A High-speed NDN Forwarding Engine on a Commercial Off-the-shelf Computer

Submitted to
Graduate School of Information Science and Technology
Osaka University

January 2019

JUNJI TAKEMASA

List of Publications

Journal Papers

1. Kaito Ohsugi, Junji Takemasa, Yuki Koizumi, Toru Hasegawa and Ioannis Psaras, "Power Consumption Model of NDN-Based Multicore Software Router Based on Detailed Protocol Analysis," *IEEE Journal on Selected Area in Communications*, vol. 34, no. 5, pp. 1631–1644, May. 2016.
2. Junji Takemasa, Yuki Koizumi and Toru Hasegawa, "Is Caching a Key to Energy Reduction of NDN Networks?," *IEICE Transactions on Communications*, vol. 99, no. 12, pp. 2489–2497, Dec. 2016.
3. Junji Takemasa, Yuki Koizumi and Toru Hasegawa, "Lightweight Cache Admission Algorithm for Fast NDN Software Routers,"*to appear in IPSJ Journal of Information Processing*.

Refereed Conference Papers

1. Junji Takemasa, Yuki Koizumi, Toru Hasegawa, and Ioannis Psaras, "On Energy Reduction and Green Networking Enhancement due to In-Network Caching," in *Proceedings of IEEE MASS Workshop on Content-Centric Networking*, pp. 513–518, Oct. 2015.
2. Kosuke Taniguchi, Junji Takemasa, Yuki Koizumi and Toru Hasegawa, "Poster: A Method for Designing High-Speed Software NDN Routers," in *Proceedings of ACM Conference on Information-Centric Networking, Poster Session*, pp. 203–204, Sep. 2016.
3. Junji Takemasa, Kosuke Taniguchi, Yuki Koizumi and Toru Hasegawa, "Identifying Highly Popular Content Improves Forwarding Speed of NDN Software Router," in *Proceedings of IEEE GLOBECOM Workshop on Information Centric Networking Solutions for Real World Applications*, pp. 1–6, Dec. 2016.
4. Junji Takemasa, Yuki Koizumi and Toru Hasegawa, "Toward an Ideal NDN Router on a

Commercial Off-the-shelf Computer," in *Proceedings of ACM Conference on Information-Centric Networking, Full-Paper Session*, pp. 43–53, Sep. 2017.

5. Junji Takemasa, Yuki Koizumi and Toru Hasegawa, "Analysis of Mutual Exclusion Overhead of NDN Packet Forwarding on Multi-core Software Router," in *Proceedings of ACM Conference on Information-Centric Networking, Poster Session*, Sep. 2018.

Non-Refereed Technical Papers

1. Junji Takemasa, Yuki Koizumi, Toru Hasegawa, "A Study on a Power Trade-off between Cache Processing Overheads and Traffic Reduction in CCN Networks," in *IEICE Communication Society Conference*, B-7-45, Sep. 2014 (in Japanese).
2. Junji Takemasa, Yuki Koizumi, Toru Hasegawa and Ioannis Psaras, "A Study on Energy Reduction Effects due to In-Network Caching in ICN Networks," in *IPSJ Kansai Society*, E-17, Sep. 2015 (in Japanese).
3. Junji Takemasa, Kosuke Taniguchi, Yuki Koizumi, and Toru Hasegawa, "Cache Admission for Improving Forwarding Speed of NDN Software Router," in *IEICE Technical Report*, vol. 116, no. 361, IN2016-83, pp. 97–102, Dec. 2016 (in Japanese).
4. Junji Takemasa, Yuki Koizumi, Toru Hasegawa, "A Study on Power Consumption Trade-Off Due to Caching in NDN Networks," in *IEICE General Conference*, B-7-5, Mar. 2016 (in Japanese).
5. Junji Takemasa, Yuki Koizumi, Toru Hasegawa, "A Study on Power Reduction due to Caching in an NDN Access Network," in *IEICE Communication Society Conference*, B-7-25, Sep. 2016 (in Japanese).
6. Junji Takemasa, Yuki Koizumi, and Toru Hasegawa, "Bottleneck Analysis on Packet Forwarding of an NDN Software Router," in *IEICE Technical Report*, vol. 116, no. 485, IN2016-83, pp. 97–102, Mar. 2017 (in Japanese).
7. Junji Takemasa, Kosuke Taniguchi, Yuki Koizumi, Toru Hasegawa, "On a Bottleneck Analysis on Packet Forwarding of an NDN Software Router," in *IEICE General Conference*, B-7-74, Mar. 2017 (in Japanese).
8. Junji Takemasa, Yuki Koizumi, and Toru Hasegawa, "An Analysis on Data Structures and Algorithms for FIB of a Fast PC-based NDN Router," in *IEICE Technical Report*, vol. 116, no. 205, IN2017-25, pp. 13–18, Sep. 2017 (in Japanese).
9. Junji Takemasa, Kosuke Taniguchi, Yuki Koizumi, Toru Hasegawa, "On an Analysis on

Increase in Packet Processing Latency Due to Parallelizing a PC-based NDN Router," in *IEICE Communication Society Conference*, B-7-27, Sep. 2017 (in Japanese).

10. Junji Takemasa, Yuki Koizumi, and Toru Hasegawa, "An Analysis on Cache Algorithm for CS of Fast PC-based NDN Router," in *IEICE Technical Report*, vol. 117, no. 460, IN2017-130, pp. 243–248, Mar. 2018 (in Japanese).
11. Junji Takemasa, Yuki Koizumi, Toru Hasegawa, "A Study on Fast Cache Algorithm on PC-based NDN Router," in *IEICE General Conference*, B-7-36, Mar. 2018 (in Japanese).
12. Junji Takemasa, Yuki Koizumi, Toru Hasegawa, "A Study on Maintaining Consistency of PIT States in NDN Parallel Packet Processing," in *IEICE Communication Society Conference*, B-7-11, Sep. 2018 (in Japanese).

Preface

A software router, which is built on a hardware platform based on a commercial off-the-shelf (COTS) computer, becomes feasible because of recent advances in multi-core CPUs and fast networking technologies for COTS computers. Fast IP packet forwarding has been a research issue for a long time and hardware architecture designed based on a CPU device with fast memory devices like static random access memory (SRAM) devices shows that storing data structures for IP forwarding on SRAM devices enables fast IP forwarding for about 5×10^4 IP prefixes. Successively, compact data structures have become a research issue to keep up with the increasing number of IP prefixes in the Internet. Most of the studies have addressed compact trie-based data structures like multi-bit trie data structures by replacing consecutive elements in a trie to a single element. Such efforts enable it to store increasing IP prefixes on latest SRAM devices.

Recently, studies on high-speed algorithms and compact data structures are being revisited due to the emergence of the new Internet architecture called Named Data Networking (NDN), wherein rich data, including large video data and small sensor data, is delivered by a single architecture. High-speed NDN packet forwarding is not trivial compared to IP packet forwarding in terms of memory space and time complexity because name-based forwarding and packet caching, which are key components of NDN, need much larger data structures than IP forwarding. The number of names in the NDN-based Internet is believed to be at least the same that of unique domain names, and the number as of December 2017 is about 2.1×10^8. Such a large number of name prefixes needs large memory space and heavy computation. In addition to packet forwarding, caching needs extra data structures for managing packets in a router's cache and computation for it. Handling variable-length and human-readable names and stateful forwarding also incur heavy computation. Heavy name-based forwarding and caching make high-speed NDN packet forwarding challenging.

The goal of this thesis is to present what a high-speed NDN forwarding engine on a COTS computer is supposed to be. In this thesis, the author designs a reference forwarding engine by selecting well-established high-speed techniques and then analyzes a state-of-the-art prototype

implementation to identify its performance bottlenecks. The microarchitectural analysis at the level of CPU pipelines and instructions provides us with two observations. The first observation is that the DRAM access latency due to accesses to forwarding tables, which include tables of name-based forwarding and caching, is one of bottlenecks for high-speed forwarding engines and the other one is that wasteful cache computation for unpopular Data packets is the rest of the bottlenecks. According to the first observation, the author designs prefetch algorithms for name-based forwarding and caching in order to to hide DRAM access latency. According to the second observation, the author designs a lightweight cache admission algorithm so that wasteful cache computations for unpopular Data packets are avoided with fast computation. The prototype of an NDN software router with the two techniques achieves a forwarding rate exceeding 40 million packets per second on a COTS computer.

The main contributions of this thesis are summarized as follows: First, as far as the author knows, this is the first study that summarizes observations found in existing studies to build high-speed NDN routers on computers and compiles the observations into a design guideline. According to the guideline, the author reveals that a hash table-based algorithm and a frequency-based cache admission algorithm are suitable for name-based forwarding and caching, respectively. Second, the author designs a prefetch algorithm for hiding most of the DRAM access latency for both name-based forwarding and caching. The prefetch algorithm reorders instruction executions for a pair of packets so that the DRAM access latency due to data fetches is successfully hidden by execution time of instructions independent of the data fetches. Third, the author carefully designs a lightweight frequency-based cache admission algorithm, Filter, on the basis of the results of the empirical measurement of an NDN software platform. As the results of the careful design, Filter consumes a few tens of CPU cycles in average with providing as high cache hit rate as ARC, which is one of the sophisticated cache eviction algorithms. Fourth, the author implements a proof-of-concept prototype of an NDN software router with the proposed two techniques and empirically proves that forwarding speed with the two techniques increases linearly as the number of CPU cores increases, whereas that without the two techniques does not increase so. A detailed analysis at the level of CPU pipelines and instructions reveal that most DRAM access latency are successfully hidden by the prefetch algorithm and that the number of CPU cycles of NDN packet processing is reduced by the Filter admission algorithm. Fifth, the author sheds light on power efficiency of NDN software routers as a positive effect of CPU cycle reduction at the different view point from forwarding speed improvement. By modeling how NDN packet processing consumes power of hardware devices on an NDN software router, the author reveals that reducing the number of CPU cycles spent for NDN packet processing contributes to power consumption reduction of an NDN software router.

Acknowledgments

My research activities would not have been achieved without many individuals.

First of all, I would like to express my greatest appreciation to my supervisor, Professor Toru Hasegawa, Graduate School of Information Science and Technology, Osaka University, for his invaluable support and guidance during my study at Osaka University. He has always been encouraging and provided insights to help me develop research ideas. I have always been impressed by how he simplifies complex issues based on practical considerations.

I would like to express my gratitude to the members of my thesis committee, Professor Masayuki Murata, Professor Takashi Watanabe, and Professor Teruo Higashino, Graduate School of Information Science and Technology, Osaka University, and Professor Morito Matsuoka, Cyber Media Center, Osaka University, for carefully reviewing this thesis and providing valuable comments.

I am deeply grateful to Associate Professor Yuki Koizumi, Graduate School of Information Science and Technology, Osaka University, for irreplaceable guidance and support. His apt comments have stimulated my daily research activities and helped me to make a breakthrough in research.

I am cordially thankful to Dr. Ioannis Psaras, Department of Electronic & Electrical Engineering, University College London, for valuable comments to raise the quality of my research.

I would like to thank Dr. Keiichiro Tsukamoto of Dwango Corporation, Yuto Nakai of NTT Communications Corporation, Kaito Ohsugi of KDDI Corporation, and Yuji Manaka of NTT Communications Corporation, for their direction during my bachelor and master coursework. Their direction also has promoted my daily research activities.

Special thanks are due to Kosuke Taniguchi, Yoji Yamamoto and Yasunaga Murai for their ideas and valuable discussions. They have strengthened the base of my researches and reinforced the value of this thesis.

I would like to extend my appreciation to the members of the Information Sharing Platform Laboratory. Especially, I am thankful to Ms. Tomoko Ueshima for her kind help. I thank Yasutaka Tara and Keita Hasegawa of past members of the Laboratory. I think that daily conversation with them has given me many inspirations. I also express my appreciation to all members of the Information Sharing Platform Laboratory, Graduate School of Information Science and Technology, Osaka University.

Finally, I would like to express my heartfelt appreciation to my father, mother, old brother, and little sister, for their warm-hearted help and support. With the help of them, I could concentrate on this work and complete it with an easy mind.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Background

A software router, which is built on a hardware platform based on a commercial off-the-shelf (COTS) computer, becomes feasible because of recent advances in multi-core CPUs and fast networking technologies for COTS computers. For notation simplicity, hereinafter, a COTS computer is simply referred to as a computer. Fast IP packet forwarding has been a research issue for a long time and hardware architecture designed based on a CPU device with fast memory devices like static random access memory (SRAM) devices [1, 2] shows that storing data structures for IP forwarding on SRAM devices enables fast IP forwarding for about 5×10^4 IP prefixes. Inspired by the success, fast IP forwarding on software routers is being studied to keep up with the increasing number of IP prefixes in the Internet. Dobrescu et al. [3] exploit parallelism by leveraging multiple CPU cores or servers. The success of forwarding engines, which the studies assume to be built on current COTS computers, shows feasibility of fast packet forwarding of software routers without using special devices like general-purpose computing on graphics processing unit (GPGPU) devices [4]. As a consequence, compact data structures have become a research issue and many trie-based data structures have been designed. Among them, Degermark et al. [5] designed multi-bit trie data structures by replacing consecutive elements in a trie to a single element. Such efforts enable it to store an increasing number of IP prefixes, e.g., 7×10^5 prefixes [6], on the latest SRAM devices. In summary, high-speed IP forwarding on COTS computers is an almost solved research question because IP prefixes are able to be stored on fast SRAM devices.

Recently, studies on high-speed algorithms and compact data structures are being revisited due to the emergence of the new Internet architecture called Named Data Networking (NDN) [7].

High-speed NDN packet forwarding is not trivial compared to IP packet forwarding in terms of space and time complexity because name-based forwarding and packet caching, which are key components of NDN, need much larger data structures than IP forwarding. The number of names in the NDN-based Internet is believed to be at least the same that of unique domain names and the number as of December 2017 is about 2.1×10^8 [8]. Such a large number of name prefixes needs large memory space and heavy computation. In addition to name-based forwarding, caching needs extra data structures for managing packets in a router's cache and computation for it. Handling variable-length and human-readable names and stateful forwarding also incur heavy computation.

Heavy name-based forwarding and caching result in poor performance of NDN forwarding on computers. The goal of this thesis is to identify what a high-speed NDN forwarding engine on a computer is supposed to be.

1.2 Approaches

In this thesis, the author identifies bottlenecks for high speed forwarding by carefully analyzing the existing NDN forwarding engine in terms of hardware and software constraints. First of all, the author carefully analyzes what is the most significant constraint among the following three ones: the computation capability of a CPU device, bandwidths of Dynamic Random Access Memory (DRAM) devices and bandwidths of Input/Output (I/O) devices. This analysis reveals that the bottleneck is not the memory bandwidth but the number of CPU cycles, and hence reducing the number of CPU cycles is a key to high speed NDN forwarding.

In order to know bottlenecks for CPU cycle consumption, the author carefully analyzes how NDN packet processing consumes CPU cycles at the level of CPU pipelines and instructions and identifies two bottlenecks. The first bottleneck is the pipeline stall. More precisely, the stall to wait data to be fetched from the DRAM device is the most significant bottleneck among all of pipeline stalls like the instruction fetch stall and the instruction execution stall. The above analysis also reveals that the stall due to data fetches is caused by accessing forwarding tables, which include tables for both name-based forwarding and caching, and they are placed on the slow DRAM device. This implies that eliminating the DRAM access latency due to forwarding table accesses is a key to high-speed NDN forwarding.

The second bottleneck is redundant computation of cache eviction. From the perspective of software constraints, the author analyzes how each function of NDN packet processing consumes CPU cycles and found the two facts related to wasteful cache eviction. The first fact is that the packet processing flow in the case of a cache miss consumes more CPU cycles than that in the case of a cache hit because a router consumes CPU cycles for forwarding an Interest packet[1], which would not be needed if the Interest packet hit the cache. The second fact is that the function of cache eviction and insertion for Data packets consumes many CPU cycles among functions constituting the packet processing flow in the case of a cache miss. The above two facts imply that cache algorithms need to increase cache hit rate without incurring computation overheads for high-speed NDN forwarding.

According to the above two implications, in this thesis, the author addresses to eliminate the DRAM access latency due to forwarding table accesses and to realize cache algorithm with fast computation and high cache hit rate.

1.2.1 Elimination of DRAM Access Latency

The first issue is how to eliminate the DRAM access latency due to forwarding table accesses. Significant efforts have been directed toward compact data structures and algorithms of name-based forwarding [9–12] to eliminate the DRAM access latency. A research issue of special concern is designing compact data structures and some studies show that trie-based data structures can be designed to be so compact that can be stored on SRAM devices by using a bit as a component of a trie [12]. Although the assumption that data structures are placed on fast memory devices is reasonable for NDN hardware routers, it is not reasonable for NDN software routers. Unlike on hardware routers, data cannot be pinned at CPU caches on computers since eviction and replacement in CPU caches are governed by their CPUs rather than user-space programs and operating systems.

The argument in the last paragraph implies that hiding DRAM access latency is a vital key to high speed NDN packet forwarding rather than compactness of the data structures. This is because the number of DRAM accesses never gets zero even though the sophisticated data structure and algorithm are adopted. Latest CPUs support *instruction and data prefetch*, which allows instructions and data to be fetched to CPU caches in advance of actual use. With the prefetch mechanism, NDN software routers can hide the large DRAM access latency. In order to exploit the prefetch mechanism, the address of data to be prefetched must be known in advance. In the case of a trie-based data structure, it is hard to hide the DRAM access latency with the prefetch mechanism because where to access cannot be estimated in advance of its access. More precisely, since a trie is searched by traversing vertices from its root, the address of the next vertex to be traversed cannot be estimated before its parent is fetched. In contrast, in the case of a hash table-based data structure, to hide

[1]Interest and Data packets are request and response packets in NDN, respectively. Interest packets transfer requests of consumer and Data packets do the corresponding data objects.

the DRAM access latency is easier than in the case of a trie-based one since access patterns of the hash table are predictable if hashes of keys are available in advance. Thus, a hash table-based data structure is more promising than a trie-based data structure.

In this thesis, the author addresses to hide the DRAM access latency as follows: First, the author carefully compares existing data structures and algorithms both for name-based forwarding and caching from the viewpoint of the ease of hiding DRAM access latency and chooses a hash table-based algorithm and a frequency-based cache admission algorithm for name-based forwarding and caching, respectively. Second, the author designs a prefetch algorithm for hiding most of the DRAM access latency for both name-based forwarding and caching. The prefetch algorithm reorders instruction executions for packets so that the DRAM access latency due to data fetches is successfully hidden by execution time of instructions independent of the data fetches. Finally, the author implements a proof-of-concept prototype and empirically proves that forwarding speed with the prefetch algorithm increases linearly as the number of CPU cores increases, whereas that without the prefetch algorithm does not increase so. A detailed analysis at the level of CPU pipelines and instructions reveal that most DRAM access latency are successfully hidden by the prefetch algorithm.

1.2.2 Fast Cache Algorithm with High Cache Hit Rate

The second issue is how to design fast cache algorithm with high cache hit rate. As the first step, the author carefully analyzes how two types of cache algorithms, i.e., cache eviction and admission algorithms, consumes CPU cycles in NDN packet processing and identifies the two obstacles of fast cache algorithm. First, cache insertion of a Data packet is relatively time-consuming among function blocks which are executed when a cache miss occurs. It means that insertion of unpopular Data packets which are not hit later wastes many CPU cycles. Second, sophisticated cache eviction algorithms like Adaptive Replacement Cache (ARC) [13] incur computational overheads for choosing a victim Data packet sophisticatedly, which will be evicted from the cache.

This implies that cache algorithms need to increase cache hit rate and reduce unpopular Data packet insertion rate simultaneously without incurring computation overheads for high-speed NDN software routers. The author proposes cache admission as a better candidate than cache eviction according to two reasons: First, cache admission obviously resolves the first obstacle because only the selected Data packets are inserted into the cache. Second, computation of cache admission is lighter than that of cache eviction to achieve high cache hit rate. The heavy computation of cache eviction comes from the fact that a cache eviction algorithm decides exactly one Data packet evicted from the cache by maintaining an eviction priority queue, where all the Data packets are sorted according to their eviction priority values in the cache according to recency or frequency. On the other hand, a cache admission algorithm performs lighter computation such that it only decides whether an incoming Data packet should be inserted into the cache or not and hence it does not need to maintain the eviction priority queue.

In this thesis, the author develops a light-weight cache admission algorithm, *Filter*, and shows an example case that cache admission is better than cache eviction for an NDN router. Filter exploits frequency, which is the number of Interest packets for a Data packet, as a history of requests to Data packets. The average computation time handling one Interest packet is about a few tens of CPU cycles. Nevertheless, Filter with FIFO eviction achieves high cache hit rate similar to that of ARC eviction with the above small computation. The author implements a proof-of-concept prototype of an NDN software router with Filter and empirically evaluates the overall packet forwarding speed of an NDN router with Filter based on its prototype implementation.

1.2.3 Power Efficiency Due to CPU Cycle Reduction

The above two approaches in Sections 1.2.1 and 1.2.2 address how to reduce the number of CPU cycles consumed by NDN packet processing in order to improve forwarding speed of NDN software routers. The author further addresses a positive effect of CPU cycle reduction from a different angle and to answer the question whether CPU cycle reduction contributes to improving power efficiency of NDN software routers. The author sheds light on power efficiency, which is also a crucial design issue of NDN routers as being addressed by many previous studies [14–17].

The author empirically models how NDN packet processing consumes power of hardware devices constituting an NDN software router on a computer. This model reveals that the CPU device consumes much power and that its consumed power is in proportion to its load, i.e., the number of consumed CPU cycles. This means that reducing the number of CPU cycles consumed by NDN packet processing contributes to power efficiency of an NDN software router. Based on the model, the author conducts a case study of analyzing power consumption of naive NDN software routers and those which use the proposed techniques. The case study reveals that reduction of the number of CPU cycles due to the proposed techniques contributes to power reduction of modern computers, which are energy proportional to their load.

