiou, P. Nikander, D. Trossen, and G. C. Polyzos, "Developing information networking further: From PSIRP to PURSUIT," in *Proceedings of the 7th International ICST Conference on Broadband Communications,Networks, and Systems*, pp. 1–13, October 2010.

[5] T. Levä, J. Gonçalves, R. J. Ferreira *et al.*, "Description of project wide scenarios and use cases," pp. 1–99, February 2011. [Online]. Available: http://www.sail-project.eu/wp-content/uploads/2011/02/SAIL_D21_Project_wide_Scenarios_and_Use_cases_Public_Final.pdf

[6] B. Ahlgren, C. Dannewitz, C. Imbrenda, D. Kutscher, and B. Ohlman, "A survey of information-centric networking," *IEEE Communications Magazine*, vol. 50, no. 7, pp. 26–36, July 2012.

[7] V. Jacobson, D. K. Smetters, N. H. Briggs, M. F. Plass, P. Stewart, J. D. Thornton, and R. L. Braynard, "VoCCN: Voice-over Content-Centric Networks," in *Proceedings of the 2009 workshop on Re-architecting the internet*, pp. 1–6, December 2009.

[8] D. Chang, J. Suh, H. Jung, T. T. Kwon, and Y. Choi, "How to realize CDN Interconnection (CDNI) over OpenFlow?" in *Proceedings of the 7th International Conference on Future Internet Technologies*, pp. 29–30, September 2012.

[9] N. Melazzi, A. Detti, G. Mazza, G. Morabito, S. Salsano, and L. Veltri, "An OpenFlow-based testbed for information centric networking," in *Proceedings of the Future Network Mobile Summit 2012*, pp. 1–9, July 2012.

[10] I. Carvalho, F. Faria, E. Cerqueira, and A. Abelem, "ContentFlow: An introductory routing proposal for Content Centric Networks using Openflow," pp. 1–2, June 2012.

[11] M. Varvello, D. Perino, and J. Esteban, "Caesar: a content router for high speed forwarding," in *Proceedings of the 2nd edition of the ICN workshop on Information-centric networking*, pp. 73–78, August 2012.

[12] W. You, B. Mathieu, P. Truong, J. Peltier, and G. Simon, "DiPIT: A distributed bloom-filter based PIT table for CCN nodes," in *Proceedings of the 21st ICCCN 2012*, pp. 1–7, July 2012.

[13] Y. Wang, T. Pan, Z. Mi, H. Dai, X. Guo, T. Zhang, B. Liu, and Q. Dong, "NameFilter: Achieving fast name lookup with low memory cost via applying two-stage bloom filters," in *Proceedings of the IEEE INFOCOM 2013*, pp. 95–99, April 2013.

[14] Y. Wang, K. He, H. Dai, W. Meng, J. Jiang, B. Liu, and Y. Chen, "Scalable name lookup in NDN using effective name component encoding," in *Proceedings of the IEEE 32nd International Conference on Distributed Computing Systems 2012*, pp. 688–697, June 2012.

[15] H. Dai, B. Liu, Y. Chen, and Y. Wang, "On pending interest table in Named Data Networking," in *Proceedings of the ACM/IEEE 8th Symposium on Architectures for Networking and Communications Systems 2012*, pp. 211–222, October 2012.

[16] Y. Wang, Y. Zu, T. Zhang, K. Peng, Q. Dong, B. Liu, W. Meng, H. Dai, X. Tian, Z. Xu, H. Wu, and D. Yang, "Wire speed name lookup: a GPU-based approach," in *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pp. 199–212, April 2013.

[17] W. K. Chai, D. He, I. Psaras, and G. Pavlou, "Cache "less for more" in information-centric networks," *Computer Communications*, vol. 36, no. 7, pp. 758–770, May 2012.

[18] A. Safari Khatouni, M. Mellia, L. Venturini, D. Perino, and M. Gallo, "Performance comparison and optimization of ICN prototypes," in *Proceedings of 2016 IEEE GLOBECOM*, pp. 1–6, December 2016.

[19] D. Rossi and G. Rossini, "Caching performance of content centric networks under multi-path routing (and more)," Telecom ParisTech, Tech. Rep., July 2011.

[20] J. Ran, N. Lv, D. Zhang, Y. Ma, and Z. Xie, "On performance of cache policies in named data networking," in *Proceedings of the International Conference on Advanced Computer Science and Electronics Information 2013*, pp. 668–671, July 2013.

[21] L. Wang, S. Bayhan, and J. Kangasharju, "Optimal chunking and partial caching in information-centric networks," *Computer Communications*, vol. 61, no. 1, pp. 48–57, May 2015.

[22] S. Arianfar, P. Nikander, and J. Ott, "Packet-level caching for information-centric networking," Finnish ICT SHOK, Tech. Rep., June 2010.

[23] A. Ooka, S. Ata, T. Koide, H. Shimonishi, and M. Murata, "A deployment of Content-Centric Networking by using OpenFlow networks," *IEICE Technical Report (IN2013-19)*, vol. 113, no. 36, pp. 43–48, May 2013.

[24] ——, "OpenFlow-based Content-Centric Networking architecture and router implementation," in *Proceedings of Future Network and MobileSummit 2013 Conference*, pp. 1–10, Lisboa, July 2013.

[25] ONF Market Education Committee, "Software-Defined Networking: The new norm for networks," ONF, Tech. Rep. ONF White Paper, April 2012. [Online]. Available: https://www.opennetworking.org/images/stories/downloads/white-papers/wp-sdn-newnorm.pdf

[26] A. Ooka, S. Ata, K. Inoue, and M. Murata, "Hardware design and evaluation of a high-speed ccn router using cam," *IEICE Technical Report (IN2013-161)*, vol. 113, no. 473, pp. 105–110, March 2014.