1.3 Contributions

The main contributions of this thesis are summarized as follows: First, as far as the author knows, this is the first study that summarizes observations found in existing studies to build high-speed NDN routers on computers and compiles the observations into a design guideline. According to the guideline, the author reveals that a hash table-based algorithm and a frequency-based cache admission algorithm are suitable for name-based forwarding and caching, respectively. Second, the author designs a prefetch algorithm for hiding most of the DRAM access latency for both name-based forwarding and caching. The prefetch algorithm reorders instruction executions for a pair of packets so that the DRAM access latency due to data fetches is successfully hidden by execution time of instructions independent of the data fetches. Third, the author carefully designs a lightweight frequency-based cache admission algorithm, Filter, on the basis of the results of the empirical measurement of an NDN software platform. As the results of the careful design, Filter consumes a few tens of CPU cycles in average with providing as high cache hit rate as ARC. Fourth, the author implements a proof-of-concept prototype of an NDN software router with the proposed prefetch algorithm and the Filter admission algorithm and empirically proves that the router realizes a nearly optimal forwarding speed. Fourth, the author implements a proof-of-concept prototype of an NDN software router with the proposed two techniques and empirically proves that forwarding speed with the two techniques increases linearly as the number of CPU cores increases, whereas that without the two techniques does not increase linearly. A detailed analysis at the level of CPU pipelines and instructions reveal that most DRAM access latency are successfully hidden by the prefetch algorithm and that the number of CPU cycles of NDN packet processing is reduced by the Filter admission algorithm. Fifth, the author sheds light on power efficiency of NDN software routers as a positive effect of CPU cycle reduction at the different view point from forwarding speed improvement. By modeling how NDN packet processing consumes power of hardware devices on an NDN software router, the author reveals that reducing the number of CPU cycles for high speed NDN forwarding contributes to power consumption reduction of an NDN software router.

1.4 Outline

The rest of this thesis is organized as follows. Chapter 2 summarizes related work. Chapter 3 identifies true bottlenecks for high speed forwarding based on the instruction level analysis after explaining a state-of-the-art software NDN implementation used for the analysis. Chapter 4 designs a prefetch algorithm after choosing data structures and algorithms for name-based packet forwarding and caching among the existing algorithms from the view point of the ease of hiding DRAM access latency. Chapter 5 proposes an NDN packet forwarding scheme with the cache admission based on Filter after introducing a problem of wasteful caching computation. Chapter 6 models how hardware devices of NDN software routers consume power and then reveals that the CPU cycle reduction due to prefetch and Filter mechanisms contribute to reducing power consumed by NDN software routers. Finally, Chapter 7 concludes this thesis.

Chapter 2

Related Work

The thesis is compared with the four series of studies: name-based forwarding, caching, software packet processing, and power efficiency of routers.

2.1 Name-based Forwarding

Perino and Varvello investigate Content-Centric Networking (CCN) software in detail and claim that CCN deployment is feasible at ISP scale, whereas today's technology is not yet ready to support an Internet scale deployment of NDN/CCN [18]. To achieve high-speed NDN packet processing, many researches attack issues of heavy NDN functions, focusing on heavy name-based forwarding and caching.

To cope with the issue of heavy name-based forwarding, previous studies address efficient data structures of name-based forwarding and its efficient longest prefix match algorithms. Regarding efficient data structures, the author has enough design choices such as *Bloom filter-based*, *trie-based*, and *hash table-based* data structures. Dai et al. [9] propose a counting bloom filter-based data structure with the unified index of all of NDN forwarding tables, i.e., CS, PIT and FIB, so that the data structure is stored in the SRAM device. Song et al proposes a binary patricia trie using a bit as a component of a trie so that all of rules of name-based FIB is stored in the compact data structure [11]. So et al. propose a hash table with fast collision-resistant hash computation and compact arrays in order to reduce the number of DRAM accesses. Bloom filter-based and trie-based data structures address compactness of data structures of name-based forwarding, whereas the compactness is not suitable for an NDN software router because the number of DRAM accesses never gets zero on the NDN software router. From the ease of hiding DRAM access latency, the author chooses a hash table-based data structure [10], whereas its naive design incurs a few of unhidden data fetches, which are the root cause of forwarding speed degradation. In this thesis, the author designs prefetch techniques so that the DRAM access latency due to such unhidden data fetches is successfully hidden.

Regarding efficient longest prefix match algorithms, Yuan and Crowley propose a longest prefix matching algorithm based on binary search of hash tables, which reduces the worst case computation complexity [12]. So et al. design an efficient FIB lookup algorithm that provides good average and worst case FIB lookup time complexity. Fukushima et al. design a protocol that avoids redundant longest prefix matching by exploiting neighboring router's cooperation [19]. Although these studies successfully reduce the average time-complexity of longest prefix matching, these incur unhidden data fetches, of which latency is difficult to be hidden. From the ease of hiding DRAM access latency, the author proposes to use a simple longest-first search algorithm, as used in [20].

2.2 Caching

In contrast to name-based forwarding, only a few studies focus on fast computation of caching [21,22]. These studies address the large access latency to the secondary memory device like the Solid State Drive (SSD) storage, which is used for the packet cache of the NDN router. So et al. [21] propose to explicitly separate CPU cores for accessing the slow SSD storage and those for fast name-based forwarding to mitigate forwarding speed degradation of an entire NDN router. Mansilha et al. propose a caching algorithm which hides insufficient memory bandwidth between the main memory device and the secondary memory device used for a per-packet caching [22]. Both approaches use one or more CPU cores exclusively to compensate the large access latency to SSDs and thus it requires heavy computation with multiple CPU cores.

In this thesis, the author proposes to use cache admission according to the observation that handling Data packets of unpopular content is wasteful. At first, cache admission algorithms [23–25] are designed by leveraging cooperation of caches of routers in a network so that the same Data packet is not likely to be inserted into different caches. ProbCache [23] and Betweenness [25] insert Data packets equally into caches on a delivery path, whereas Leave Copy Down (LCD) [24] focuses on off-path caching. Nevertheless, the author uses cache admission algorithms without such cooperation, i.e., cache admission algorithms working independently on individual routers, being inspired by the observation found by Sun et al. [26] that the LFU-based cache algorithms without such cooperative cache admission achieves higher traffic reduction in a whole network as well as sufficiently similar cache hit rate compared with the cache admission algorithms with cooperation. A cache admission algorithm without such cooperation is simply referred to as a cache admission algorithm, hereafter.

Precisely, the author adopts a frequency-based cache admission algorithm which is executed independently from the other caches. TinyLFU [27] is a frequency-based cache admission algorithm and inserts Data packets which are received often recently into a cache. Although Filter of this thesis is also a frequency-based cache admission, Filter focuses on fast computation of cache admission itself compared to the TinyLFU, as the author will discuss later in Section 5.2.

2.3 Software Packet Processing

Prototypes of NDN software routers have been developed. Kirchner et al. [28] have implemented their NDN software router, named Augustus, in two different ways: a standalone monolithic forwarding engine based on the DPDK framework [29] and a modular one based on the Click framework. Though the forwarding speed of Augustus is very high, it does not approach the potential forwarding speed realized by computers. Hence, analyzing the bottlenecks of NDN software routers remains an issue.

How to exploit CPU caches is an important issue to realize the fast packet processing. Vector packet processing (VPP) [30] enhances packet processing speed by processing a vector of packets at once rather than processing each of them individually. The key idea behind VPP is to reduce the processing time by increasing the hit probability at the instruction caches. However, in the case of the NDN software router, the instruction cache misses, i.e., the instruction pipeline stalls at the frontend stage, do not occupy a significant proportion of the overall pipeline stalls. Furthermore, unlike VPP, approaches of this thesis focus on hiding the DRAM access latency, assuming that data cannot be pinned on the caches.

Fast packet processing is also a hot research topic in the area of networked applications. Li et al. [31] developed a multi-threaded in-memory key-value store that exploits DPDK and RSS. They have revealed that the number of cache misses at the L3 cache increases as the number of threads increases. However, NDN software implementation designed in this thesis scales up to 20 threads without increasing the number of L3 cache misses.

This thesis addresses the two research issues for high-speed forwarding of NDN software routers. First, high speed hash table handling is also an important issue in hash table-based systems such as key-value store systems [32] and OpenFlow switches [33] as well as high speed NDN software routers. These studies focus on fast mutual exclusion for read-intensive workloads because hash table lookups dominate most of operations, whereas this thesis avoids such mutual exclusion by having threads not access the same hash table at the same time. On the contrary, hiding long DRAM access latency in hash table handling is important for the both hash table-based systems. The author believes that the proposed prefetch technique for hiding DRAM access latency is applicable to the other hash table based systems. Second, a lightweight cache admission algorithm, filter, proposed in Section 5, is used together with cash algorithms of the above systems such as key-value store [32] and reverse proxy [34]. The studies focus on reducing meta data structures which are stored at caches because such data structures are larger than data pieces themselves, e.g., 64 bytes data pieces, whereas the filter focuses on reducing the number of data pieces stored at caches.

2.4 Hardware Packet Processing

Due to the trend that the integration degree of transistors has not increased on the CPU device recently [35], hardware packet processing with the ASIC-based chip becomes a hot research topic [36, 37]. A programmable switch hardware with the ASIC-based chip is being released by Barefoot Networks [38] and the switch hardware realizes both high processing speed by 6.4 Tbps and the programmability that various types of network applications like IP router [36] and network load balancer [37] can be implemented. The switch hardware typically implements packet processing as on-chip operations where all of data structures are pinned on SRAM or TCAM devices and packets are processed on ASICs using the data structures [36]. Most of network applications using the switch hardware sheds light on compact data structures to place them on on-chip SRAM or TCAM devices [37].

The programmable switch hardware is considered to be a promising platform for future NDN routers requiring about 1 Tbps forwarding speed [39]. However, many issues should be resolved so that NDN packet processing is implemented on the switch hardware. Among them, a large amount of NDN data structures raises an implementation issue since all of NDN data structures cannot be placed on on-chip TCAM and SRAM devices, as discussed in Section 1.1. This means that a part of NDN data structures must be placed on the off-chip DRAM device of the switch hardware like [36] or the remote DRAM device of the COTS computer connected to the switch hardware like [40]. In such the cases, the ASICs need to fetch data pieces of the data structures from the DRAM device to on-chip SRAM or TCAM devices before using them. Thus, the latency to fetch data from the DRAM device becomes a issue of high-speed NDN packet processing on the programmable switch hardware in similar to the COTS computer hardware. The author believes that the data prefetch techniques proposed in Section 4.3 contribute to hiding such the latency on the programmable switch hardware.

2.5 Power Efficiency of Routers

Many studies which address power efficiency of NDN routers and networks focus on the caching function because its traffic reduction would contribute to energy savings. Lee et al. [14,15] investigated how much power is reduced by reducing hop counts to obtain contents. In their simulations, they used a power consumption model which only considers the power consumed by lower layer packet forwarding devices. Choi et al. [16] show that the power consumed by memory devices used for caching and for the forwarding processes is not insignificant. Imai et al. [17] proposed a method of determining capacities of NDN routers' memory devices so that the power they consume is minimized.

Many power consumption models focus on the power consumed by memory devices because they are power-hungry even while in idle state. In contrast, the author focuses on the power consumed by the CPU device for packet forwarding, taking into account the trend towards low power consumption by memory devices [41,42].

Bolla et al. [43, 44] model the power consumed by a CPU device with power optimization techniques by considering two power management states. Although the study of this thesis is partially motivated by [43,44], in this thesis, the author models multiple CPU devices (compared to only one in [43,44]) and empirically validates the developed model by applying it to two different commercial CPU devices. Finally, the author takes into account all elements of the router (*e.g.*, chassis, memory devices and CPU), in contrast to [43,44], where only the CPU is taken into account.

Chapter 3

Microarchitectural Bottleneck Analysis of an NDN Software Router

As the first step to achieve a high-speed NDN software router, in this chapter, the author identifies true bottlenecks of NDN packet forwarding on a computer from the perspectives of hardware and software constraints as described below: First, the author considers what is the most significant constraint among the constraints on the computation capability of a CPU device, bandwidths of DRAM devices and bandwidths of Input/Output (I/O) devices by analyzing how a current state-of-the art NDN software router implementation behaves on a computer. From the analysis, the author reveals that computation on a CPU is the most significant constraint. Second, the author analyzes how individual hardware components of the CPU spend time for processing NDN packets at the level of instructions and instruction pipelines. By carefully analyzing the behavior at the level of CPU pipelines and instructions, the author identifies two types of bottlenecks of NDN packet forwarding. The first bottleneck is the unhidden DRAM access latency due to forwarding table accesses and the other one is the number of instructions executed for NDN packet processing. Third, the author analyzes how each of NDN functions consumes CPU cycles for processing a packet in order to identify the cause of the large number of executed instructions. The analysis reveals that packet processing flow in the case of a cache miss consumes many CPU cycles and that among functions constituting the packet processing flow in the case of a cache miss, a function of cache eviction and insertion consumes many CPU cycles. Finally, the author compiles the above analysis results into two design rationales for high-speed NDN packet forwarding. The first rationale is that data structures and algorithms for NDN packet processing should be designed so that the DRAM access latency is hidden. The second rationale is that cache algorithm should be designed so that high cache hit rate is achieved with fast computation.

The rest of this Chapter is organized as follows. The author explains NDN packet forwarding behavior and a state-of-the-art software NDN implementation used for the bottleneck analysis in Section 3.1. In Section 3.2, the author identifies bottlenecks for high speed forwarding based on the instruction level analysis from the perspective of hardware constraints. In Section 3.3, the author identifies bottlenecks of computation time of NDN packet processing from the perspective of software constraints. Based on analysis results, the author finally provides design rationales for a high speed NDN router in Section 3.4.

3.1 Hardware and Software Platforms

After describing a hardware platform, this section summarizes software design practices to exploit the parallel computation capabilities of CPU cores, which are essential to high-speed NDN packet forwarding.

3.1.1 Hardware Platform

A reference hardware platform is illustrated in Fig. 3.1. It is based on Intel Xeon family CPUs [45], double data rate fourth-generation synchronous (DDR4) DRAM devices, and latest network interface cards (NICs).

CPU Cores and Caches Each CPU has several CPU cores, and they are interconnected with one or two full-duplex on-die ring interconnects. The CPU has L1 instruction, L1 data, L2, and L3 caches. For the sake of notation simplicity, L1 instruction and L1 data caches are referred to as L1I and L1D caches, respectively. In contrast to the L3 cache, which is shared with all CPU cores, each of the CPU cores has exclusive L2, L1I, and L1D caches.

Data Transfer between CPU and NIC/DRAM The DRAM and NICs are connected with the CPU via a memory controller and a PCI express (PCIe) 3.0 controller on its die. PCIe devices, such as network interface cards (NICs), are connected via the PCIe controller. The CPUs and NICs support the Intel Data Direct I/O (DDIO) technology [46], which allows data to be transferred directly between NICs and the L3 cache without passing through the DRAM device. The data transfer rate via the PCIe controller is 96 Gbytes/s and data is prefetched at any time after all packets in the NIC

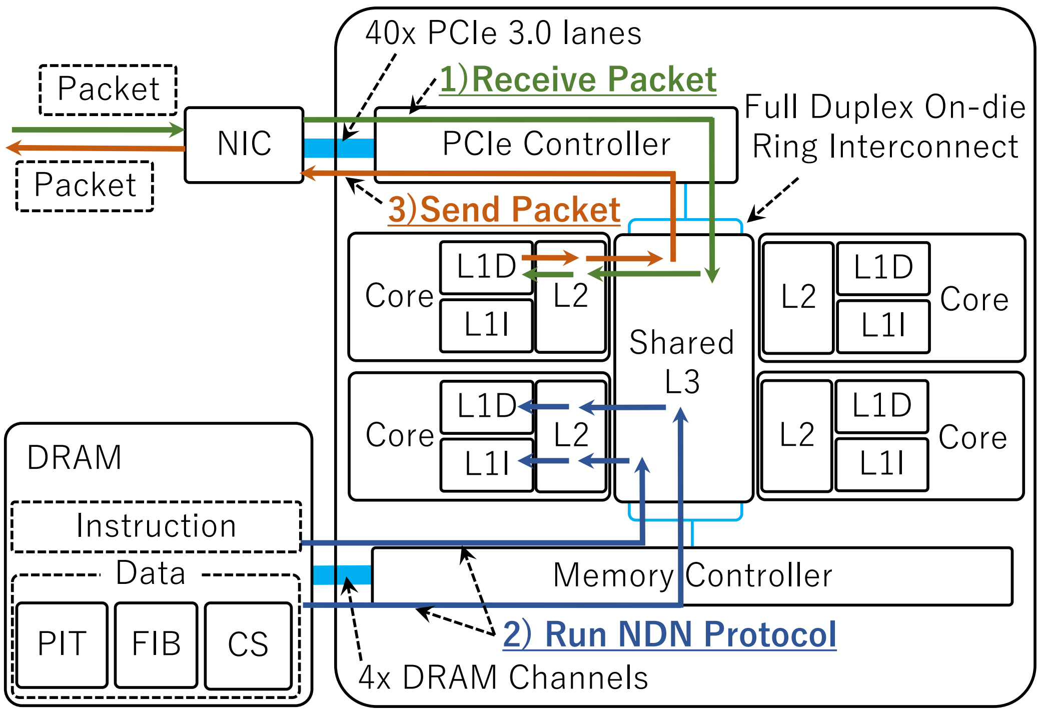


Figure 3.1: Router hardware architecture

are transferred to the L3 cache in a batch. The memory controller supports DDR4 DRAM devices and has four DDR4 DRAM channels. The allocation rule of memory requests from the memory controller to the DRAM channels is determined by the memory management library provided by DPDK [29]. Data, including NDN tables and instructions, is placed on the DRAM device and it is fetched via the memory controller. The access latency to the DRAM device is one or more orders of magnitude greater than that to CPU caches. To hide the DRAM access latency, the CPU has two types of prefetch mechanisms: *hardware* and *software prefetch mechanisms*. The CPU automatically fetches instructions and data with the hardware prefetch mechanism if their accesses can be predicted in advance. In addition, data can be speculatively fetched to the caches using the software prefetch mechanism, which is controlled by the software with the PREFETCH instruction [47]. How to issue the PREFETCH instruction is vital to hiding the large DRAM access latency.

3.1.2 NDN Packet Forwarding

The packet forwarding flow of NDN on the above router hardware is illustrated in Fig. 3.1. When a packet arrives at a NIC, it is directly transferred from the NIC to the L3 cache. Then, the packet is fetched from the L3 cache to a CPU core and NDN protocol processing is performed for the packet. Finally, the packet is transmitted from the L3 cache to a NIC. To fully exploit the computation capabilities made possible by many CPU cores, received packets are processed by several threads in parallel. the author briefly describes NDN in the following paragraphs.

NDN realizes request/response communication of a named object using two types of packets: an Interest packet and a Data packet, which correspond to a request packet and a response packet, respectively. Both Interest and Data packets have a name which uniquely identifies a piece of an object and only the Data packet has the piece of an object.

NDN protocol processing mainly consists of Content Store (CS), Pending Interest Table (PIT) and Forwarding Information Base (FIB). CS is a cache memory[1] which stores previously processed Data packets. FIB is a forwarding table which stores outgoing interfaces for Interest packets. PIT is a table which stores arrival interfaces of pending Interest packets and it is used as a forwarding table for Data packets. An NDN router realizes NDN protocol processing by using the above three components as follows: When an Interest packet arrives, the router looks up a piece of an object from its exact name in the CS. If the router finds the piece, it sends back a Data packet with the piece to the arrival interface of the Interest packet. Otherwise, the router inserts the arrival interface with the name of the Interest packet into the PIT. Finally, the router looks up the outgoing interface having the longest name prefix of the name of the Interest packet and sends the Interest packet to the interface. When a Data packet arrives, the router looks up its outgoing interface from its exact name in the PIT and deletes it from the PIT. Then, the router inserts the Data packet into the CS and evict the most unpopular Data packet from the CS according to cache algorithms, i.e., cache eviction and admission. Finally, the router sends the Data packet to the outgoing interface.

3.1.3 Practices to Exploit Parallel Computation

Exploiting the parallel computation capabilities of CPU cores is the key to high-speed NDN packet forwarding, which is more complex than IP packet forwarding. To exploit capabilities of computers, the software must eliminate obstacles as follows: 1) *mutual exclusion*, 2) *hardware interrupts*, 3) *instruction pipeline stalls*, and 4) *time complexity*. Obstacles due to mutual exclusion 1) and hardware interrupts 2) have already been resolved by existing studies. The author summarizes the design practices for the first two obstacles in the rest of the section, and then addresses the last two obstacles in the rest of this thesis. These are the main research issues of this thesis.

[1]To avoid confusion, hereinafter, the author refers to caches on a CPU and an NDN router as *cache*, which is the only SRAM device in a computer, and *content store (CS)*, respectively.

Eliminating Mutual Exclusion

When an area on memory devices is locked with mutual exclusion, threads waiting for the area to become free is being idle. In a similar way, hardware interrupts and instruction pipeline stalls also make threads being idle. One way to compensate for such idle time is to increase the concurrency, i.e., the number of threads, so that other threads can use idle computing units. For instance, a prototype NDN software router, named Augustus [28], uses Hyper-Threading Technology. However, this approach may introduce many context switches, and the context switches also results in significant overheads. In this sense, the author suggests that the number of threads should be kept equal to or less than the number of CPU cores.

To utilize computing capabilities derived from multi-core CPUs, multi-threading strategies must be designed carefully. The author chooses the strategy of assigning one thread to one NDN packet rather than assigning a sequence of threads to one packet as a pipeline to prevent the pipeline from being disturbed by I/O waits. The author focuses on two important issues for the multi-threading strategies: elimination of mutual exclusion and balancing the workloads for processing packets equally across all threads. To resolve two issues, the author adapts exclusive NDN tables and packet-level sharding. Mutual exclusion for any shared data would result in serious performance degradation.

To eliminate mutual exclusion, each thread exclusively has its own FIB, PIT, and CS, thereby preventing it from accessing the tables of the other threads, as the existing prototypes of NDN software routers adopt [10, 28].

Arriving packets are forwarded to threads according to their names by using receiver side scaling (RSS), which allows NICs to send different packets to different queues to distribute the workloads among CPUs. As Saino et al. [48] revealed in their investigation, packet-level sharding, which distributes Data packets to CSs based on the result of a hash function computed on the names of the packets, distributes them equally to each CS. Based on their results, the author assumes in this thesis that the software distributes packets equally to each thread according to the hashes of their names at the NICs by using RSS. Please note that partial name matching of Interest and Data names is not supported in this thesis because the sharding mechanism using RSS requires that a Data packet is explicitly requested by its exact name.

Eliminating Hardware Interrupts

To eliminate hardware interrupts, the author adopts the DPDK user-space driver [29], which allows user-space software to bypass the protocol stacks of operating systems and provides a way to transfer data directly from NICs to the software.

Remaining Issues

The remaining issues are twofold: one is to eliminate instruction pipeline stalls caused by NDN packet processing and the other is to design NDN packet processing itself to optimize its time complexity. To identify the main cause of instruction pipeline stalls, the author conduct a bottleneck analysis at the level of the instruction pipeline in Sec. 3.2. To address time complexity, the author identifies the root cause of such high time complexity in Sec. 3.3.

3.2 Microarchitectural Hardware Bottleneck Analysis

To identify a true bottleneck of NDN packet forwarding on a computer, the author conducts a *microarchitectural hardware analysis*, which analyzes how individual hardware components of the CPU spend time for processing NDN packets at the level of instructions and instruction pipelines, with the method described in the previous section.

3.2.1 Bottleneck Analysis Method

This subsection summarizes the microarchitectural analysis method.

Analysis of Packet Forwarding Flow

First, the author identifies which of the factors, computation time or I/O bandwidth, is the bottleneck limiting the maximum packet forwarding rate by defining how these two factors limit the maximum rate.

The packet forwarding rate, λ packet/s, is defined as the number of packets, including Interest and Data packets, forwarded per second. When the packet forwarding rate is λ, the consumed CPU cycles are derived as λC, where C is the average number of CPU cycles consumed to process a packet. Assuming the load for processing packets is equally assigned to N CPU cores operating at a frequency of F Hz, the maximum packet forwarding rate, λ_{max}, must satisfy $\lambda_{\text{max}} \leq NF/C$. In the same way, the average number of bytes read from and written to DRAM devices for processing a

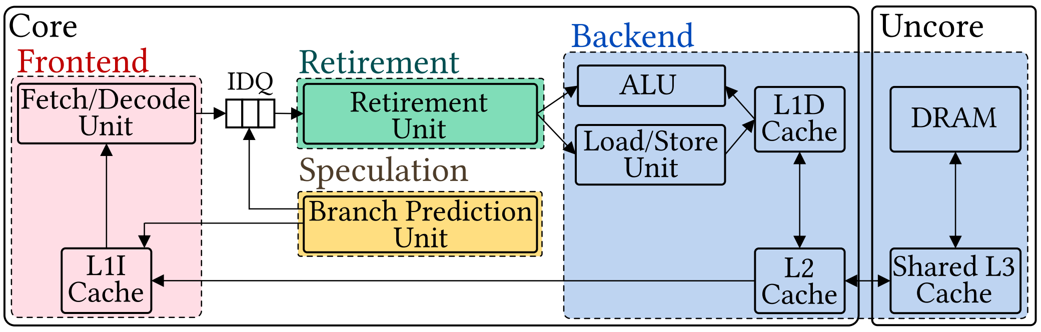


Figure 3.2: Hardware units on a CPU and four main stages of instruction pipeline

packet is expressed as M and it can be modeled similarly to the CPU cycles. The maximum packet forwarding rate $\lambda_{\max}$ satisfy $\lambda_{\max} \leq B/M$, where B is the bandwidth of DRAM channels, thereby $\lambda_{\max}$ satisfying

$$\lambda_{\max} \leq \min\left(NF/C, B/M\right). \tag{3.1}$$

If the maximum packet forwarding rate is NF/C, computation time is a bottleneck. Otherwise I/O bandwidth is a bottleneck.

Analysis of Pipeline Stalls

Second, the author analyzes the bottleneck for computation since computation time is the bottleneck, as shown later in Section 3.2.3. Specifically, the author analyzes where hazards occur in the pipeline and how long the pipeline is stalled due to the hazards. The author uses the top-down micro-architecture analysis method proposed by Yasin [49]. Before analyzing the pipeline hazards, the author explains the pipeline and flows of instructions and data on a CPU.

The pipeline has four main stages: *frontend*, *backend*, *speculation*, and *retirement*, as shown in Fig. 3.2. The frontend stage is responsible for fetching instructions from the DRAM device or the CPU caches and feeding them to the backend stage, and the speculation stage supports the frontend stage in its instruction fetches, using the branch prediction unit. In contrast, the backend stage is responsible for executing the instructions, fetching necessary data from the DRAM device or the caches and storing data on the caches. Finally, instructions executed successfully are retired at the retirement stage.

The CPU cycles spent for processing the pipeline are also categorized into four major parts: CPU cycles spent during the pipeline stalls at the *frontend* stage due to *instruction cache misses*, that spent during pipeline stalls at the *backend* stage due to *data cache misses*, that wasted at the *speculation* stage due to *branch prediction misses*, and that spent when *instructions are successfully executed and retired* at the *retirement* stage. In the ideal case, where instructions are executed without pipeline stalls, the CPU cycles spent at all stages except for the retirement stage become zero. That is, the CPU cycles spent at the retirement stage are equivalent to the minimum computation time.

The pipeline stalls at each of the stages can be further broken down into more specific categories based on the reasons for the stalls. The pipeline stalls at the backend stage are divided into the following three categories: pipeline stalls due to waits at *load* units, at *store* units, and at *core* units, such as instruction execution units. Since the pipeline stalls at the backend stage dominate the overall pipeline stalls, the author omits any detailed investigation of the pipeline stalls at the frontend and speculation stages. Among them, the pipeline stalls due to waits at load units dominate the overall pipeline stalls at the backend stage, and hence the author further categorizes the pipeline stalls at the load units into the following four categories: pipeline stalls to wait for data to be fetched from the *L1D*, *L2*, *L3* caches, and the *DRAM* device.

CPU Cycle Measurement Method

The number of CPU cycles spent at each of the pipeline stages are measured with a performance monitoring unit (PMU) on CPUs. A PMU has a set of counters, which count the number of occurrences of hardware events. Such events include pipeline stalls at the frontend, backend, and speculation stages and the instruction retirement at the retirement stage.

3.2.2 Experiment Settings

Prototype Implementation

The author implements a prototype of basic NDN functions which include longest prefix match on FIB lookup, exact match on CS lookup and PIT lookup, and FIFO-based CS eviction according to a state-of-the-art software implementation [10]. The processing flow of the prototype is shown in Fig. 3.3. When an Interest packet is received, the prototype performs NDN functions in the same order as the block diagram of [7] except for the functions below. A nonce is not calculated because the number of CPU cycles needed for the calculation is much smaller than that of the whole Interest packet forwarding [50]. Only the Best Route Strategy is supported. Partial name matching of Interest and Data names and selectors are not supported because the goal of this thesis is fast name-based longest prefix match. The FIB does not have entries other than those for Interest forwarding such as those for RTT estimation because fast stateful forwarding is not a goal of this thesis. When a Data

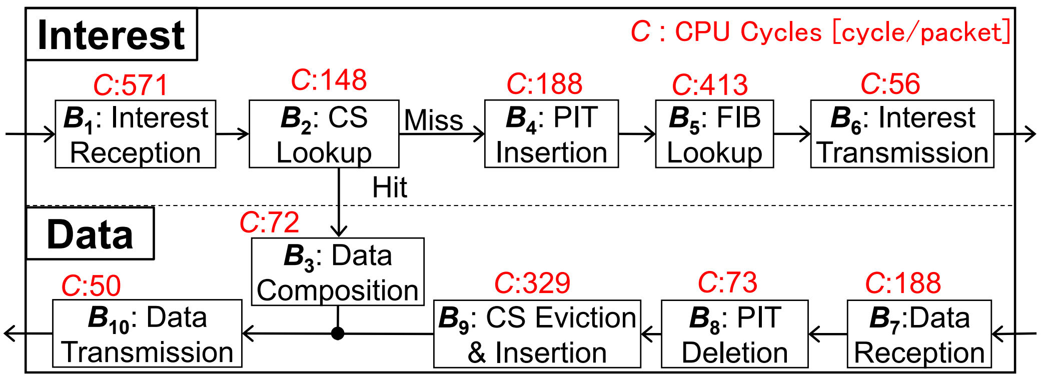


Figure 3.3: NDN software block diagram and analysis of computation time

packet is received, the prototype performs NDN functions in the same order. The CS is stored on the DRAM device to avoid long read latency to the hard disk drive. The prototype employs Interest and Data packets that conform to the NDN-TLV packet specification [51], and Interest and Data packets are encapsulated by IP packets, of which the headers hold the hashes of their name and are used for RSS at the NICs.

Experiment Conditions

For the experiments, the author uses two computers with a Xeon E5-2699 v4 CPU (2.20 GHz $\times$ 22 CPU cores), eight DDR4 16 Gbytes DRAM devices, and two Intel Ethernet Converged Network Adapter XL710 (dual 40 Gbps Ethernet ports) NIC. The operating system on the computers is an Ubuntu 16.04 Server.

One computer is used as a router and the other is used as a consumer and a producer. The two computers are connected using four 40 Gbps direct-attached QSFP+ copper cables. The two links are used for connecting the consumer and the router and the other two are used for the router and the producer. That is, the total bandwidth between the consumer and the router and the router and the producer is 80 Gbps. The consumer sends Interest packets to the producer via the router. The producer returns Data packets in response to the Interest packets via the router. The author sets the payload size of Data packets to 128 bytes. For the experiments, the consumer generates 80 million different Interest packets with different 80 million names so that all Interest packets miss at the CS and the PIT.

To generate workloads and FIB entries for experiments, the author uses the results of the analysis on HTTP requests and responses of the IRCache traces [52] conducted in [10]. 13,548,815 FIB

Table 3.1: The consumption of CPU cycles and I/O bandwidth for processing one Interest or Data packet

	I-miss	I-hit	Data
CPU Cycles	1470	860	565
Memory I/O Bandwidth	909	511	816

entries are stored in the FIB. The average number of components in Interest packets is set to 7 and the average length of each component is set to 9 characters. The average number of components in prefixes of FIB entries is set to 4 and consequently, the average number of FIB lookup operations per one Interest packet is slightly larger than that in [10].

3.2.3 Analysis Results

This subsection conducts the microarchitectural bottleneck analysis in accordance with the steps described in Section 3.2.1.

Analysis Results of Packet Forwarding Flow

The evaluation is accomplished by using the prototype in Fig. 3.3 with two steps: I) Measuring the CPU cycles and I/O bandwidth consumed to process a packet and II) estimating the maximum packet forwarding rate limited by CPU computation time and I/O bandwidth.

I) CPU Cycles and I/O Bandwidth Consumption: The author measures CPU cycles and I/O bandwidth consumed for processing one Interest packet or one Data packet in the following three cases: The Interest packet processing flow in the case of CS misses, the Interest packet processing flow in the case of CS hits, and the Data packet processing flow. CPU cycles and I/O bandwidth are measured using the time-stamp counter and the performance monitoring unit on the CPU, respectively. The CPU cycles and the DRAM channel bandwidth consumed for processing one Interest or Data packet are shown in Tab. 3.1.

II) Maximum Throughput: Based on the measured results, the author estimates the maximum packet forwarding rate limited by either computation time or DRAM channel bandwidth. the author assumes two computers: Computer 1 has one CPU (2.20 GHz $\times$ 22 CPU cores), which has the highest number of CPU cores among Intel Xeon E5 CPUs, and computer 2 has one CPU (2.60 GHz $\times$ 4 CPU cores) [45]. The maximum bandwidth of DRAM channels on both computers scales up to 76.8 Gbytes/s [45]. For this analysis, the author assumes a 30% CS hit rate. The average number of CPU cycles and DRAM bandwidth consumed for processing a packet are $C = 925.4$

Table 3.2: The number of CPU cycles spent in the instruction pipeline (Entire pipeline)

	I-miss	I-hit	Data
Total	1470	860	565
Frontend	161	40	73
Backend	662	359	220
Speculation	161	18	42
Retirement	486	443	230

Table 3.3: The number of CPU cycles spent in the instruction pipeline (Backend)

	I-miss	I-hit	Data
Backend	662	359	220
Load	512	256	146
Store	10	11	14
Core	140	92	60

cycles and $M = 800.7$ bytes, respectively. The maximum packet forwarding rate limited by the DRAM bandwidth of both computers is $B/M = 95.9 \times 10^6$ packets/s. In contrast, the maximum packet forwarding rates limited by CPU cycles, i.e., NF/C, for computer 1 and 2 are calculated to be $NF/C = 52.3 \times 10^6$ and 11.2×10^6 packet/s, respectively.

In summary, the computation time is the more significant constraint limiting the maximum packet forwarding rate, i.e., $NF/C \ll B/M$, in both cases where the number of CPU cores is high and where it is small. The author obtains an observation that computation time is the most significant constraint for high-speed forwarding on modern computer platforms. The observation that, computation time is a bottleneck, is considered to hold true on future computer platforms. This is because the memory bandwidth B continues to increase year by year [53], whereas either the CPU clock speed F or the CPU cores number N has not been increasing recently [35].

Analysis Results of Pipeline Stalls

The CPU cycles spent during the pipeline stalls at the frontend, backend, and speculation stages and the instruction executions at the retirement stage are summarized in Table 3.2. The author measures the CPU cycles spent for processing one Interest packet or one Data packet in the following three cases: The Interest packet processing flow in the case of CS misses, the Interest packet processing flow in the case of CS hits, and the Data packet processing flow. To simplify the notation, these three cases are denoted by *I-miss*, *I-hit*, and *Data* in the tables. Among all the three cases, I-miss consumes the largest number of CPU cycles. Only the flow includes the FIB lookup procedure, and

Table 3.4: The number of CPU cycles spent in the instruction pipeline (Load Unit)

	I-miss	I-hit	Data
Load	512	256	146
L1D	33	66	20
L2	0	0	0
L3	19	28	32
DRAM	460	162	94

Table 3.5: The number of retired load instructions

	I-miss	I-hit	Data
Load	328	289	127
L1D	322	286	122
L2	1	1	2
L3	1	1	2
DRAM	4	1	1

hence the design of FIB data structures and FIB lookup algorithms is a critical issue, as discussed in the previous subsection. For all the three cases, the pipeline stalls at the backend stage consume the nearly one third or more of all the CPU cycles. In the rest of this chapter, the author focuses on eliminating stalls at the backend stage because they are more significant than those at the other stages and the root cause of such stalls is identified, as described below. Please note that eliminating stalls at the other stages is a topic left for future study.

Backend Stalls

To elucidate the causes of the long stalls at the backend stage, the author measures the stalls due to the load, store, and core units. The results are summarized in Table 3.3. Most of the pipeline stalls at the backend stage are caused by the load units waiting data to be fetched from either the caches or the DRAM device.

The author further drills down into the stalls at the load unit level: stalls to wait for data to be fetched from the L1D, L2, L3 caches, and the DRAM device. The results are summarized in Table 3.4. Most of the CPU cycles due to the stalls at the load units are spent to wait for data to be fetched from the DRAM device. It is a straightforward result since the DRAM access latency is much longer than those of the L1D, L2, and L3 caches. However, the number of accesses to the DRAM device, which is measured by the number of retired load instructions, is surprisingly small, i.e., 4 or less, as shown in Table 3.5. This is because most of the data fetches from the DRAM device are

Table 3.6: The number of CPU cycles against the number of threads

The number of threads	1	10	20
Entire Pipeline	1470	1567	2033
DRAM cycles	460	640	878
The number of DRAM accesses	4	4	4

hidden by the hardware prefetching mechanisms of the CPU and such data would be provided to core units without stalling the pipeline. However, a few DRAM accesses that cannot be prefetched using the hardware prefetching mechanism cause significant pipeline stalls and these lead to significant degradation of the forwarding speed.

Multi-Threaded Case

The author next evaluates the CPU cycles and the number of DRAM accesses in the case of multi-threaded software. The results are summarized in Table 3.6. The author allocates 2 CPU cores to the operating system and the remaining 20 CPU cores to the NDN software router. The number of CPU cycles due to the stalls to wait for data to be fetched from the DRAM device increase as the number of threads increases while the number of DRAM accesses does not increase.

Observations

From the above analysis results, the author obtains three observations related to bottlenecks of an NDN software router. The first observation is that the pipeline stalls to wait for data to be fetched from the DRAM device spent the majority of CPU cycles. For the Interest packet processing flow, 31.3% of the CPU cycles are spent during the pipeline stalls to wait for data to be fetched from the DRAM device. Hence, to hide the latency due to DRAM accesses is an important design issue for high-speed NDN software routers.

The second observation is that the CPU cycles due to the stalls to wait for data to be fetched from the DRAM device increases as the number of threads increases. This is because there is contention at the memory controller on the CPU and the wait time due to the contention increases as the number of threads increases. In contrast, the average number of DRAM accesses does not change even if the number of threads increases. This is counterintuitive because the L3 cache is shared by all CPU cores. The reason for this effect is that the most part of the three tables, i.e., the PIT, the CS, and the FIB, is stored on the DRAM device regardless of the number of threads, and therefore the data of the tables will be fetched every time they are used.

The third observation is that the number of CPU cycles spent at the retirement stage is not negligible from the result shown in Table 3.2. The CPU cycles spent at the retirement stage are equivalent to the minimum computation time without any pipeline stalls. This implies that there is still room to reduce computation time of NDN packet processing.

In the next section, the author identifies the cause of long computation time of NDN packet processing by NDN software analysis.

3.3 Microarchitectural Software Bottleneck Analysis

This section analyzes computation time of NDN packet processing from the perspective of software constraints. The author identifies software constraints by analyzing how each of NDN functions consume CPU cycles based on the state-of-the-art NDN software implementation after describing its packet processing flow.

3.3.1 NDN Packet Processing of Reference NDN Software

This section describes the NDN packet processing flows within the cases of both cache hits and misses according to the diagram illustrated in Figure 3.3. Block B_1 receives and decodes an incoming Interest packet and then computes hash values of prefixes of its name. B_2 checks whether a data piece of a Data packet corresponding to the Interest packet is stored in the CS. If the data piece is in the CS, B_3 fetches it and composes the Data packet with it, and then B_{10} sends back the Data packet to the incoming interface.

Otherwise, i.e., if the Interest packet does not hit the CS, B_4 inserts the incoming interface into the PIT. After B_5 gets an outgoing interface by longest name prefix lookup of FIB, B_6 sends the Interest packet to the outgoing interface. When a returned Data packet corresponding to the Interest packet arrives at the router, B_7 receives and decodes the Data packet and then computes a hash value of its name. After B_8 gets and deletes an outgoing interface from the PIT, B_9 evicts a victim from the cache according to the cache eviction algorithm and then inserts the incoming Data packet into the CS. Finally, B_{10} sends the Data packet to the outgoing interface.

3.3.2 Analysis Results

The author analyzes how each function of NDN packet processing described in Fig. 3.3 consumes CPU cycles for processing a packet. For the analysis, the author employs the same experiment settings that are used in the bottleneck analysis described in Section 3.2.2. The measured CPU cycles spent at each block for processing one packet are summarized in Fig. 3.3. The value after the character "C" represents the CPU cycles spent at each block. From this figure, the author founds the following two facts related to software constraints of NDN packet processing.

First, among two packet processing flows, i.e., the packet processing flow in the case of a cache miss and that in the case of a cache hit, that in the case of a cache miss consumes many CPU cycles, where packet processing flows in cases of a cache miss and a cache hit consume 2016 and 791 cycles, respectively. This is because a router consumes CPU cycles for forwarding an Interest packet at B_4, B_5 and B_6, which are not needed if the Interest packets hits the CS, in addition to those for processing the corresponding Data packet at B_7, B_8, B_9 and B_{10}.

Second, among function blocks which are executed in the case of a cache miss, the block B_5 spends the most CPU cycles and B_9 does the second most CPU cycles. The number of CPU cycles spent at B_9 is caused by computations of the cache eviction algorithm, whereas the number of CPU cycles spent at B_5 is mostly caused by the stall to data pieces of the FIB to be fetched from the DRAM.

From the above two facts, the author obtains the following observations. First, the packet processing flow in the case of a cache miss should be prevented from being executed because it consumes many CPU cycles. Second, computation of cache algorithms should be fast in order to reduce computation time of the packet processing flow in the case of a cache miss. These two observations imply that the cache algorithm needs to improve cache hit rate with fast computation. Finally, the stall to wait data pieces of the FIB to be fetched from the DRAM device should be successfully hidden.

3.4 Design Rationale

Finally, this subsection summarizes design rationales for high-speed NDN forwarding on the basis of bottleneck analysis results in Sections 3.2 and 3.3. The first principle is to hide instruction pipeline stalls to wait for data to be fetched from the DRAM device by leveraging the software prefetch mechanism. In the ideal case, all DRAM access latency is hidden behind instructions at the retirement stage. The second principle is to reduce computation time of NDN packet processing. This is because if the computation time is longer than the DRAM access latency, computation becomes a bottleneck for fast name-based forwarding and caching. To reduce the computation time, cache algorithm for NDN software routers needs to improve cache hit rate with fast computation time, as identified in

Section 3.3.

According to the first principle, the author chooses data structures and algorithms and then designs prefetch algorithms to hide all of the DRAM access latency in Chapter 4. According to the second principle, in Chapter 5, the author designs an NDN packet forwarding scheme with a simple cache admission algorithm so that high cache hit rate is achieved with fast computation.

Chapter 4

Data Prefetches to Eliminate Pipeline Stalls on Hash Table-based Forwarding Tables

In Chapter 3, the author identified two bottlenecks. The first bottleneck is unhidden DRAM access latency due to forwarding table accesses and the other one is wasteful computation of cache algorithm. Among them, in this chapter, the author addresses how to hide DRAM access latency.

The rest of this chapter is organized as follows. First, in Sections 4.1 and 4.2, the author carefully compares existing data structures and algorithms both for name-based forwarding and caching from the viewpoint of the ease of hiding DRAM access latency and chooses a hash table-based algorithm and a frequency-based cache admission algorithm for name-based forwarding and caching, respectively. Then, in Section 4.3, the author designs a prefetch algorithm for hiding most of the DRAM access latency for both name-based forwarding and caching by reordering instruction executions for a pair of packets so that the DRAM access latency due to data fetches is successfully hidden by execution time of instructions independent of the data fetches. In Section 4.4, the author implements a proof-of-concept prototype and empirically prove that it realizes a nearly optimal forwarding speed. A detailed analysis at the level of CPU pipelines and instructions reveal that most DRAM access latency are successfully hidden by the prefetch algorithm. Finally, the author concludes this chapter in Section 4.5.

4.1 Data Structure and Algorithm for FIB

4.1.1 Overview

In this section, the author chooses a FIB data structure appropriate for hiding DRAM access latency between two representative FIB data structures, i.e., a hash table-based FIB [11] and a trie-based one [12] by comparing the average numbers of dependent DRAM accesses for longest name prefix matching for a queried name. The word “dependent” means that the address of a “dependent” data piece is not determined until a data piece which contains the pointer to the dependent data piece is fetched from a DRAM device. Since such data pieces in the DRAM device should be fetched in sequence, fetching dependent data pieces creates a bottleneck even if multiple data pieces in DRAM devices can be fetched in parallel. It is clear that time of longest name prefix matching is at least the dependent data piece number multiplied by time of one data fetch from a DRAM device, i.e., about 115 CPU cycles.

The author does not consider a bloom filter-based FIB [9, 54] because it accesses a bloom filter before accessing a hash table. This obviously increases the number of dependent DRAM accesses compared to a hash table-based FIB.