[27] ——, "Design of a high-speed content-centric-networking router using content addressable memory," in *Proceedings of IEEE INFOCOM 2014 Workshop on Name-Oriented Mobility*, pp. 1–6, Toronto, April 2014.

[28] A. Ooka, "Hardware design and evaluation of CAM-based high-speed CCN router," Master's thesis, Graduate School of Information Science and Technology, Osaka-University, February 2013.

[29] A. Ooka, S. Ata, K. Inoue, and M. Murata, "High-speed design of conflict-less name lookup and efficient selective cache on CCN router," *IEICE Transactions on Communications*, vol. E98-B, no. 04, pp. 607–620, April 2015.

[30] D. Perino and M. Varvello, "A reality check for Content Centric Networking," in *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, pp. 44–49, August 2011.

[31] S. Arianfar, P. Nikander, and J. Ott, "On content-centric router design and implications," in *Proceedings of the ACM Re-Architecting the Internet Workshop*, pp. 1–6, November 2010.

[32] P. Bosshart, G. Gibb, H.-S. Kim, G. Varghese, N. McKeown, M. Izzard, F. Mujica, and M. Horowitz, "Forwarding metamorphosis: fast programmable match-action processing in hardware for SDN," in *Proceedings of the ACM SIGCOMM 2013*, pp. 99–110, August 2013.

[33] A. Ooka, S. Ata, and M. Murata, "A proposal and evaluation of cache replacement policy for the implementation of ICN router," *Technical Committee on Information-Centric Networking (ICN)*, pp. 1–10, July 2015.

[34] Atsushi Ooka, Suyong Eum, Shingo Ata and Masayuki Murata, "Compact CAR: Low-overhead cache replacement policy for an ICN router," submitted to *International Journal of Communication Systems*, pp. 1–26, February 2017.

[35] ——, "A proposal and evaluation of feasible cache replacement policy for ICN based on CLOCK-Pro," *Technical Committee on Information-Centric Networking (ICN)*, pp. 1–14, December 2016.

[36] ——, "Scalable cache component in ICN adaptable to various network traffic access patterns," submitted to *IEICE Transactions on Communications*, pp. 1–12, February 2017.

[37] F. Guillemin, B. Kauffmann, S. Moteau, and A. Simonian, "Experimental analysis of caching efficiency for YouTube traffic in an ISP network," in *Proceedings of the 25th International Teletraffic Congress*, pp. 1–9, September 2013.

[38] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, pp. 187–200, March 2004.

[39] S. Jiang, F. Chen, and X. Zhang, "CLOCK-Pro: An effective improvement of the CLOCK replacement," in *Proceedings of the USENIX 2005*, pp. 323–336, April 2005.

[40] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, "Impact of traffic mix on caching performance in a content-centric network," in *Proceedings of the IEEE Conference on Computer Communications 2012*, pp. 310–315, March 2012.

[41] "CCNx," PARC, 2014. [Online]. Available: http://www.ccnx.org/

[42] H. Song, F. Hao, M. Kodialam, and T. V. Lakshman, "IPv6 lookups using distributed and load balanced bloom filters for 100Gbps core router line cards," in *Proceedings of the IEEE INFOCOM 2009*, pp. 2518–2526, April 2009.

[43] "CCNx 1.0 protocol specifications roadmap," PARC, November 2013. [Online]. Available: http://www.ietf.org/mail-archive/web/icnrg/current/pdfZyEQRE5tFS.pdf

[44] S. Iyer, R. Kompella, and N. McKeown, "Designing packet buffers for router linecards," *IEEE/ACM Transactions on Networking*, vol. 16, no. 3, pp. 705–717, June 2008.

[45] "netcraft," December 2013. [Online]. Available: http://www.netcraft.com/

[46] N. Megiddo and D. S. Modha, "ARC: a self-tuning, low overhead replacement cache," in *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, pp. 115–130, March 2003.

[47] T. Johnson and D. Shasha, "2Q: a low overhead high performance buffer management replacement algorithm," in *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 439–450, September 1994.

[48] S. Jiang and X. Zhang, "Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance," *IEEE Transactions on Computers*, vol. 54, no. 8, pp. 939–952, August 2005.

[49] F. J. Corbato, "A paging experiment with the Multics system," DTIC Document, Tech. Rep., May 1968.

[50] J. Wang, "A survey of web caching schemes for the Internet," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 5, pp. 36–46, October 1999.

[51] K.-Y. Wong, "Web cache replacement policies: a pragmatic approach," *IEEE Network*, vol. 20, no. 1, pp. 28–34, January 2006.

[52] A.-M. K. Pathan and R. Buyya, "A taxonomy and survey of content delivery networks," University of Melbourne Grid Computing and Distributed Systems Laboratory, Tech. Rep., February 2007.

[53] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., and J. Emer, "High performance cache replacement using re-reference interval prediction (RRIP)," *ACM SIGARCH Computer Architecture News*, vol. 38, no. 3, pp. 60–71, June 2010.

[54] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and Zipf-like distributions: evidence and implications," in *Proceedings of IEEE INFOCOM'99*, vol. 1, pp. 126–134, March 1999.

[55] L. A. Belady, "A study of replacement algorithms for a virtual-storage computer," *IBM Systems Journal*, vol. 5, no. 2, pp. 78–101, June 1966.

[56] G. Zhang, Y. Li, and T. Lin, "Caching in information centric networking: A survey," *Computer Networks*, vol. 57, no. 16, pp. 3128–3141, 2013.

[57] M. Zhang, H. Luo, and H. Zhang, "A survey of caching mechanisms in Information-Centric Networking," *IEEE Communications Surveys Tutorials*, vol. 17, no. 3, pp. 1473–1499, April 2015.