4.1.2 Hash Table-based FIB

Data Structure and Exact Matching Algorithm

A hash table-based FIB [10] uses a *hash table*, which is a data structure to realize an associative array, to store mappings of name prefixes and interfaces. In the case where a hash table is used as an underlying data structure of a FIB, its keys are name prefixes and its values are interfaces corresponding to the keys. Hereinafter, the author refers to name prefixes simply as prefixes. Although a naive hash table is a chained hash table with linked lists where several hash entries are connected via pointers, it needs to serially traverse several hash entries through pointers and the traverse causes a large number of dependent DRAM accesses. To resolve this problem, the author employs a sophisticated chained hash table proposed by So et al. [10]. It packs pointers to several hash entries on a hash bucket, which is a single cache line-sized array, so that this eliminates the above traverse of hash entries. The author refers to buckets and entries of a hash table as *hash buckets* and *hash entries*.

The author first discusses exact matching with a hash table from the perspective of dependent DRAM accesses since a longest prefix matching operation on a name is realized by iteratively executing exact matching operations on its name prefixes. Figure 4.1 shows the exact matching

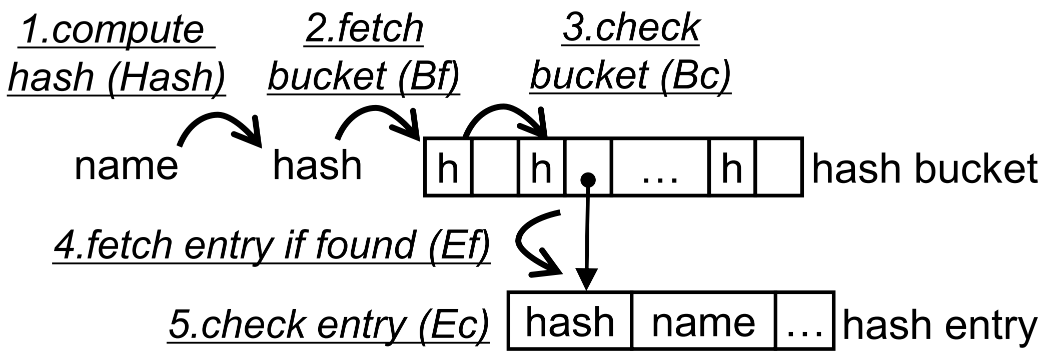


Figure 4.1: Flow of Exact Matching Algorithm

algorithm in the case where the sophisticated chained hash table is used. It looks up a prefix in a hash table which is matching to a queried name of an Interest packet, as described below: The algorithm, first, computes a hash value of the name (1 of Fig. 4.1) and fetches a hash bucket which corresponds to the hash value (2 of Fig. 4.1). This is the first fetch from the DRAM device. Then it iteratively checks hash values in the hash bucket to look up a pointer to a hash entry having a matching prefix (3 of Fig. 4.1) and then fetches the hash entry to which the pointer points in the DRAM device (4 of Fig. 4.1). This is the second fetch. The two fetches of the hash bucket and the hash entry are dependent because the hash entry is fetched by using the pointer in the hash bucket. Thus, the number of dependent DRAM accesses for an exact matching operation is 2. Finally, it checks a name in the hash entry to validate whether the name in the hash entry is the queried name or not only if the hash entry is found (5 of Fig. 4.1). An advantage of the sophisticated chained hash table proposed in [10] is that their hash buckets in DRAM devices can be fetched in parallel.

Longest Prefix Matching Algorithm

The number of dependent DRAM accesses is determined by a longest prefix matching algorithm which searches for the longest matching prefix in the hash table corresponding to the queried name. The author compares the number of dependent DRAM accesses for the two promising algorithms: *longest-first search* and *2-stage search* algorithms.

The longest-first search algorithm is a commonly used algorithm. It starts to run the above exact matching algorithm for the name itself, which is its longest prefix, and continues to run the algorithm for a shorter prefix until it finds the longest matching prefix. It ensures that a hash entry which is firstly found during lookup iterations has the longest matching prefix. All the hash buckets can be fetched in parallel but the two fetches of the hash entry and the hash bucket that holds the pointer of the entry are dependent. Thus, the number of dependent DRAM accesses is 2.

The 2-stage search algorithm [10] reduces the average number of iterations by starting the search from a certain shorter prefix (M components) than the queried name. Every FIB entry having a prefix with M components maintains the maximum length of a prefix that starts with this prefix. By checking the maximum length, it determines whether to continue searching for shorter prefixes or for longer ones. The algorithm first accesses a hash entry with a prefix having M components by fetching its hash bucket and hash entry (the first and second fetches). Then, it fetches a hash bucket and a hash for all shorter or longer prefixes in parallel for the search (the third and fourth fetches). Hence, the number of dependent DRAM accesses is 4.

Finally, the author chooses the longest-first search algorithm since the number of its dependent DRAM accesses is fewer than that of the 2-stage search algorithm.

Please note that this section considers only those cases where the exact matching algorithm successfully matches a queried prefix with a prefix in a hash entry. There are two cases where the exact matching algorithm fails. First, the queried prefix is not stored in the hash table. Second, the name in the hash entry is not the queried prefix which is caused by hash collision. In the above cases, the exact matching algorithm is rerun for a shorter queried prefix. How to handle such cases is described later in Section 4.3.2.

4.1.3 Trie-based FIB

A *trie* is a tree-based data structure that stores an associative array. In a way similar to a hash table-based FIB, its keys are prefixes and its values are corresponding interfaces. Interfaces are looked up by traversing vertices from the root vertex. If the whole data structure of a trie-based FIB is not stored in the CPU caches, several vertices are sequentially accessed through pointers from the DRAM device. In the rest of this subsection, the author estimates the number of vertices fetched from the DRAM devices as the number of dependent DRAM accesses.

As a reference data structure, the author selects a binary Patricia trie [11] because its data structure is the most likely to be stored in CPU caches among trie-based FIBs since its memory consumption is the smallest among such FIBs [11]. The author derives the number of DRAM accesses in the following steps: First, the author calculates the probability P_d of a vertex v_d for which the depth is d being in the L3 cache by using R_d, which is the number of bytes fetched to the L3 cache during two consecutive accesses to v_d. $I(v_d)$ denotes the interval of the two consecutive accesses to v_d. If R_d is larger than the L3 cache size S, v_d will be in the DRAM device; otherwise it will be in the L3 cache. Second, R_d is derived from the product of N_d, which is the average number of packets processed during the interval $I(v_d)$, and r, which is the number of bytes fetched to the L3 cache for processing a packet.

Assuming that the probabilities of digits $\{0, 1\}$ in prefixes and names are identical, the probability of v_d being accessed at each longest prefix match operation is derived as $1/2^d$. Thus, v_d is accessed once every $1/(1/2)^d = 2^d$ longest prefix match operations on average. Since Interest and Data packets are one-for-one in principle, the author can assume that one Data packet is processed between two longest prefix match operations. Under this assumption, the average number of Data packets being processed during the interval $I(v_d)$ is 2^d. Therefore, R_d is estimated as $R_d = r2^d$. If $R_d > S$, the vertex v_d is fetched from the DRAM device. The total number of DRAM accesses is derived by adding such DRAM accesses for all vertices between the root vertex and leaf vertices. According to the analytical model [11], the average depth of all leaf nodes, which hold FIB entries, is $\log(n) + 0.3327$, where n denotes the number of FIB entries. The author defines a binary variable B_d and $B_d = 1$ if $R_d > S$, 0 otherwise. By using B_d, the average number of DRAM accesses is derived as $\sum_i^{\lfloor \log(n)+0.3327 \rfloor} B_i$.

According to the analyses conducted in [10], the author sets the number of FIB entries to 13,548,815, which is the number of unique names in the IRCache traces [52]. The author empirically measures r with PMU registers, and the number of bytes fetched from the DRAM device for processing one Data packet is 511 bytes. Assuming a Xeon E5-2699 v4 CPU, of which the L3 cache size is the largest among commercially available CPUs, the author sets the L3 cache size S to 55 Mbytes. In this case, the average depth of the Patricia-based FIB is 24.01, and thus the average number of vertices from the DRAM device is $\sum_i^{\lfloor \log(n)+0.3327 \rfloor} B_d = 7$. This result means that the last 7 vertices are sequentially accessed from the DRAM device among all vertices accessed in one FIB lookup operation. The number of dependent DRAM accesses of the trie-based FIB is greater than that of the hash table-based FIB.

4.1.4 Summary

Finally, the author chooses a hash table as an underlying data structure for FIB and a longest-first search algorithm for a lookup algorithm. To hide the latency to fetch a hash entry of the hash table-based FIB, the author designs a prefetch algorithm for the hash entry in Section 4.3.

4.2 Cache Algorithm for CS

4.2.1 Overview

In this section, the author chooses a cache algorithm appropriate for hiding DRAM access latency from representative algorithms, cache eviction algorithms and cache admission algorithms. Cache eviction determines a victim, i.e., a Data packet evicted from a CS, whereas cache admission decides whether a new Data packet is inserted into the CS or not. Usually, a cache admission algorithm is used with a simple cache eviction algorithm based on FIFO. An important difference from the choice of FIB data structure described in Section 4.1 is that time complexity and cache hit rates should be considered as well. The author chooses a cache algorithm in the following steps: First, the author excludes the FIFO eviction algorithm from the candidates because cache hit rate provided by it are obviously less than the other algorithms. In other words, the other cache eviction and cache admission algorithms, of which cache hit rate is sufficiently high, are candidates and hereinafter, the author does not discuss cache hit rate differences among the candidates. Second, the author excludes some cache algorithms due to their heavy computation time, as described in Section 4.2.2. Finally, the author compares the numbers of dependent DRAM accesses for the other representative cache algorithms.

4.2.2 Cache Eviction Algorithm

Two promising kinds of algorithms to realize cache eviction exist: frequency-based eviction and recency-based eviction. These algorithms exploit the history of requests to Data packets in order to improve cache hit rates. The author excludes frequency-based eviction algorithms such as WLFU [55] and LFU-DA [56] because it is well known that updating the priority queue, which maintains the numbers of requests, requires computation with logarithmic time complexity. Representative recency based eviction algorithms require only computation with constant time complexity and hereinafter, the author compares the number of dependent DRAM accesses for them.

Recency-based eviction exploits the recency of Data packets, which is defined as the elapsed time from when the last request for a Data packet arrives to the current time, as a history. There are two kinds of data structures representing the history of recencies. Some algorithms such as LRU, LIRS [57] and ARC [13] use a doubly linked list where nodes corresponding to Data packets stored in a CS are ordered according to their recencies, called an LRU list. These algorithms reorder nodes in the LRU list every time a request for a Data packet is received. Other algorithms such as

CLOCK [58], CAR [59] and CLOCK-Pro [60] avoid such reordering by adding each node in the list a flag bit, which specifies which Data packet has been hit in the CS recently.

Recency-based Eviction with LRU List

This subsection describes the number of dependent DRAM accesses for recency-based eviction algorithms using only the LRU list [13, 57]. The number of dependent DRAM accesses is the largest when the request hits the CS as described below. First, the eviction algorithm fetches the hash bucket of the hash table storing the cached Data packet (the first fetch) after computing the hash value of the queried name. Then, it fetches the hash entry (the second fetch) which holds the pointer to the corresponding node in the LRU list. Then, it moves this node corresponding to the hit Data packet to the head of the LRU list. For this movement, it fetches this node and the head node in the LRU list in parallel (the third fetch) and then fetches the next and previous nodes of this node (the fourth fetch). Hence, the number of dependent DRAM accesses is 4.

Recency-based Eviction with Flag Bits

A recency-based eviction algorithm [58–60] with flag bits does not need to maintain an exact recency history. It searches the LRU list to find the node of which the flag bit is reset when a received Data packet is inserted and a victim Data packet is deleted to and from the CS, respectively, whereas it just sets the flag bit of the node corresponding to the Data packet stored in the CS. If the LRU node list was pinned at SRAM devices, such search would incur low computational overhead. However, since the LRU list is stored in DRAM devices, the algorithm needs to fetch nodes in the LRU list sequentially to check their flag bits and hence the number of dependent DRAM accesses is equal to the number of fetched nodes. Song et al. investigated the average number of traversals of nodes in the LRU list to find a victim Data packet in [60] and revealed that CLOCK-Pro serially traverses an average of 16.6 list nodes. This means that the average number of dependent DRAM accesses is 16.6. The author concludes that the recency-based eviction algorithm using only the LRU list [13, 57] is better than that with flag bits [58–60].

4.2.3 Cache Admission Algorithm

Two promising kinds of algorithms to realize a cache admission exist: frequency-based admission and recency-based admission. Recency-based admission such as 2Q [61] causes a similar number of dependent DRAM accesses as recency-based eviction since it handles an LRU list when a request hits a CS. Therefore, in the rest of this subsection, the author focuses on only frequency-based admission.

Frequency-based admission is generally used in conjunction with simple cache eviction such as FIFO. The author assumes that the underlying cache eviction for cache admission is FIFO eviction. Although a common objective of both frequency-based eviction and admission is to identify popular/unpopular Data packets by exploiting frequency, there is a difference between their computations. The eviction identifies the most unpopular Data packet in a CS, whereas the admission only decides whether a new Data packet is more popular than one of the Data packets in a CS. Note that here the word "popular" means that a Data packet is likely to be requested in the future. The above difference means the admission has an advantage in terms of computational overhead because it does not need to maintain a priority queue where Data packets are ordered according to their frequencies, which requires an operation with logarithmic time complexity. That is, the frequency-based admission requires only computation with constant time complexity.

A Representative frequency-based cache admission algorithm is TinyLFU [27]. TinyLFU uses a counting bloom filter to record frequencies of Data packets stored in a CS and they are indexed by the hash values of names of Data packets. FIFO eviction uses a simple ring buffer as a queue to store Data packets and their names and the hash values of their names are also stored in the ring buffer. First, the TinyLFU admission algorithm fetches the frequency of a new Data packet after computing the hash value of its name (the first fetch). Then, it compares the frequency to the frequency of a victim Data packet as follows: To obtain the hash value of the name of the victim Data packet, it fetches the node of the victim Data packet, i.e., the tail node, in the FIFO ring buffer (the second fetch). Then, it fetches the frequency of the victim Data packet by using the hash value (the third fetch). The first and the second fetches are operated in parallel because they are independent. When the first fetch occurs, the address of the tail node in the FIFO ring buffer is obviously known. In contrast, the second and third fetches are dependent. Hence, the number of dependent DRAM accesses is 2.

4.2.4 Summary

Finally, the author selects TinyLFU admission with FIFO eviction for the NDN software router of this chapter. With respect to the cache hit rate, Gil et al. evaluated cache hit rates with TinyLFU and other sophisticated cache eviction algorithms in [27]. The evaluation revealed that TinyLFU admission achieves high cache hit rates comparable to ARC, LIRS and WLFU evictions under realistic traffic traces of various services such as YouTube video and Wikipedia. According to the evaluation result, the author concludes that TinyLFU admission is one of the best algorithms to provide both fast computation and a high cache hit rate.

4.3 Preferch-Friendly Packet Processing

In this section, the author first identifies data fetches causing instruction pipeline stalls which cannot be hidden by a conventional packet processing flow. Then, the author designs a prefetch-friendly packet processing flow to circumvent instruction pipeline stalls caused by the identified data fetches. Finally, the author experimentally evaluates the performance gains obtained by using the prefetch-friendly packet processing.

4.3.1 Unhidden Data Fetches in Conventional Packet Processing

Since data pieces stored on the DRAM device are only the hash buckets and hash entries of the three hash tables, i.e., a CS, a PIT and a FIB, it is important to know precisely both when a thread of handling an NDN packet knows the address of a hash bucket or a hash entry and when it fetches the bucket or the entry. Figure 4.2 shows typical flows for handling an Interest packet and a Data packet. The figure is based on the exact matching algorithm for looking up a hash table, as shown in Fig. 4.1. The flow includes of computation and fetches of hash buckets and entries. The computation includes hash computation, computation of checking a hash bucket to know the address of the corresponding hash entry and that of processing a CS, a PIT or a FIB. The two example flows in Fig. 4.2 are the most time-consuming among those handling Interest and Data packets. In the case of an Interest packet, all three hash tables are accessed as follows: First, the thread computes the hash values of all prefixes in the name of the received Interest packet since the hash values are used as indexes to access the three hash tables. Next, the thread search the hash table of the CS. Since the Interest packet does not hit the CS, the thread next searches the hash table of the FIB for the outgoing interface, after searching the PIT and creating an entry at the PIT. Though PIT entries created in the PIT must be written back to the DRAM device, such write back operations can be performed in parallel with other computations. Thus, the author does not discuss the DRAM access latency for such write back operations.

In Fig. 4.2, the two packets are received in sequence and the author calls them the first packet and the second packet, respectively. The suffixes 1 and 2 mean the first packet and the second one, respectively. From Fig. 4.2, the author obtains two observations on instruction pipeline stalls which are difficult to be hidden. First, the first hash bucket (Bf_1 after $Hash_1$ or Bf_2 after $Hash_2$ in the figure) cannot be perfected because its address is determined by the hash computation just before. Apparently, there is no time between the address computation, i.e., hash computation, and the data

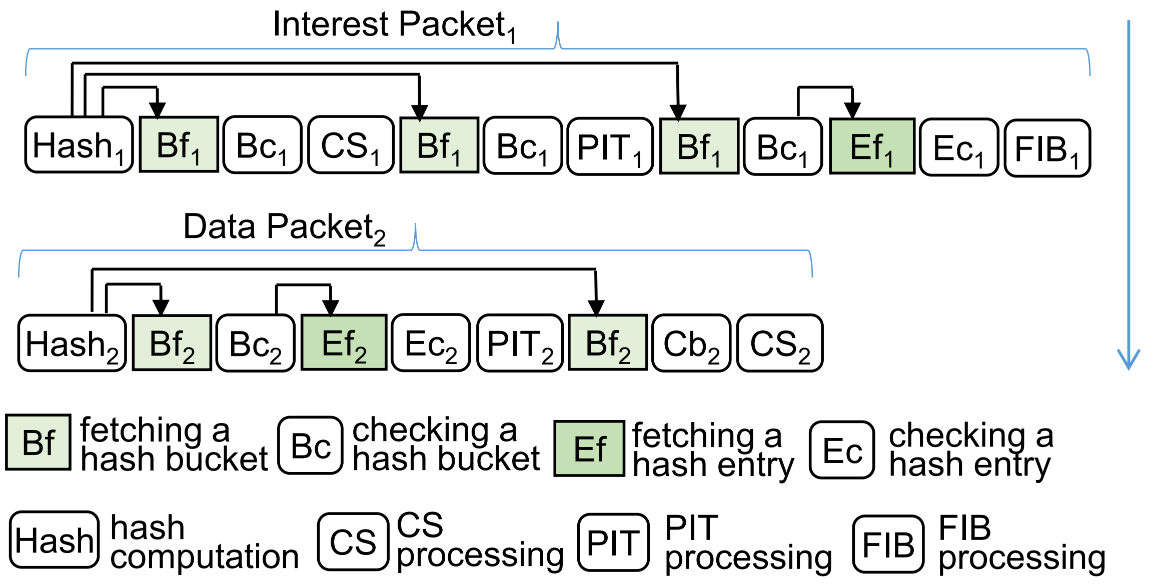


Figure 4.2: Conventional packet processing flow

fetch. Second, the hash entry of a PIT or a FIB cannot be fetched after reading its address which is stored in the corresponding hash bucket. For example, the addresses of hash entries Ef_1 and Ef_2 are stored in the hash buckets Bc_1 and Bc_2, respectively.

4.3.2 Prefetch Strategies

To hide instruction pipeline stalls caused by fetches of the first hash bucket and the hash entry, the author designs the two prefetch strategies, i.e., *Pre-Hash Computation* and *Speculative Hash Table (HT) Lookup*, based on the following assumptions: Since four DDR4 DRAM channels are provided and each channel supports eight DRAM ranks each of which is able to independently access a cache line-sized data piece [62], it is reasonable for multiple data pieces in the DRAM device to be fetched in parallel. This implies that up to 32 data pieces are prefetched in parallel while the thread is running some computations whose CPU cycles are greater than those consumed by one data fetch from the DRAM device.

Pre-Hash Computation

First, Pre-Hash Computation is inspired by the first observation of the previous subsection. In order to hide instruction pipeline stalls caused by a fetch of the first hash bucket, the author should leverage the computation of a subsequent packet. It means that a sequence of computations should be

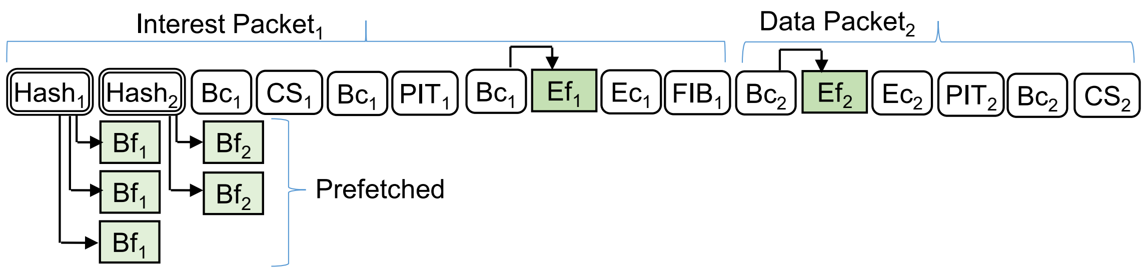


Figure 4.3: Prefetcth-friendly packet processing flow with Pre-Hash Computation

reconsidered for multiple packets. Figure 4.3 shows a prefetch-friendly packet processing flow with Pre-Hash Computation. The author chooses to handle two consecutive packets because handling a large number of packets incurs state information for handling multiple packets, which may be stored on the DRAM device in a worst case scenario. Thus, the author precomputes the hash value of the subsequent packet just after finishing the hash computation of the first packet. In parallel, hash buckets of the first packet are fetched from the DRAM device illustrated by Bf_1 in Fig. 4.3. The hash computation of the second packet hides instruction pipeline stalls caused by the fetch of the first hash bucket.

Speculative Hash Table Lookup

Second, after fetching the above-mentioned hash buckets, it becomes possible for the hash entries of all the hash tables, i.e., the PIT and FIB, of the first packet to be fetched from the DRAM device by checking the hash values in the hash buckets. Thus, all the hash entries, including those which may not be accessed due to the longest prefix match results, are speculatively fetched even if some of the hash entries are not used later. The strategy is called Speculative HT Lookup. Figure 4.4 shows the prefetch-friendly packet processing flow with Speculative HT Lookup. Speculative HT Lookup is applied to all hash buckets and hash entries which correspond to shorter prefixes than the queried name of the packet to hide the instruction pipeline stalls caused by the longest-first search which is described in Section 4.1.2. Please note that the flow in the figure includes the Pre-Hash computation strategy. Speculative HT Lookup is applied to hash entries for the subsequent packet after fetching hash buckets for it.

If the longest-first search algorithm naively uses Speculative HT Lookup, some computations of the exact matching algorithm are unnecessarily executed for shorter prefixes than the matching prefix. Specifically, the exact matching algorithm is repeatedly executed by finding a prefix in a hash table which satisfies the two conditions which are described in Section 4.1.2: The first condition is that a

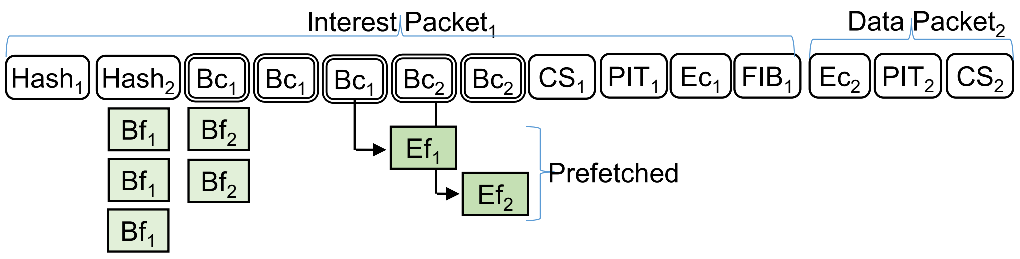


Figure 4.4: Prefetcth-friendly packet processing flow with Speculative Hash Table Lookup (and Pre-Hash Computation)

hash bucket stores the address of a hash entry. The second condition is that the hash entry stores the name of a queried prefix. If either condition does not hold, the longest-first search algorithm executes the exact matching algorithm for a shorter prefix by one component. During the iteration, computations of checking hash buckets are performed in sequence, whereas all hash buckets are fetched in parallel. This means that unnecessary exact matching operations consumes CPU cycles in proportion to the number of prefixes which are shorter than the matching prefix. The problem is that the above unnecessary CPU cycles are not negligible, as described below: The number of CPU cycles consumed by the above operation for one prefix on a Xeon E5-2699 v4 CPU are 48, whereas the number of CPU cycles consumed by one hash bucket fetch is 115. In the case of the prefix distribution which this thesis assumes in Section 3.2.2, the average prefix length and matching prefix length are 7 and 4, respectively, and thus the above unnecessary computation of the exact matching algorithm is executed 3 times. This incurs about 144 CPU cycles.

To circumvent this problem, the author introduces a short-circuit evaluation of the result of exact matching for each prefix. First, the longest-first search algorithm prefetches all the hash buckets of all the prefixes. Then it repeatedly checks the first condition for hash buckets from the longest prefix to the shortest one until the condition is satisfied. When such a hash bucket is found, it fetches the hash entry. If the hash entry has the name of the queried prefix, i.e., the queried prefix matches this prefix, the longest-first search algorithm terminates. This avoids the above unnecessary exact matching operations. Otherwise the longest-first search algorithm continues to find the next hash entry by iteratively checking the hash bucket of shorter prefixes. In many cases, the longest-first search algorithm, however, terminates when the first hash entry is fetched, as the author will empirically investigate in the following section.

4.4 Performance Evaluation

First, the author evaluates how the proposed prefetch-friendly packet processing eliminates instruction pipeline stalls due to DRAM accesses. Then, the author evaluates the forwarding speed of NDN routers with conventional packet processing and the proposed prefetch-friendly packet processing. For the evaluation, the author employs the same experiment settings that the author used in the bottleneck analysis described in Section 3.2.2

4.4.1 CPU Cycles in Single-Threaded Case

First, the author evaluates the CPU cycles for processing packets in the single-threaded case to focus on evaluating how Pre-Hash Computation and Speculative HT Lookup contribute to hiding instruction pipeline stalls due to DRAM accesses. For comparison purposes, the author uses the four variants of the NDN software routers: *Naive*, *Bucket Prefetch*, *Pre-Hash*, and *SHTL*, to which existing and the proposed optimization techniques are incrementally applied. *Naive* is the NDN software router which is based on the router platform in Section 3.1. It is designed according to the design rationales presented in Sections 4.1 and 4.2 but it does not have any prefetch techniques. This is equivalent to the software proposed by So et al. [10] except that it uses the longest-first search algorithm rather than the 2-stage search algorithm and it employs TinyLFU admission rather than FIFO eviction. *Bucket Prefetch* is based on *Naive* but it prefetches hash buckets for arriving packets according to the existing prefetch technique proposed by So et al. [10]. *Pre-Hash* is based on *Naive* but it employs Pre-Hash Computation proposed in Section 4.3. *SHTL* represents the NDN software router that has Speculative HT Lookup in addition to Pre-Hash Computation.

Figure 4.5 shows the CPU cycles for processing one Interest and Data packet. *I-miss* represents the CPU cycle for processing one Interest packet in the case of CS misses. That is, all the three hash tables of the CS, the PIT and the FIB are accessed in the I-miss flow. Bucket Prefetch reduces the number of CPU cycles by 15.03% compared with Naive. Pre-Hash reduces the number of CPU cycles by 9.16% compared with Bucket Prefetch because the hash buckets of the CS are efficiently prefetched with Pre-Hash Computation. In addition to Pre-Hash Computation, SHTL further reduces the number of CPU cycles by 7.82% compared to Pre-Hash Computation since Speculative HT Lookup hides instruction pipeline stalls for fetching hash entries of the FIB. Finally, SHTL reduces the number of CPU cycles by 28.85% compared to Naive.

In contrast to the case for Interest packets, the CPU cycles for processing one Data packet do not decrease when Bucket Prefetch and Pre-Hash are used. This phenomenon can be explained

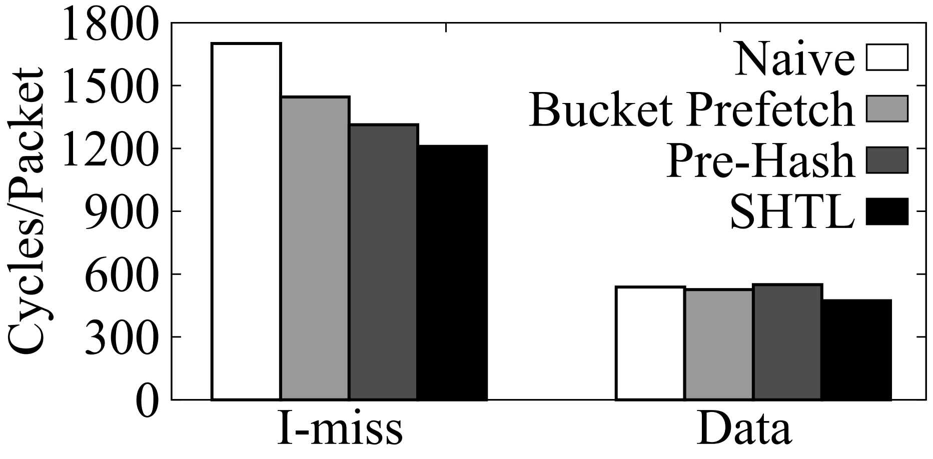


Figure 4.5: The number of CPU cycles for processing one NDN packet in the single-threaded case

as follows: In the experiments, Data packets are returned shortly after the corresponding Interest packets are processed. When a Data packet arrives at the router, hash buckets and entries necessary for processing the Data packet are likely to be on the CPU caches since they are fetched to the CPU caches for processing the corresponding Interest packet. Bucket Prefetch and Pre-Hash, therefore, do not contribute to reducing the number of CPU cycles. In contrast to the experiments, if Data packets are not returned promptly, some hash buckets and entries would not be in the CPU caches. In this case, the number of CPU cycles of Naive increases because it does not prefetch any data from the DRAM device. In contrast, those of Bucket-Prefetch and Pre-Hash do not change. That is, CPU cycle reduction due to Bucket Prefetch and Pre-Hash is slightly underestimated in the result in Fig. 4.5.

In contrast, SHTL prefetches such hash buckets and hash entries irrespective of whether they are still in the CPU cache or not. SHTL reduces the number of CPU cycles by about 11.98% compared to Naive, Bucket Prefetch, and Pre-Hash because instruction pipeline stalls due to accesses to hash entries and buckets of a replaced Data packet in the CS is hidden by SHTL.

4.4.2 CPU Cycles in Multi-Threaded Case

Next, the author evaluates the reduction in CPU cycles resulting from prefetch-friendly packet processing in the case of multi-threaded software. Figure 4.6 shows the number of CPU cycles required for processing one packet in the case where the software is running with 20 threads. The number of CPU cycles of SHTL, which includes both Pre-Hash Computation and Speculative HT Lookup, does not increase compared with the single-threaded case, whereas for Naive and Bucket Prefetch, the number of CPU cycles in the multi-threaded case is larger than that in the single-threaded

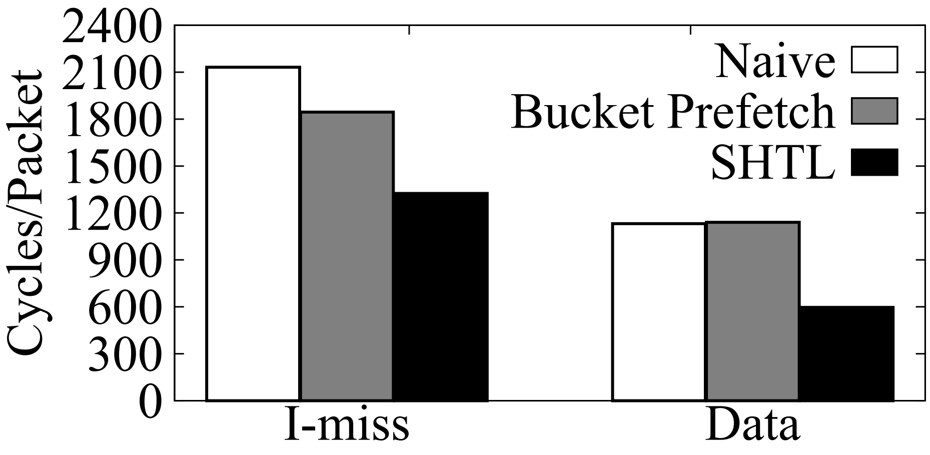


Figure 4.6: The number of CPU cycles for processing one NDN packet in the multi-threaded case (20 threads)

case. As shown in Table 3.6, the DRAM access latency increases when the number of threads becomes greater. Naive and Bucket Prefetch have unhidden DRAM accesses and the latency of the unhidden DRAM accesses causes the increase in the total number of CPU cycles required for processing an Interest packet and a Data packet. In contrast, SHTL hides most of the DRAM accesses. More precisely, SHTL does not hide accesses to hash entries of the CS since the software does not have enough amount of computation between the accesses to a hash bucket and a hash entry of the CS to hide the accesses, but SHTL hides all the other DRAM accesses. Therefore, the CPU cycles required for processing packets are approximately constant regardless of the number of threads. As a result, the absolute difference between the CPU cycles of Bucket Prefetch and SHTL becomes larger compared to the single-threaded case due to the larger DRAM access latency.

4.4.3 Forwarding Speed

Finally, the forwarding speed of the NDN software routers is evaluated. To show the worst case forwarding speed, the author chooses names for Data packets such that all the Interest packets miss the CS since I-miss is computationally heavier than I-hit, as shown in Table 3.2.

Figure 4.7 shows the forwarding speed against the number of threads. The forwarding speed of SHTL is approximately linear to the number of threads, whereas the speeds of Naive and Bucket Prefetch are not. SHTL hides most of the DRAM accesses, whereas Bucket Prefetch still has unhidden DRAM accesses. As was shown in Fig. 4.6, the number of CPU cycles needed for processing packets increase in the case of Naive and Bucket Prefetch due to the unhidden DRAM accesses. This

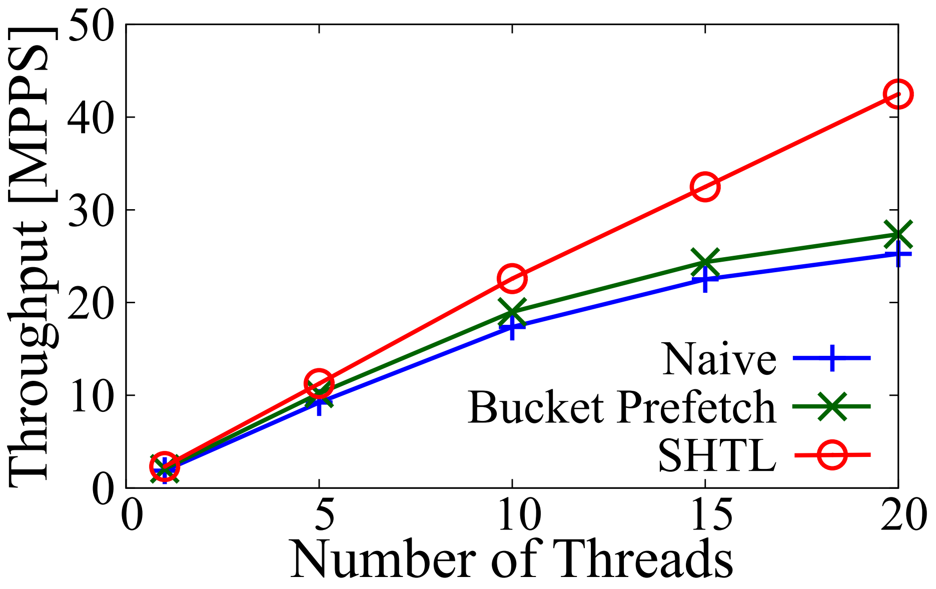


Figure 4.7: The forwarding speed against the number of threads

degrades the forwarding speed. In contrast, SHTL hides most of the DRAM accesses and therefore the forwarding speed is not degraded. Please note that although Fig. 4.7 shows that the forwarding speed increases as the number of threads increases, the speed does not linearly increase 22 threads because the maximum capacity of DRAM I/O is not enough for the speed. As analyzed in Section 3.2, if the number of bytes read from and written to DRAM devices per second reaches the bandwidth of DRAM I/O channels, the forwarding speed does not increase even if the number of threads increases. On a computer used in Fig. 4.7, when the number of threads is 40, the number of bytes read from and written to DRAM devices per second just reaches the maximum capacity of DRAM I/O channel.

SHTL finally realizes a forwarding speed of up to 42.49 MPPS. The author believes that it is one of the fastest NDN software routers realized on a computer. Note that the forwarding speed of 42.49 MPPS is not the upper bound of the router since the author uses only one of two CPUs. If additional NICs can be installed on this computer, the forwarding speed will be scaled up to about 80 MPPS.

4.4.4 Discussion

Short Circuit Evaluation Fail in Speculative HT Lookup

Speculative HT Lookup employs short circuit evaluation for the longest-first search algorithm, as described in Section 4.3.2. The short circuit evaluation could incur false positives such that a fetched hash entry does not store the name of a queried prefix. In order to reduce such false positives, the author employs a collision-resistant hash function, as suggested by So et al. in [10]. Specifically, the author chooses the *SipHash* function because it provides a good balance between collision resistance and computation time. As a result, in the evaluation, no failure in the prefetching of a hash entry was observed.

Cache Collision Due to Prefetching Data Pieces of Subsequent Packet

Another concern is that in Pre-Hash Computation, prefetching hash buckets and entries of a subsequent packet, i.e., the second packet, to the CPU cache would evict or overwrite those of a prior packet, i.e., the first packet from the CPU cache; however, such evictions would not occur because the average size of newly fetched data for processing the two packets is considerably smaller than that of the CPU cache. The average size of data fetched for processing two packets can be estimated from the DRAM channel bandwidth for processing a packet measured in Section 3.2.3 and the size is 1601.4 (800.7 bytes/packet), whereas the average size of the L3 cache for one CPU core is approximately 2.5 Mbytes because 55 Mbytes is shared with 22 CPU cores [45].

4.5 Conclusion

In this chapter, the author identified the ideal form of an NDN software router on computers in the following steps: 1) the author conducted a detailed study of the existing techniques for high-speed NDN packet forwarding and compiled them into the design rationale toward the realization of an ideal NDN software router. 2) The author conducted microarchitectural and comprehensive bottleneck analyses on the NDN software router and revealed that to hide the DRAM access latency is vital to the realization of an ideal NDN software router. 3) The author proposed two prefetch-friendly packet processing techniques to hide the latency. Finally, the prototype implemented according to the rationale and the prefetch-friendly packet processing techniques demonstrated that a forwarding speed exceeding 40 million packets per second (MPPS) could be achieved on a single computer.

Chapter 5

A Lightweight Cache Admission Algorithm for Optimizing Time Complexity

In the previous chapter, the author tackled the unhidden DRAM access latency, which is one of two bottlenecks for high-speed NDN forwarding, identified in Chapter 3. In this chapter, the author tackles wasteful computation of cache algorithm, which is the rest of the two bottlenecks. In Chapter 3, the author found that cache algorithm for NDN routers needs to improve cache hit rate with fast computation. Nevertheless, the cause of heavy computation of cache algorithm itself is unclear.

To understand the cause of heavy cache algorithm computation, the author firstly analyzes computation time of two types of cache algorithms, i.e., cache eviction and admission algorithms and finds that the cache eviction always performs cache insertions for unpopular Data packets while a function of the cache insertion is relatively time-consuming among functions of NDN packet processing. This finding means that cache insertion of unpopular Data packets which are not hit later wastes many CPU cycles.

The finding obtained in Chapter 3 and that described above imply that cache algorithms for NDN routers need to increase cache hit rate and reduce unpopular Data packet insertion rate simultaneously without incurring computation overheads. According to this implication, the author designs lightweight cache admission algorithm called Filter so that both high cache hit rate and low cache insertion rate are achieved with fast computation. The rest of this chapter is organized as follows: The author introduces a problem of wasteful caching computation by analyzing computation time of cache eviction and admission algorithms in Section 5.1 The author proposes an NDN packet forwarding scheme with the caching admission based on Filter to optimize the wasteful cache computation in Section 5.2. In Section 5.3, the author analytically investigates cache hit rate of caching with Filter and then, the author evaluates performance of Filter and the proposed NDN packet forwarding scheme with Filter in Section 5.4. Finally, the author concludes this chapter in Section 5.5.

5.1 Computation Time Analysis of Cache Algorithms

In this section, the author analyzes computation time of two types of cache algorithms, i.e., cache eviction and admission, to show that cache admission is better than cache eviction by analyzing packet flows of the NDN software described in Chapter 3.

5.1.1 NDN Packet Processing with Cache Algorithms

A cache algorithm is classified into cache eviction and cache admission. Cache eviction determines a victim, i.e., a Data packet evicted from the cache when an incoming Data packet is inserted into the cache, i.e., the CS, whereas cache admission decides whether an incoming Data packet is inserted into the cache or not. A common objective of them is to identify popular/unpopular Data packets by using a history of incoming requests to Data packets. The word “popular” means that a popular data packet is likely to be requested in the future.

This section compares computation time of NDN packet processing flows with cache eviction and cache admission algorithms by carefully analyzing how individual function blocks consume CPU cycles of the flows.

Packet Processing Flows with Cache Eviction

This section describes packet processing flows with cache eviction in the cases of both cache hits and misses according to the diagram illustrated in Figure 5.1, where blocks B_2 and B_{12} are used for cache eviction. First, block B_1 receives and decodes an incoming Interest packet. B_2 updates the history of incoming requests to Data packets, which is later used for cache eviction, and B_3 checks whether a data piece of a Data packet corresponding to the Interest packet is stored in the CS. If the data piece is in the CS, B_4 fetches it and composes the Data packet with it, and B_{11} sends back the Data packet to the incoming interface.

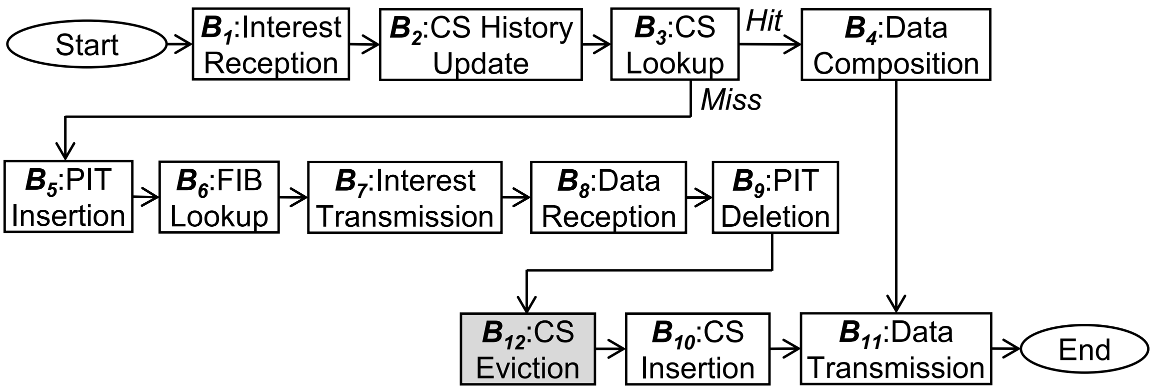


Figure 5.1: Packet processing flow with cache eviction

Otherwise, i.e., if the Interest packet does not hit the CS, B_4 inserts the incoming interface into the PIT. After B_5 gets an outgoing interface from the FIB by performing longest name prefix matching, B_6 sends the Interest packet to the outgoing interface. When a returned Data packet corresponding to the Interest packet arrives at the router, B_6 receives and decodes the Data packet, and then B_7 gets and deletes an outgoing interface from the PIT. After B_{12} chooses and evicts a victim from the cache according to the history of incoming requests to Data packets, B_8 inserts the incoming Data packet into the CS. Finally, B_9 sends the Data packet to the outgoing interface.

The author groups the blocks so that the average CPU cycles spent for handling a pair of an Interest and a Data packet are calculated with the two parameters, the cache hit rate and the cache insertion rate, as below:

- G_1 is the set of blocks executed always for each Interest packet: B_1, B_2, B_3 and B_{11}.
- G_2 is the block executed at a cache hit: B_4.
- G_3 is the set of blocks executed at a cache miss except for the blocks for cache eviction, i.e., B_{12} and B_{10}: B_5, B_6, B_7, B_8 and B_9.

The average CPU cycles spent for processing a pair of an Interest and a Data packet $C^{\text{Avg}}_{\text{Evi}}$ are defined by Eq. (5.1), where p^{Hit} is the cache hit rate, $C_{b,i}$ is the CPU cycles of block B_i, and $C_{g,i}$ is the CPU cycles of group G_i. p^{Hit} is defined as the probability that Interest packets hit the cache.

$$C^{\text{Avg}}_{\text{Evi}} = C_{g,1} + p^{\text{Hit}} C_{g,2} + (1 - p^{\text{Hit}})(C_{g,3} + C_{b,12} + C_{b,10}) \tag{5.1}$$

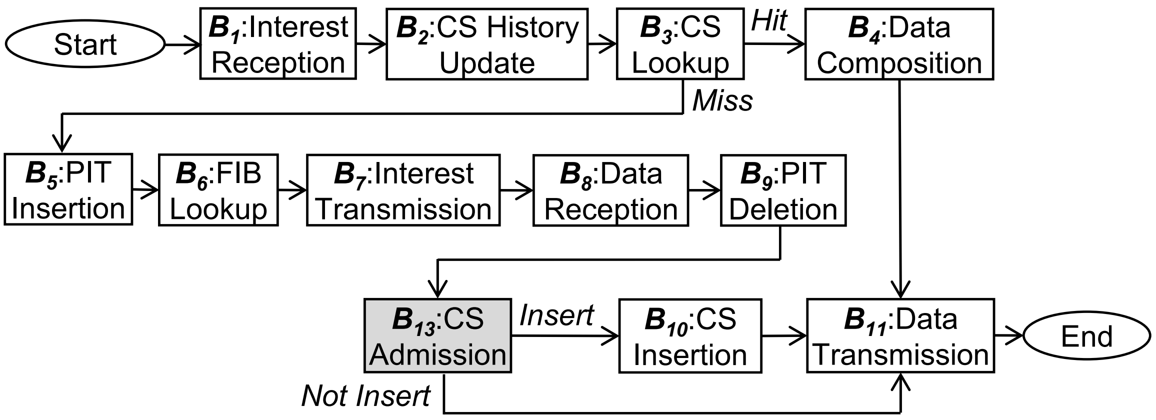


Figure 5.2: Packet processing flow with cache admission

Packet Processing Flows with Cache Admission

The packet processing flows with cache admission is illustrated by the diagram in Figure 5.2. The flow in the case of a cash hit is the same as the flow with cache eviction illustrated in Figure 5.1. Precisely, the CPU cycles of B_2 for cache eviction and admission are slightly different. On the contrary, in the case of a cash miss, the flows with cache admission are explicitly different from those with cache eviction as follows. First, block B_{12} for cache eviction is replaced with B_{13} for cache admission. Please note that the CPU cycles of B_{13} are less than those of B_{12}. Second, B_{10} for cache insertion is executed only when B_{13} decides to insert a Data packet into the CS.

The average CPU cycles $C^{\mathrm{Avg}}_{\mathrm{Adm}}$ are defined by Eq. (5.2), where $p^{\mathrm{Insertion}}$ is the cache insertion rate. $p^{\mathrm{Insertion}}$ is defined as the probability that Data packets are inserted into the cache by block B_{13}.

$$C^{\mathrm{Avg}}_{\mathrm{Adm}} = C_{g,1}+p^{\mathrm{Hit}}C_{g,2} + (1 - p^{\mathrm{Hit}})(C_{g,3} + C_{b,13} + p^{\mathrm{Insertion}}C_{b,10}) \tag{5.2}$$

5.1.2 Cache Eviction vs. Cache Admission

Assuming that the cache hit rates of both cache eviction and cache admission are same, the difference between the average CPU cycles of cache eviction and admission, i.e., $C^{\mathrm{Avg}}_{\mathrm{Evi}} - C^{\mathrm{Avg}}_{\mathrm{Adm}}$, is $(1 - p^{\mathrm{Hit}})((C_{b,12} - C_{b,13}) + p^{\mathrm{Insertion}}C_{b,10})$. If the difference is positive, the average CPU cycles $C^{\mathrm{Avg}}_{\mathrm{Adm}}$ is smaller than $C^{\mathrm{Avg}}_{\mathrm{Evi}}$, which means that cache admission is faster than cache eviction.

To validate the above claim preliminarily, the author estimates the average CPU cycles of the ARC eviction algorithm and the TinyLFU admission algorithm and the difference between the average CPU cycles of them based on the measurements of the blocks' CPU cycles in Section 5.4. Precisely, the author does so, assuming that the cache hit rate p^{Hit} of all cache algorithms of ARC and TinyLFU is 30% and that the cache insertion rate $p^{\mathrm{Insertion}}$ of TinyLFU is 10%. The measurement results in Section 5.4 show that the CPU cycles $C_{b,12}$ of ARC, the CPU cycles $C_{b,12}$ of TinyLFU and the CPU cycles $C_{b,10}$, are 202, 282 and 188 CPU cycles, respectively. Thus the average CPU cycles of cache eviction $C_{\mathrm{Evi}}^{\mathrm{Avg}}$ of ARC and those of cache admission $C_{\mathrm{Adm}}^{\mathrm{Avg}}$ of TinyLFU are 1817 and 1705 CPU cycles, respectively, and $C_{\mathrm{Adm}}^{\mathrm{Avg}}$ is 112 CPU cycles smaller than $C_{\mathrm{Evi}}^{\mathrm{Avg}}$.

The author has obtained the following observations from the above estimation results: First, cache admission reduces the CPU cycles of cache eviction by about 6%. Second, however, computation time of deciding whether a Data packet is inserted or not, i.e., the CPU cycles $C_{b,13}$, is larger than the time of deciding a victim, i.e., the CPU cycles $C_{b,12}$. This implies that there is still room for reducing CPU cycles by improving the CPU cycles $C_{b,13}$ of TinyLFU. Thus, in the rest of this chapter, the author focuses on a lighter cache admission algorithm than TinyLFU, whereas it realizes the similar cache hit rate to TinyLFU.

5.2 Design of Cache Admission Algorithm

5.2.1 Overview

The goal of the cache admission algorithm is to identify unpopular Data packets for filtering out them in order to increase the cache hit rate and reduce the cache insertion rate. Obviously, increase in cache hit rate contributes to improving forwarding speed of NDN routers. A few cache admission algorithms have been proposed [23, 24, 27]; however, computation time for filtering out unpopular Data packets is not carefully considered. The author designs a simpler cache admission algorithm, named *Filter*. In the rest of this section, the author firstly describes the design rationale behind Filter, and the author then designs Filter according to the design rationale. Finally, the author compares Filter with a well-known cache admission algorithm, TinyLFU [27], in terms of cache hit rate and computation time.

5.2.2 Design Rationale

The design rationale behind Filter is that accesses to slow memory devices, such as dynamic random access memory (DRAM) devices, should be eliminated since the time spent by one DRAM access accounts for a large part of the entire computation time. For instance, in Chapter 3, the author empirically revealed that one DRAM access spends about 11% of the average computation time of entire NDN packet processing.

The author therefore does not adopt TinyLFU [27] because it is difficult to eliminate DRAM accesses from packet processing of TinyLFU. TinyLFU compares the frequency of an arriving Data packet and that of a victim Data packet, which is chosen among Data packets in the cache, and it inserts the arriving Data packet to the cache only if its frequency is larger than that of the victim Data packet. Though this enables it for TinyLFU to decide whether the arriving Data packet is inserted into the cache or not without any threshold, this incurs unavoidable DRAM accesses due to the fact that the victim Data packet is, in general, in DRAM devices.

In contrast to TinyLFU, Filter is designed so that it avoids accesses to DRAM devices by comparing the frequency of an arriving Data packet with a predefined threshold value, both of which are not stored in DRAM devices, instead of comparing it with the frequency of a victim Data packet. Since the advantage of not accessing to DRAM devices causes a necessity to tune the predefined threshold in compensation for the fast computation, the author will develop an analytical model of Filter in Section 5.3 to clarify a guideline of choosing the threshold.

5.2.3 Design of Filter

Filtering unpopular Data packets is equivalent to admitting only highly popular Data packets. Filter identifies highly popular Data packets on the basis of the intuition that the number of highly popular Data packets is much smaller than that of unpopular ones. To identify highly popular Data packets, Filter identifies Interest packets that appear several times within a certain time window. The main intuition behind Filter is that Interest packets for highly popular Data packets may appear often and such packets will arrive again in the near future. To realize this idea with light computation, the author designs Filter as follows.

Our Filter consists of lightweight computation and two simple data structures to memorize ingress Interest packets: A FIFO queue to store a history of Interest packets, i.e., a sequence of past Interest packets, and a hash table to store the frequency of appearances of the Interest packets within the history. The author refers to the FIFO queue and the hash table as *history queue* and *counter hash table*, respectively. The schematic of Filter is shown in Figure 5.3. The history queue holds hashes of content names in the past q Interest packets, $\{x_{n-1}, x_{n-2}, \ldots, x_{n-q}\}$ in order of their arrivals. x_n is the hash of the content name in the n-th Interest packet and it is used as an index to access the counter hash table. The x_n-th slot of the counter hash table, $H[x_n]$, holds the frequency of appearances of

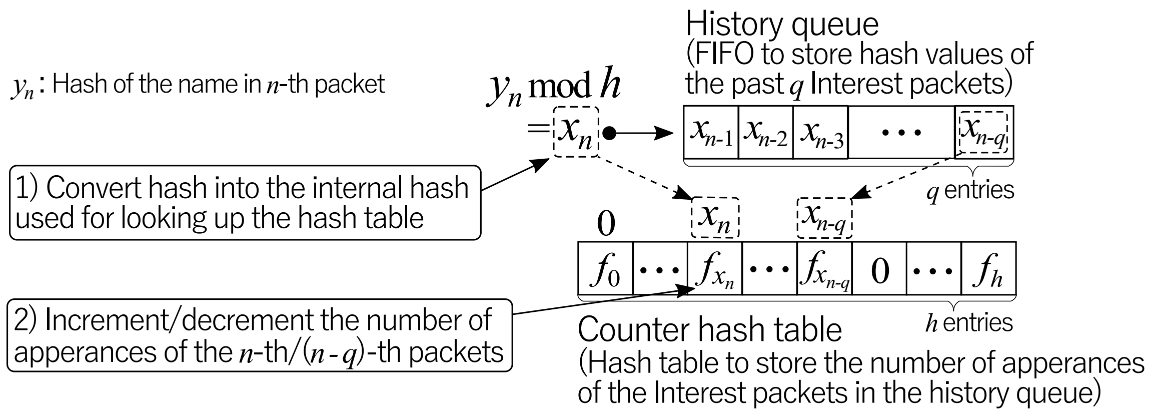

y_n: Hash of the name in n-th packet
$y_n \bmod h = x_n$
History queue (FIFO to store hash values of the past q Interest packets)
x_{n-1} x_{n-2} x_{n-3} ··· x_{n-q}
q entries
1) Convert hash into the internal hash used for looking up the hash table
0 x_n x_{n-q}
f_0 ··· f_{x_n} ··· $f_{x_{n-q}}$ 0 ··· f_h
2) Increment/decrement the number of apperances of the n-th/(n-q)-th packets
Counter hash table
h entries
(Hash table to store the number of apperances of the Interest packets in the history queue)

Figure 5.3: Schematic of data structures of Filter

Interest packets, of which hash is x_n, within the past q Interest packets.

The Filter algorithm is summarized as follows. When an Interest packet arrives (B_2 in Figure 5.2), Filter computes the hash of its content name, x_n, and updates the history queue and the counter hash table, as shown in Figure 5.3. That is, Filter dequeues an entry, x_{n-q}, from the head of the history queue and enqueues x_n to the tail of it. Then, Filter increments $H[x_n]$ and decrements $H[x_{n-h}]$. When a Data packet arrives (B_{13} in Figure 5.2), Filter determines whether the packet should be inserted into the cache or not by checking the counting hash table without updating the history queue and the counter hash table. That is, if its frequency of appearances, $H[x_n]$, satisfies $H[x_n] \geq \theta$, where θ is a threshold parameter, then the packet is forwarded to the *CS Insertion* function block (B_{10}), otherwise B_{10} is bypassed and the packet is forwarded directly to the *Data Transmission* function block (B_{11}).

To keep the computation of Filter fast, hash collisions in the counter hash table are not resolved by allowing a certain degree of false positives, i.e., several Interest packets that have different content names with the same hash are mapped to the same slot in the counter hash table. However, such false positives rarely occur as described below: The probability of false positives p^{FP} can be derived by using well-known analytical models for hash collisions [63]. Given the assumption that hash values are perfectly random, p^{FP} is calculated as $p^{\mathrm{FP}} = 1 - (1 - 1/h)^q$, where q and h are the length of the history queue and the counter hash table, respectively. Due to space limitations, the author omits the details of the equation. In the case of $q = 10^4$ and $h = 2^{24}$, p^{FP} is 0.0596%, whereas the history queue with 10^4 64-bit hash values and the counter hash table with 2^{24} 16-bit integer values consume only 32 Mbytes. Thus, the author concludes that p^{FP} is sufficiently low.

5.2.4 Comparison with TinyLFU

Finally, the author discusses the advantage and the disadvantage of the proposed Filter for existing cache admission algorithms focusing on cache hit rate and computation time. As a reference for Filter, the author chooses TinyLFU [27] among several cache admission algorithms [23, 24, 27] because its implementation is highly optimized, whereas another algorithms [23, 24] have focused on only high cache hit rate.

The key advantage of Filter is that its computation time is shorter than that of TinyLFU. The author implements Filter and TinyLFU on the proposed NDN forwarding scheme, and then measure how they spend the CPU cycles for decision of cache insertion at block B_{13} in Figure 5.2. Please note that the measurement conditions are same as those summarized in Section 5.4.1. The measured number of the CPU cycles for Filter and that for TinyLFU is 28 and 288, respectively. Filter reduces computation time of TinyLFU because it eliminates DRAM accesses by using the predefined threshold value instead of frequency of the victim Data packet in TinyLFU, as described in Section 5.2.2.

The disadvantage of Filter is that if Filter selects an ineffective threshold value, it misidentifies popular Data packets as unpopular ones in its insertion decision. That is, Filter with the ineffective threshold value causes lower cache hit rate than TinyLFU. In the next section, the author selects the threshold value for Filter based on its characteristic analysis, and then in Section 6, the author evaluates cache hit rates of Filter with the selected threshold and TinyLFU.

5.3 Characteristic Analysis of Filter

5.3.1 Overview

In this section, the author analyzes how parameters of Filter, i.e., the threshold and the history length affect the performance of Filter. The author develops an analytical model of Filter, focusing on two performance metrics: *cache hit rate* and *cache insertion rate*. Cache hit rate, which is defined as a probability that Data packets requested with incoming Interest packets exist in the cache, is one of the most important metrics for both cache eviction and cache admission algorithms. In addition, in the case with a cache admission algorithm, Data packets are inserted into the cache only when the cache admission algorithm admits them, and hence cache insertion rate, which is defined as the probability that incoming Data packets are inserted to the cache, is also an important metric. To simplify notation, the author refers to cache hit rate and cache insertion rate as *hit rate* and *insertion rate*, respectively.

5.3.2 Analysis Model

Firstly, the author computes an insertion rate for Data packet c, $p_c^{\text{Insertion}}$, wherein an insertion rate is a probability that a received Data packet c is selected to be inserted into the cache. For simplicity, the author assumes that Interest packets for Data packets c arrive according to a Poisson process with rate λ_c. The insertion rate $p_c^{\text{Insertion}}$, is derived with the conditional probability that the Interest packets for c appears $\theta - 1$ times and more in the history queue subject to the next arrival of the Interest packet. Since the future arrival events are independent from the past ones, the insertion rate is equivalent to $\mathbb{P}[X_c \geq \theta - 1]$, where X_c is a stochastic variable to express the number of arrived Interest packets for c in the history queue. Since the history queue holds the past q Interest packets, the time window W, which is the time difference between the most and least recent Interest packet arrivals in the history queue, is approximately $W = q/\sum_c \lambda_c$. To simplify the notation, the author defines λ'_c as $\lambda'_c = \lambda_c/\sum_k \lambda_k$, which is the arrival rate for Data packet c normalized by that for all Data packets. By using the cumulative distribution function of the Poisson distribution, $p_c^{\text{Insertion}}$ is derived as

$$\begin{aligned} p_c^{\text{Insertion}} &= \mathbb{P}[X_c \geq \theta - 1] = 1 - \mathbb{P}[X_c \leq \theta - 2] \\ &= 1 - \left(\exp\left(-q\lambda'_c\right) \sum_{k=0}^{\theta-2} \frac{\left(q\lambda'_c\right)^k}{k!} \right). \end{aligned} \tag{5.3}$$

The average insertion rate for all Data packets $p^{\text{Insertion}}$ is

$$p^{\text{Insertion}} = \frac{\sum_{c \in G} \lambda_c (1 - p_c^{\text{Hit}}) p_c^{\text{Insertion}}}{\sum_{c \in G} \lambda_c (1 - p_c^{\text{Hit}})}, \tag{5.4}$$

which is the average of insertion rate for each Data packet weighted by its arrival rate $\lambda_c(1 - p_c^{\text{Hit}})$. p_c^{Hit} is the cache hit rate of Interest packet for Data packet c. p_c^{Hit} is derived as follows.

As the author selects in Section 5.4.1, the author uses the FIFO eviction algorithm behind the Filter admission algorithm. In this case, the cache hit rate of Interest packet for Data packet c to the cache is derived by the analytical model of a FIFO eviction algorithm [64] and the insertion rate for Data packet c. In [64], the cache hit rate of Interest packet for Data packet c in a FIFO p_c^{Hit} is the probability that Data packet c is in the cache and p_c^{Hit} is expressed as $\lambda_c^{\text{Insertion}} \tau_c$, where $\lambda_c^{\text{Insertion}}$ is the frequency at which Data packet c enters the cache and τ_c is its mean cache eviction time. Since a Data packet c is inserted into the cache only when an Interest packet for Data packet c misses at the cache and then a Data packet c is inserted, $\lambda_c^{\text{Insertion}}$ is $\lambda_c(1 - p_c^{\text{Hit}}) p_c^{\text{Insertion}}$. Since τ_c can be approximated to be a constant independent of each Data packet when the cache size is large [64], the author assumes τ_c to be a constant value τ. From the above, the cache hit rate of Interest packet for Data packet c to the cache with FIFO eviction and inserted Data packet selection is expressed as

$$p_c^{\text{Hit}} = \frac{\lambda_c p_c^{\text{Insertion}} \tau}{1 + \lambda_c p_c^{\text{Insertion}} \tau}. \tag{5.5}$$

The author can obtain τ for a cache of size C by solving

$$C = \sum_{c \in G} \frac{\lambda_c p_c^{\text{Insertion}} \tau}{1 + \lambda_c p_c^{\text{Insertion}} \tau}. \tag{5.6}$$

Therefore, the average cache hit rate p^{Hit} is expressed as

$$p^{\text{Hit}} = \frac{\sum_{c \in G} \lambda_c p_c^{\text{Hit}}}{\sum_{c \in G} \lambda_c}, \tag{5.7}$$

which is the average of the cache hit rate for all Data packets weighted by arrival rates of Interest packets requesting for them.

5.3.3 Threshold Value

Determining the optimal value for the threshold θ, which realizes both high cache hit rate and low cache insertion rate, is not trivial because it depends on many factors, such as a cache size and traffic patterns. Instead of determining the optimal value, this section determines it according to the facts found in the existing studies.

The threshold θ is set to 2 according to the following two facts: one is that many objects, video objects in particular, are requested only once [65] and the other is that immediately evicting one-timer objects, which are not requested until they are evicted from the cache, contributes to high cache hit rate [66]. More precisely, Gill et al. [65] measured requests to YouTube videos at a gateway of a campus network and revealed that 68.1% of requests to YouTube videos from the campus network are observed only once. Imai et al. [66] analytically investigated that evicting such one-timer objects immediately from the cache greatly contributes to high cache hit rate. These studies imply that the threshold $\theta = 2$ prevents such one-timer Data packets from being inserted to the cache, and hence it contributes to high cache hit rate with sufficiently low cache insertion rate.

In the following subsection, the author will analytically investigate that $\theta = 2$ realizes sufficiently high cache hit rate and low cache insertion rate under a condition where requests are generated according to the Poisson process, although the cache hit rate and the cache insertion rate are not always optimal. In Section 5.4, the author will validate the above claim through simulations.

5.3.4 Analysis Results

The author analyzes how the threshold θ and the history length q affects cache hit rate and cache insertion rate of Filter. Please note that the author employs the same analysis conditions as those in Section 5.4.1, where the number of unique Data packets and the cache size are 1×10^7 Data packets and 1×10^5 Data packets, respectively, and Data packets are requested according to the Zipf distribution with the parameter $\alpha = 0.8$. Figure 5.4 shows the cache insertion rate $p^{\text{Insertion}}$ and the cache hit rate p^{Hit} when the history length q varies from 1×10^4 to 1×10^6 and the threshold θ varies from 1 to 30. The horizontal axis indicates the threshold θ. Figure 5.4(a) indicates that $p^{\text{Insertion}}$ is sufficiently low when θ is larger than 2 under all settings of q. In Figure 5.4(b), the threshold 2 obtains near optimal cache hit rate among various threshold values under all settings of q. In the rest of this chapter, for small memory consumption of Filter, the author sets $1 \times C$ to the history length q, where C is the cache size.

5.4 Performance Evaluation

In this section, the author evaluates performance of an NDN packet forwarding scheme with the proposed Filter in the following steps: First, the author proves that Filter achieves high cache hit rate and low cache insertion rate comparable to existing cache algorithms. Second, the author evaluates how NDN packet processing with Filter reduces computation time compared to that with a cache eviction algorithm. Third, the author implements a prototype of an NDN router with Filter and measure its packet forwarding speed as an overall performance.

5.4.1 Evaluation Conditions

Experiment Platform

For experiments, the author uses two computers with a Xeon E5-2699 v4 CPU (2.20 GHz × 22 CPU cores), eight DDR4 16 GB DRAM devices, and two Intel Ethernet Converged Network Adapter XL710 (dual 40 Gbps Ethernet ports) NICs. The operating system on the computers is Ubuntu 16.04 Server.

One computer is used as a router and the other is used as a consumer and a producer. The two computers are connected with four 40 Gbps direct-attached QSFP+ copper cables. The two links are used for connecting the consumer and the router and the other two are used for the router and the producer. That is, the total bandwidth between the consumer and the router and the router and the producer is 80 Gbps. The consumer sends Interest packets to the producer via the router, and the producer returns Data packets in response to the Interest packets via the router. If Interest packets hit the cache of the router, the router returns Data packets instead of the producer.

To generate packets and FIB entries for experiments, the author uses the results of the analysis on HTTP requests and responses of the IRCache traces [52] conducted in [10]. 13,548,815 FIB entries are stored at the FIB. The average number of components in prefixes of FIB entries are set to 4 so that the average number of FIB lookup operations per one Interest packet is slightly larger than that in [10]. The average number of components in an Interest and a Data packet is set to 7 and the average length of each component is set to 9 characters. Interest and Data packets conform to the NDN TLV packet specification [51], and they are encapsulated by IP packets so that sizes of an Interest packet and a Data packet are 121 bytes and 1143 bytes, respectively.

Traffic Loads

The author uses traffic loads generated on the basis of two typical popular applications in the Internet, i.e., web and video, because they account for a large portion of the current Internet traffic [67]. The author assumes that web objects are small enough so that each object consists of one Data packet. In contrast, a video is divided into multiple Data packets. The author chooses YouTube as a video application because it is one of the most famous video applications. Simulations for YouTube videos allow us to evaluate how Filter behaves in the condition that traffic loads are not created according to the Poisson process, whereas the analysis in Section 5.3 uses traffic loads according to it.

The author creates three types of traffic loads. The first workload is web traffic. The producer stores 1×10^7 unique web objects where one web object corresponds to one Data packet and the consumer generates Interest packets for these Data packets according to the Poisson process and the Zipf distribution with the parameter $\alpha = 0.8$ [67]. The second workload is video traffic. The producer stores 1×10^4 videos where the size of each video is 10 Mbytes [65] and each video consists of 10^4 Data packets with 1024 bytes payload. Assuming that a few thousands of consumers are accommodated on the router and each consumer views one video per day with 5 Mbps bit rate, consumers generate requests for the video, i.e., Interest packets for the first Data packet of the video, according to the Poisson process with the average rate 0.5 [request/sec] and Interest packets for the subsequent Data packets in the constant time interval with the constant rate 625 [packet/sec]. The third workload is the combination of web and video traffic. The author sets the ratio of the amount of web traffic to that of video traffic to 0.7 in this workload, as Fricker et al. investigated in [67].

The author evaluates the cache hit rate and the cache insertion rate at an edge router, to which consumers are directly connected, that is, the aforementioned Interest packets directly arrive without being cached between the consumers and the router. This is because cache hit rate in a realistic ISP network was already evaluated by Sun et al. [26] and they conclude that LFU is the best algorithm in terms of traffic reduction in the entire network and the second best one in terms of cache hit rate. Hence, the author focuses on validating that the performance of Filter is close to that of LFU at an edge router.

Reference Cache Algorithms

As a reference cache eviction algorithm for Filter, the author selects ARC [13] among existing cache eviction algorithms [13, 55] which exploits the history of incoming requests to Data packets because ARC realizes the fast computation with sufficiently high cache hit rate. Although frequency-based cache eviction algorithms such as LFU and WLFU [55] provide high cache hit rate, they cause heavy computational overheads with logarithmic time complexities. However, the computation complexity of ARC is $O(1)$ and its computational overhead is low comparable to that of Least Recently Used (LRU). Nevertheless, ARC provides the high cache hit rate comparable to WLFU as Gil et al. investigated in [27]. In this evaluation, the author implements the ARC eviction based on data structures and algorithms which the authors of ARC have proposed in [68]. Please note that the author uses the FIFO eviction algorithm behind the Filter admission algorithm.

5.4.2 Cache Insertion and Hit Rates

First, the author evaluates the cache insertion rates of Filter and TinyLFU and the cache hit rates of Filter and other cache algorithms through simulations.

Figure 5.5 shows the cache insertion rate $p^{\mathrm{Insertion}}$ with Filter under three workloads of web traffic, video traffic and traffic including both web traffic and video traffic. In Fig. 5.5, the cache size varies from 1% of the number of total Data packets stored in the producer to 10% of it. For comparison purposes, the author also plots the cache insertion rate with TinyLFU, which was chosen as a reference cache admission algorithm for Filter in Section 5.2.4. As Gil et al. [27] selected for the TinyLFU admission, the author selects the Random eviction, which randomly chooses a victim among Data packets stored in the cache, behind the TinyLFU admission. Note that the window size of TinyLFU is set to a sufficiently large number, $50 \times C$, where C is the cache size. The horizontal and vertical axes indicate the cache size and the cache insertion rate, respectively. Figure 5.5 indicates that the cache insertion rate with Filter is lower than that with TinyLFU under all of three workloads.

Next, the author shows that Filter achieves the sufficiently high cache hit rate even if it is used in conjunction with a simple cache eviction algorithm, such as FIFO and Random. For comparison purposes, the author also plots the cache hit rates of the FIFO eviction, the Random eviction, the LFU eviction, the ARC eviction, and the TinyLFU admission. Figure 5.6 shows simulation results of the cache hit rate under three workloads of web traffic, video traffic and traffic including both web and video traffic. Since the author obtains similar results under all of three workloads, the author explains the result under the workload of web traffic of Fig. 5.6(a). In the case where the cache size is 1×10^5, the cache hit rates of FIFO, Random, LFU, ARC, TinyLFU, and Filter are 0.238, 0.241, 0.389, 0.366, 0.376, and 0.341, respectively. Although the cache hit rate of Filter is slightly lower than those of LFU, TinyLFU, and ARC, Filter improves the cache hit rate of FIFO and Random by about 1.4 times and 1.4 times, respectively. Under this evaluation condition, where popularity distribution of Interest packets does not change over time, the LFU eviction achieves the optimal cache hit rate, as Fricker et al. claimed [67]. The results in Fig. 5.5 and Fig. 5.6 suggest that Filter efficiently increases the cache hit rate and reduces the cache insertion rate compared with simple cache eviction algorithms such as Random and FIFO.

5.4.3 CPU Cycles Spent for NDN Packet Processing

In this subsection, the author estimates the average CPU cycles of the NDN software with Filter, ARC, FIFO, Random, and TinyLFU on the computer platform. The author conducts the estimation with the workload of web traffic in the following three steps:

First, the author measures the CPU cycles of individual function blocks illustrated in Fig. 5.1 and Fig. 5.2. Table 5.1 and Table 5.2 summarize the measured CPU cycles of the blocks of the NDN software. Table 5.2 shows the CPU cycles of the blocks for cache eviction and admission, i.e., B_2, B_{12} and B_{13}. Note that the CPU cycles depend on cache eviction/admission algorithms, i.e., Filter, ARC, FIFO, Random, and TinyLFU.

Second, the author estimates the CPU cycles for various cache sizes by assigning the measured CPU cycles of the blocks and the cache hit and the insertion rates estimated with simulations, shown in Fig. 5.5 and Fig. 5.6, to Eqs. (5.1) and (5.2). Figure 5.7 shows the CPU cycles in the case of Filter, FIFO, Random, TinyLFU, and ARC when the cache size varies from 1×10^5 to 10×10^5

Table 5.1: CPU cycles spent for processing blocks of packet processing shown in Figs. 5.1 and 5.2

B_1	B_3	B_4	B_5	B_6	B_7	B_8	B_9	B_{10}	B_{11}
571	148	72	188	413	56	188	73	329	50

Table 5.2: CPU cycles spent for processing blocks of cache eviction and admission algorithms shown in Figs. 5.1 and 5.2

	B_2 (History Update)	B_{12} (Eviction)	B_{13} (Admission)
Filter	83	0	28
TinyLFU	91	0	282
ARC	173	202	0
Random	0	62	0
FIFO	0	0	0

packets. Filter reduces 16%, 11%, 15% and 9% of the average CPU cycles compared with ARC, FIFO, Random, and TinyLFU, respectively, when the cache size is 1×10^5 packets.

The author obtains the following observations from Fig. 5.7. i) Cache admission, such as Filter, is successful at reducing CPU cycles in the cases of the various cache hit rates by avoiding redundant cache insertions, as described in Section 5.4.2. ii) The average of the CPU cycles for NDN packet processing with Filter is even smaller than those with FIFO, whereas the cache hit rate of Filter is higher than that of FIFO. The light computation of block B_{13} of Filter contributes to this phenomenon.

Third, the author compares two cache admission algorithms, i.e., Filter and TinyLFU, in terms of trade-offs between cache hit rate and CPU cycles. Regarding cache hit rate, Filter just degrades the cache hit rate by 3% compared with TinyLFU when the cache size is 1×10^5 packets under the workload of web traffic, as evaluated in the previous section. This 3% degradation in the cache hit rate of Filter results in increasing 28 cycles in the average CPU cycles spent for NDN packet processing since the heavy computation of FIB Lookup must be executed in the case where a cache miss occurs. However, Filter entirely reduces 9% of the average CPU cycles for NDN packet processing because the light-weight computation of block B_{13} of Filter contributes to this reduction, as shown in Fig. 5.7. From these results, the author concludes that Filter provides the fast computation at the small sacrifice, i.e., the small cache hit rate degradation.

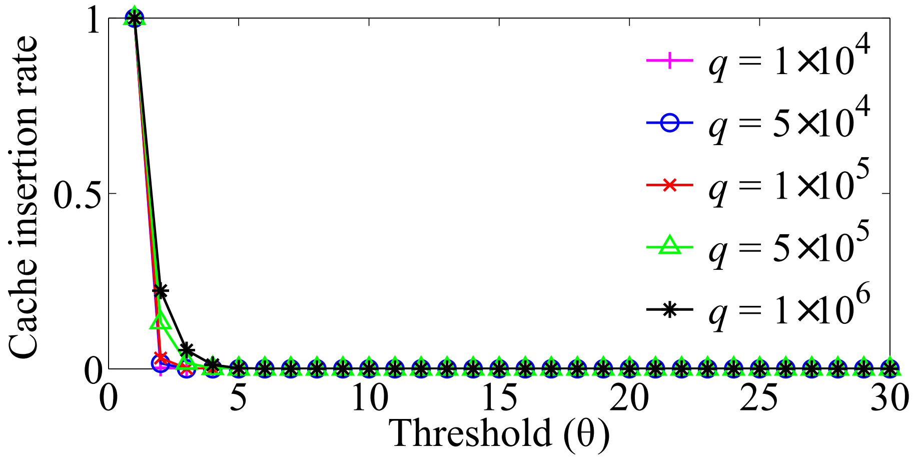
5.4.4 Forwarding Speed

Finally, the author measures the forwarding speed of NDN software routers with Filter, FIFO, and ARC. The software is executed in a single-threaded environment. The author measures the number of forwarded Interest packets per second and that of forwarded Data packets per second and calculate the sum of those numbers as the total forwarding speed. Figure 5.8 shows the measured results.

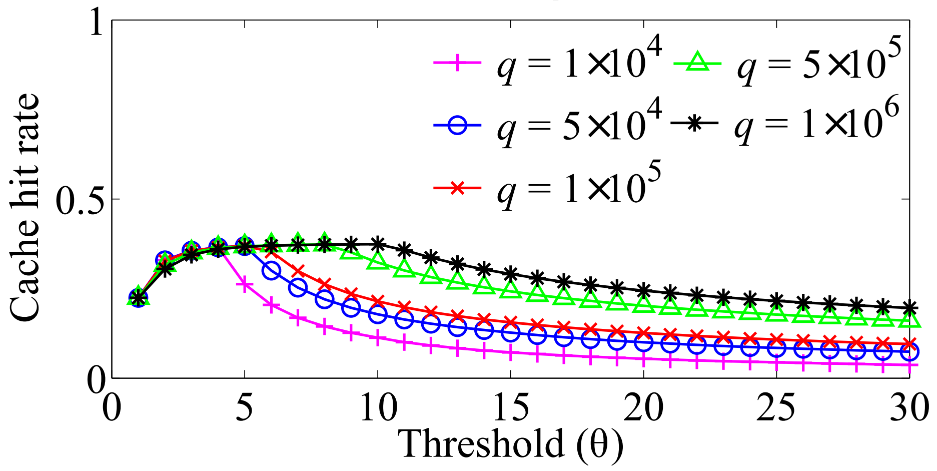
The total forwarding speed of the NDN router with Filter increases by 1.3 times and 1.1 times larger than that with ARC and that with FIFO, respectively. In the experiment conditions, where the sizes of an Interest and a Data packet are 121 bytes and 1143 bytes, the NDN router with Filter achieves the 16.1 Gbps forwarding speed in a single-threaded environment while that with ARC and that with FIFO do the 11.2 Gbps and the 13.9 Gbps forwarding speeds, respectively. These results prove that Filter achieves fast packet forwarding in NDN software routers compared to FIFO and ARC, while keeping similar cache hit rate to ARC.

5.5 Conclusion

In this chapter, the author proposes to use cache admission for achieving both high cache hit rate and fast packet forwarding speed in NDN software routers. A key of this technique is Filter, which identifies highly popular Data packets by filtering out unpopular Data packets from the cache. The author designs Filter so that it is implemented by a lightweight code of consuming a few tens of CPU cycles. A simulation-based evaluation proves that Filter achieves high cache hit rate comparable to sophisticated cache eviction algorithms such as ARC. The author implements a prototype of an NDN software router with Filter and empirically validate that the NDN router with Filter improves forwarding speed compared to that with sophisticated cache eviction like ARC and even that with simple cache eviction like FIFO.

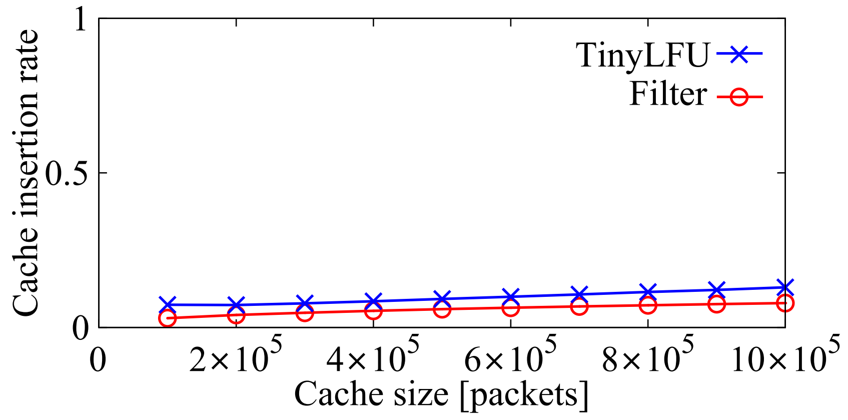


(a) Cache insertion rate ($p^{\text{Insertion}}$)

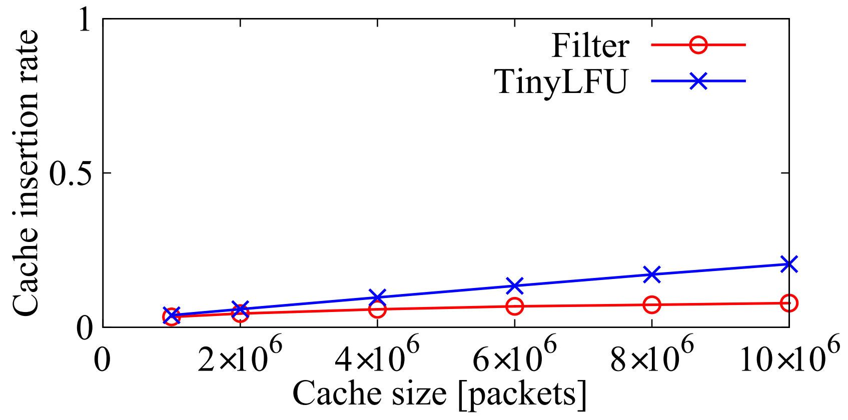


(b) Cache hit rate (p^{Hit})

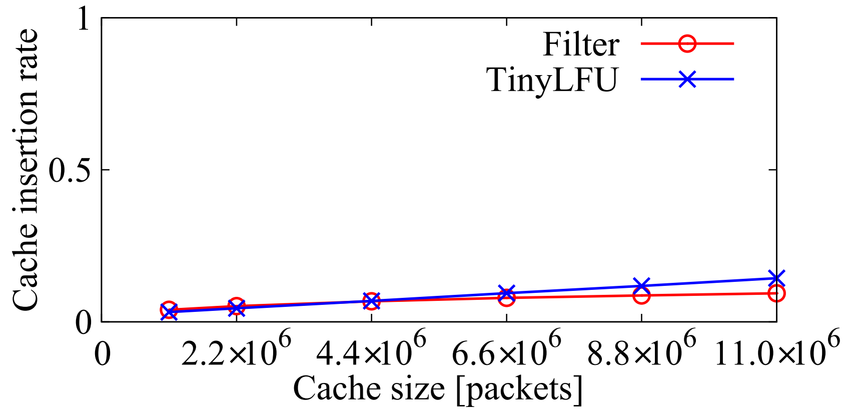
Figure 5.4: Effects of the length of the history queue q and the threshold θ on insertion rates, and hit rates



(a) Workload of web traffic

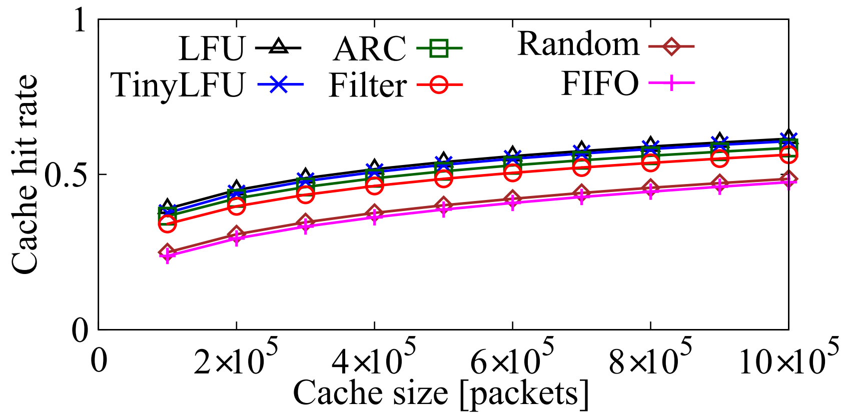


(b) Workload of video traffic

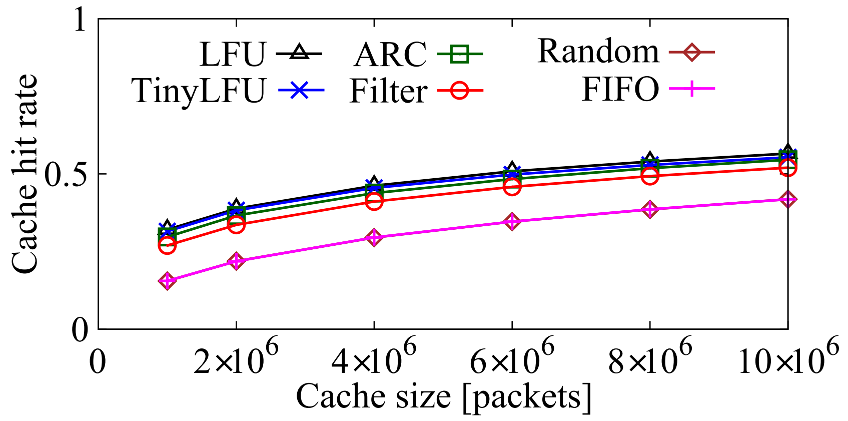


(c) Workload including web and video traffic

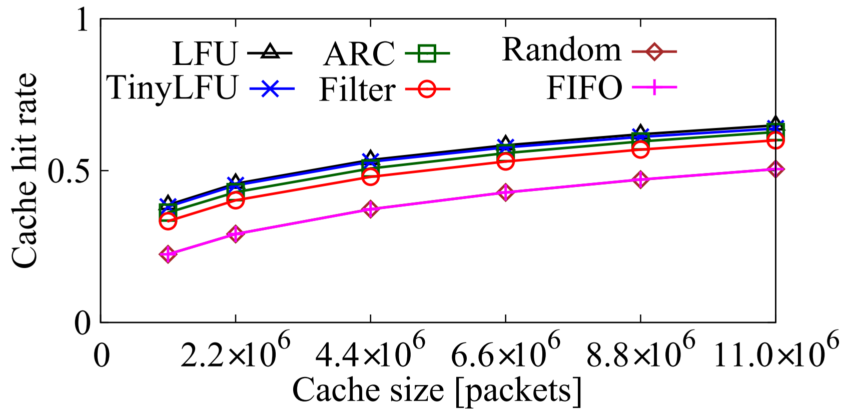
Figure 5.5: Comparisons of cache insertion rate for Filter and TinyLFU admissions



(a) Workload of web traffic



(b) Workload of video traffic



(c) Workload including web and video traffic

Figure 5.6: Comparisons of cache hit rate for FIFO eviction, Random eviction, LFU eviction, ARC eviction, Filter admission and TinyLFU admission

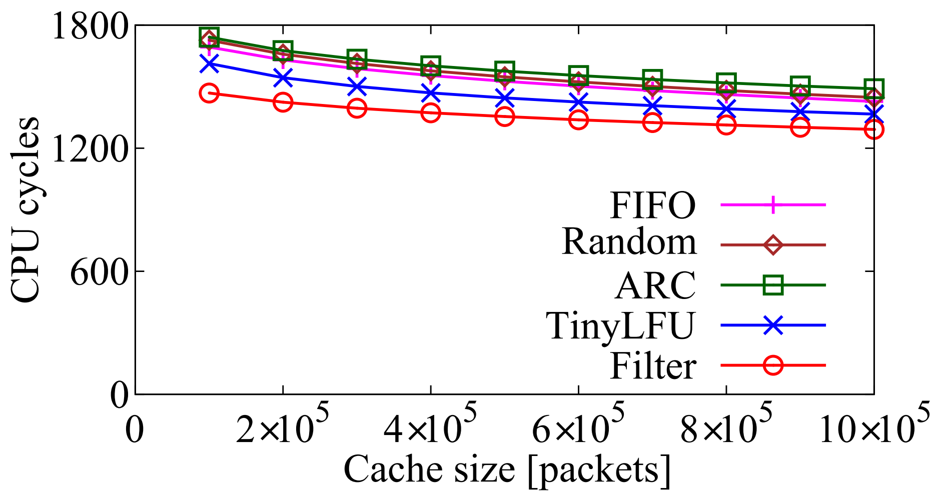


Figure 5.7: Comparison of the average CPU cycles for NDN packet processing with Filter, TinyLFU, FIFO, Random and ARC under various cache sizes

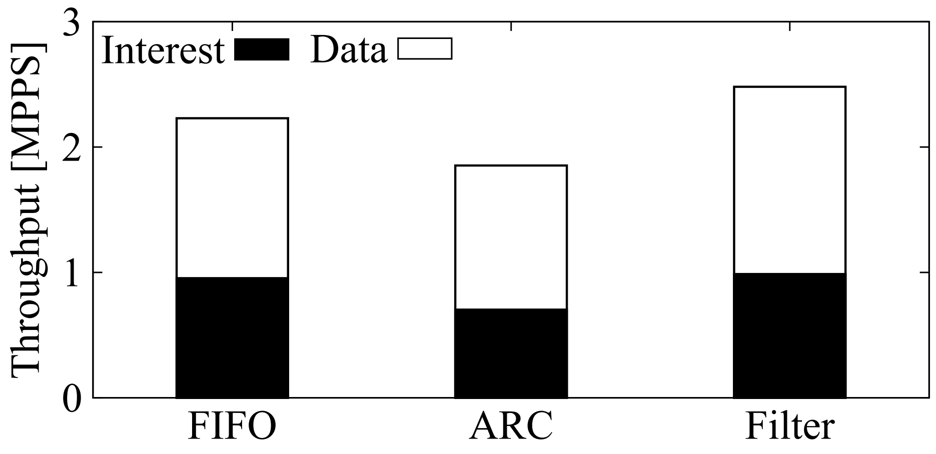


Figure 5.8: Packet forwarding speeds of NDN routers with Filter, ARC and FIFO

Chapter 6

Power Efficiency of NDN Software Routers Due to CPU Cycle Reduction

6.1 Introduction

Due to its host-based communication and end-to-end approach, IP cannot naturally provide rich functions such as mobility, multicasting and *in-network* caching. For example, locators of a host change as it moves, while the host name remains unchanged. Since IP routers do not have functions to handle such changing locators, mobility management should be implemented by special servers. In order to overcome such problems inherent to IP, NDN [7] is designed as an alternative Internet architecture. NDN naturally supports the above functions by adopting name-based routing/forwarding and ensures that NDN routers keep state of routes and caches.

Despite the fact that NDN provides numerous functions, its time-consuming name-based forwarding and caching raise a couple of issues related to power consumption [16–18] as well as forwarding performance [50]. Since name-based forwarding and caching are time-consuming, high-performance NDN router implementation has become a hot research topic [10, 18, 22] and the author also addressed high-speed NDN router implementation in Chapters 4 and 5. In this chapter, the author addresses how the proposed techniques for high speed forwarding in Chapters 4 and 5 contributes to power reduction.

Previous studies focusing on power reduction [16–18] address power reduction of memory devices, assuming that memory devices consume much power at the idle time. For example, Perino et al. assume that DRAM devices used for CSs consume 0.023 W/MB while in the idle state [18].

According to this assumption, these previous studies [16, 17] addressed to optimize cache placement, which determines where to place caches in the network [16, 17], in order to reduce the total amount of memory devices used for caching in an entire network. However, the power that memory devices consume is being reduced by lowered voltage and improvements in manufacturing technology. Vogelsang predicts that the decrease in power per bit consumed by Double-Data-Rate Synchronous DRAM (DDR SDRAM) is 1.2 per generation of DDR [41].

The author argues that this trend towards lower power consumption implies that *the power consumed by the CPU device for time-consuming name-based forwarding and caching will become larger than the power consumed by the memory devices*. This means that CPU cycle reduction of NDN packet processing contributes to power reduction of an NDN software router. According to this implication, in Chapters 4 and 5, the author addresses CPU cycle reduction of NDN packet processing and proposes the two techniques for this reduction. The first technique is the prefetch algorithm, which eliminates pipeline stalls by hiding the DRAM access latency due to accesses to forwarding tables and the other one is the cache admission algorithm, Filter, which reduces computation time by bypassing wasteful cache computations for unpopular Data packets with lightweight computation. Although forwarding speed improvement by CPU cycle reduction is evaluated, power reduction obtained by this CPU reduction need be clarified.

In this chapter, the author models how a multicore NDN software router on a computer consumes power. The author chooses a software router built on a computer as a target NDN router as Chapters 4 and 5 of this thesis show that well-engineered NDN software routers achieve high-performance name-based forwarding and caching. This model focuses on precise estimation of the power consumed by multicore CPU devices since the CPU device is a key to power efficiency of an NDN software router. Based on the model, the author empirically evaluates the power reduction effect of the above prefetch algorithms and Filter.

The main contributions of this chapter are summarized as follows:

- This chapter sheds light on the power consumed by name-based forwarding and caching, whereas most studies on power reduction in NDN consider power consumed by memory devices while in the idle state [16, 17].
- This chapter develops a general method for developing a power consumption model of a multicore NDN software router built on a computer by applying the method to the three different server computers.
- This chapter precisely models multicore CPU devices with power management states. This makes it clear that power consumption of CPU devices is proportional to their load if they

are highly loaded and thus this energy-proportionality is an important requirement to reduce power consumption of an NDN software router.

- By using the power consumption model of the well-engineered NDN router, this chapter empirically proves that prefetch algorithms and Filter reduces the total power consumption of an entire NDN software router when the traffic load of the router is high.

The rest of this chapter is organized as follows. Section 6.2 describes hardware and software architectures of the target router platform. Sections 6.3 and 6.4 empirically models how a multicore software router based on the router platform consumes power. Section 6.3 models the power consumed by the hardware platform and Section 6.4 focuses on the power consumed by NDN packet processing. Section 6.5 applies the model to estimate power consumed by an NDN software router. Section 6.6 concludes this chapter.

6.2 Reference Architecture

6.2.1 Hardware Platform

For NDN software routers with and without Filter and prefetch algorithms, the author selected multicore software routers as target hardware platforms due to the following reasons. First, full-fledged NDN routers that have all NDN functions would not be used in backbone networks but only in access networks. This is because caching in backbone networks is not as effective as it is in access networks [69]. Second, it is natural that NDN functions are not implemented by interface cards, but by service cards like the ISM (Integrated Service Module) cards in a vendor's routers. So et al. [10] showed that 20 Gbps throughput is feasible on such service cards in commercial routers, which are multicore software routers consisting of a service card and multiple interface cards. In this chapter, the author uses the three computer hardware platforms, each of which consists of one CPU device with 4 CPU cores or two CPU devices with 4 CPU cores, one DDR3 memory device, a chassis and one Network Interface Card (NIC) device. The DDR3 memory device is used to store both tables and packets.

6.2.2 Software Platform

This subsection describes software platforms with and without prefetch algorithms and Filter. As a software platform without prefetch algorithms and Filter, the author employed the software platform, designed in Section 3.1 of Chapter 3. Since its design rationale and detailed description are already provided in Section 3.1, the author briefly summarizes key features of the platform as follows: First, this software platform employs a software proposed in [10] as the NDN software. The software realizes the minimum NDN router having only functionalities of the CS, PIT and FIB. All data structures of the CS, PIT and FIB are implemented by the chained hash table described in Section 4.1. As a cache eviction algorithm, the simple FIFO cache eviction algorithm is employed. Second, in order to eliminate mutual exclusion, each thread exclusively has its own FIB, PIT, and CS and arriving packets are forwarded to threads according to their names so that accessing the tables of the other threads is prevented. Third, in order to eliminate hardware interrupts, this software platform employs the DPDK user-space driver [29], which allows user-space software to bypass the protocol stacks of operating systems and provides a way to transfer data directly from NICs to the software.

A software platform with prefetch algorithms and Filter is almost same as the above platform but differences between them are as follows: First, in the CS, Filter, which is a frequency-based cache admission algorithm designed in Chapter 5, is employed in addition to FIFO cache eviction algorithm. Second, prefetch algorithms, which prefetches data pieces to access hash tables of the CS, PIT and FIB, are employed. The algoritms are designed in Chapter 4. Both Filter and prefetch algorithms contribute to CPU cycle reduction of NDN packet processing.

Based on the above two platforms, in this chapter, the author evaluates how CPU cycle reduction due to Filter and prefetch algorithms contributes to power reduction of an NDN software router. In the next section, the author develops a power consumption model of the NDN software router.

6.3 Power Consumption Model

The author develops a power consumption model of a multicore NDN software router to satisfy the following requirements. The first requirement is that the model should reflect the loads on a hardware platform, that is, the consumed power should be a function of the loads. Such loads include a CPU load, an access rate to DRAM (DDR3) devices, and so forth. The second requirement is that the above-mentioned loads on the hardware platform should be derived from loads on NDN packet forwarding. Sections 6.3 and 6.4 address the first and second requirements, respectively.

6.3.1 Formulation

The author formulates the power consumed by a hardware platform based on a computer having the minimum configuration. The power ϕ_{router} [W] is defined as,

$$\phi_{\text{router}}(c_{\text{ndn}}, r_{\text{mem}}, \lambda_{\text{ip}}) = \phi_{\text{cpu}}(c_{\text{ndn}}) + \phi_{\text{mem}}(r_{\text{mem}}) + \phi_{\text{nic}}(\lambda_{\text{ip}}) + \Phi_{\text{chassis}}. \quad (6.1)$$

It is parameterized by the following three parameters: the load on the CPU device (c_{ndn} [cycle]), the number of bytes accessed in the DDR device per second (r_{mem} [byte/s]), and the IP packet forwarding rate (λ_{ip} [packet/s]). To measure the CPU load, the author uses the CPU cycles incurred for processing tasks, such as NDN packet processing. Each term in Eq. (6.1), which indicates the power consumed by each hardware component, is explained as follows:

- $\phi_{\text{cpu}}(c_{\text{ndn}})$ [W] is the power consumed by the CPU device. It is a function of the CPU load c_{ndn}.
- $\phi_{\text{mem}}(r_{\text{mem}})$ [W] is the power consumed by accessing the DDR3 device. It is a function of the number of bytes accessed per second r_{mem}.
- $\phi_{\text{nic}}(\lambda_{\text{ip}})$ [W] is the power consumed by the NIC. It is a function of the IP packet forwarding rate λ_{ip}.
- Φ_{chassis} [W] is the power consumed by the chassis when the router is idle. It includes the power consumption of all devices.

Note that the author uses capital and small letters to express constants and variables, respectively.

In this section, the author empirically measures the above four terms in order to model them. For the measurement, the author employs two different kinds of computer architectures: one for general-purpose computing (x86-64), and one for mission-critical computing (IA-64). From the two architectures, the author selects three server computers: a high-performance server computer for general-purpose computing (server 1), a low-energy server computer for general-purpose computing (server 2), and a high-performance server computer for mission-critical computing (server 3). Server 1 has two Intel® Xeon® E5620 processors (2.4 GHz × 4 cores), one DDR3 12 GB memory device, one Intel® X540-T2 10GBASE-T NIC. Server 2 has one Intel® Xeon® E3-1220 processor (3.10 GHz × 4 cores), one DDR3 16 GB memory device, one Intel® X540-T2 10GBASE-T NIC. Server 3 has one Intel® Itanium® 9520 processor (1.73 GHz × 4 cores), one DDR3 8 GB memory device, and one HP® Integrity Ethernet I/O adapter. To simplify the notation, the author refers to Intel® Xeon® E5620, Intel® Xeon® E3-1220, and Intel® Itanium® 9520 processors as E5620, E3-1220, and 9520 processors, respectively hereafter. As operating systems, the author uses Ubuntu 13.10 for the general-purpose servers and HP-UX 11.31 for the mission-critical server. The author uses a power meter and a current transformer developed by Omron® (ZN-CTX21 and ZN-CTS51-200As). The power is measured in joules per second (watt). Each measurement is performed for 30 minutes, consisting of a 10-minute warm-up phase and a 20-minute measurement phase. To ensure that the author is measuring the servers in their steady state, the author puts the target load on the servers without taking measurements during the warm-up phase and then measure the power consumption every minute, i.e., the author takes 20 samples during the measurement phase.

6.3.2 Power Consumed by Chassis

The author measures the power consumed by the computer under the condition where all the CPU cores are idle and the NIC is connected to an Ethernet switch but no frames are sent or received. The averages of the power consumption of servers 1, 2, and 3 are 68.43, 34.20, and 117.82 [W] and their two-sided 95% confidence intervals are [34.09, 34.30], [68.24, 68.62], and [117.62, 118.02], respectively. Since the confidence intervals are narrow enough, the author determines Φ_{chassis} for servers 1, 2, and 3 to be 34.20, 68.90, and 117.82 [W], respectively.

6.3.3 Power Consumed by CPU Devices

Overview

Modern CPU devices employ several power management states to achieve energy-efficient computing. Therefore, the author hypothesizes that CPU devices consume power in proportion to their loads if they are highly loaded. To validate the hypothesis, the author carefully models several power management states that a modern CPU device employs. The CPU power consumption model estimates the power consumed by multiple multicore CPU devices by incorporating power management states inside each CPU core and device. The author briefly describes the power management states of CPU devices and strategies for controlling the power management states. Then, the author builds the CPU power consumption model. Finally, the author validates it on the basis of empirical measurements and determine the parameters of the model for several commercial CPU devices.

CPU Power Management States

Modern CPU devices have two kinds of power management states, low-power idle states (C-states) and performance states (P-states) [70].

The C-states are used to save the power at idle time. When a CPU core is idle, it enters one of idle states, which are numbered from C1 to Cn states. When tasks arrive, it reverts to the active state, which is referred to as the C0 state. A CPU core has several idle states, e.g., C1, C3, and C6 for E5620, E3-1220, and 9520 processors, and a CPU core in a higher numbered C-state requires less power. The author models those idle states together and thus simply refer to the idle states as the C1 state, hereafter. In the case of multicore CPU devices, there are several restrictions on controlling C-states. A CPU device has a per-device C1 state, which is a power saving state for the entire CPU device, and each CPU core has a per-core C1 state. Each CPU core can enter its per-core C1 state independently of other CPU cores, whereas an entire CPU device can enter its per-device C1 state only if all its CPU cores are in their per-core C1 states.

In contrast to the C-states, the P-states are used for reducing the power consumption while the CPU device is active, i.e., the C0 state. While a CPU core is working in the C0 state, its power consumption depends on the P-states, which are pairs of the selected operating voltages and frequencies. A CPU device has several P-states numbered from P0 to Pn. For instance, an E5620 processor has seven P-states, P6 to P0, with frequencies ranging from 1.6 to 2.4 GHz. The P0 state is the highest performance state, i.e., the highest operating frequency, voltage, and power consumption. In subsequent P-states, the operating frequency and voltage are progressively reduced in tandem. Note that, in a similar way to the per-device C-states, the P-states are per-device states and thus they affect all CPU cores in a CPU device, i.e., all CPU cores work at the same frequency. The power management using the P-states is also referred to as dynamic voltage and frequency scaling (DVFS).

Strategy for Controlling CPU Power Management States

In order to build a power consumption model of multiple multicore CPU devices, the author has to model *i)* a strategy for assigning the load incurred for processing NDN packets to multiple CPU devices and cores, and *ii)* a strategy for controlling P-states according to the load on each CPU core. The author crafts those strategies so that the number of active CPU devices and cores would be minimized.

The author assumes that a computer platform has N_{cpu} identical CPU devices, each of which has N_{core} identical CPU cores. The CPU devices offer a set of P-states $\mathbf{P}$ and the operating frequency in the P-state p ($p \in \mathbf{P}$) is h_p. The author denotes the highest performance P-state (P0 state) as p_0, i.e., the highest operating frequency of the CPU device is h_{p_0}. The author uses CPU cycles per second to measure the CPU load [cycle/s]. Since a CPU core working in the P-state p can process a h_p CPU workload per second [cycle/s], the processing capacity of a CPU core is equivalent to its operating frequency.

Regarding strategy *i)*, the load for processing NDN packets, c_{ndn}, is assigned to CPU devices, so that the number of active CPU devices n_{cpu} is minimized. Since the power consumption of a CPU device increases steeply when any of CPU cores enter their per-core C0 state, which results in the transition of the entire CPU device to its per-device C0 state, minimizing n_{cpu} is a good strategy for minimizing the energy consumed by CPU devices. To minimize n_{cpu}, all CPU cores of the $n_{\text{cpu}} - 1$ CPU devices, which have N_{core} CPU cores, must work at their maximum frequency h_{p_0}. Therefore, using a ceiling function, n_{cpu} is expressed as

$$n_{\text{cpu}} = \left\lceil \frac{c_{\text{ndn}}}{N_{\text{core}} h_{p_0}} \right\rceil. \tag{6.2}$$

In this case, $n_{\text{cpu}} - 1$ CPU devices are fully utilized, i.e., all N_{core} CPU cores in the $n_{\text{cpu}} - 1$ CPU devices process NDN packets at their maximum frequency h_{p_0}. Therefore, the load processed by those fully utilized $n_{\text{cpu}} - 1$ CPU devices is $\left(n_{\text{cpu}} - 1\right) \cdot N_{\text{core}} h_{p_0}$ and the remainder of the load,

$$c_{\text{device}} = c_{\text{ndn}} - \left(n_{\text{cpu}} - 1\right) \cdot N_{\text{core}} h_{p_0}, \tag{6.3}$$

is processed by the least loaded CPU device.

The load c_{device} is assigned to CPU cores in the least loaded CPU device so that the number of active CPU cores is minimized. Since the power consumption of a CPU core is also a concave increasing function of its load, as shown in Section 6.3.3, it is preferable to assign as much load as possible to $n_{\text{core}} - 1$ CPU cores and the remainder to another one. The number of active CPU cores n_{core} in the least loaded CPU device is expressed as

$$n_{\text{core}} = \left\lceil \frac{c_{\text{device}}}{h_{p_0}} \right\rceil = \left\lceil \frac{c_{\text{ndn}} - \left(n_{\text{cpu}} - 1\right) \cdot N_{\text{core}} h_{p_0}}{h_{p_0}} \right\rceil. \tag{6.4}$$

In this case, N_{core} CPU cores in the $n_{\text{cpu}} - 1$ CPU devices and $n_{\text{core}} - 1$ CPU cores in the least loaded CPU device process NDN packets at their maximum frequency h_{p_0}. Hence, the least loaded CPU core in the least loaded CPU device processes the load c_{core} and it is calculated as

$$\begin{aligned} c_{\text{core}} &= c_{\text{device}} - (n_{\text{core}} - 1)\, h_{p_0} \\ &= c_{\text{ndn}} - \left((n_{\text{core}} - 1) + \left(n_{\text{cpu}} - 1\right) \cdot N_{\text{core}}\right) \cdot h_{p_0}. \end{aligned} \tag{6.5}$$

Next, the author defines a strategy for selecting P-states for the least loaded CPU device according to the CPU load c_{device}. The author constructs the strategy so that the operating frequency is increased if the utilization of a CPU core, which is defined as the CPU load divided by the processing capacity of the CPU core, exceeds a threshold θ. Such a strategy is often used in the current operating systems. For instance, an advanced configuration and power interface (ACPI) CPU driver employs this strategy. The P-state for the load c_{device} is modeled as,

$$p(c_{\text{device}}) = \begin{cases} p_0 & \text{if } \dfrac{c_{\text{device}}}{h_{p_0}} \geq \theta \\ \operatorname{argmin}_k h_k \text{ s.t. } \dfrac{c_{\text{device}}}{h_k} < \theta & \text{otherwise.} \end{cases} \tag{6.6}$$

For instance, the threshold θ is defined as 0.8 for the "conservative" mode in the ACPI CPU driver embedded in Linux kernels [71]. If the utilization $c_{\text{device}}/h_{p_0}$ exceeds the threshold θ, the P-state of the CPU device is set to p_0. In this case, all CPU cores work at the maximum frequency h_{p_0} since P-states are per-device states.

CPU Power Consumption Model

The power consumed by a CPU device ϕ_{cpu} [W] is determined by the load incurred for processing NDN packets c_{ndn} as

$$\phi_{\text{cpu}}(c_{\text{ndn}}) = \Phi_{n_{\text{core}},p} \cdot \left(\frac{c_{\text{core}}}{h_p}\right)^{\sigma_{n_{\text{core}},p}} + \sum_{k=1}^{n_{\text{core}}-1} \Phi_{k,p} + \Phi \cdot (n_{\text{cpu}} - 1). \tag{6.7}$$

The first and second terms on the right hand side are the power consumption of the least loaded CPU device and the third one is that of the fully utilized CPU devices. Since the power consumption of the fully utilized CPU devices, Φ, is constant, the total power consumption of the $\left(n_{\text{cpu}} - 1\right)$ fully utilized CPU devices is simply derived as $\Phi \cdot (n_{\text{cpu}} - 1)$. In contrast, the power consumption of the least loaded CPU device depends on its P-state. Since the P-states are the per-device states, the P-state of the CPU cores in the least loaded CPU device are the same and p is derived by using Eq. (6.6). The second term is the power consumed by fully utilized CPU cores in the least loaded CPU device. Empirical measurements show that the maximum power consumption of CPU cores differs slightly depending on the number of active CPU cores though the CPU cores are identical and all of them are operated at the same frequency h_p. Therefore, the author models the maximum power consumption of CPU cores as $\Phi_{n_{\text{core}},p}$, which indicates the maximum power consumption of a CPU core in the case that n_{core} CPU cores are active and their P-states are p. To derive the power consumption of the fully utilized CPU cores, the author uses their sum from $k = 1$ to $n_{\text{core}} - 1$. Note that if $n_{\text{core}} = 1$, the second term is zero, and otherwise p is always p_0 because the P-states are per-device states.

Therefore, for deriving the second term in Eq. (6.7), the author needs only $\Phi_{n_\text{core},p}$ in the case where $p = p_0$. The author describes an empirical measurement of Φ_{n_core,p_0} in the following subsection.

Finally, the author explains the first term on the right hand side of Eq. (6.7), which indicates the power consumed by the least loaded CPU core in the least loaded CPU device. This depends on its P-state p and the load c_core. Therefore, to derive this, the author has to model the C-states, i.e., the power consumption when the CPU cores transit between the C0 and C1 states depending on the load c_core, and the P-state, i.e., the power consumption of the CPU core for each P-state while the CPU core is in the C0 state. The model of this thesis is based on the power consumption model of a single core CPU device proposed in [44], which models the C-states as $((1-c/h_p)\cdot(\Phi_i^{(p)})^{1/\sigma}+c/h_p\cdot(\Phi_a^{(p)})^{1/\sigma})^\sigma$, where $\Phi_i^{(p)}$ and $\Phi_a^{(p)}$ are the power consumption of a CPU core with the P-state p in the C1 and C0 state, respectively. The constant parameter σ determines the shape of the power consumption curve. The case $\sigma = 1$ corresponds to the ideal case, where there is no overhead when entering or exiting the idle state. In reality, a latency exists when reverting from the C1 state to the C0 state and a certain amount of power is consumed during the transition. Thus, the power consumption is an increasing concave function of the load λ, i.e., the parameter σ for a current CPU device satisfies $0 < \sigma < 1$. To model the P-states, i.e., the maximum power consumption $\Phi_a^{(p)}$ in the case of the P-state p, this model uses the following equation, $\Phi_a^{(p)} = (\Phi_\text{amin}^{1/\nu} + (\Phi_\text{amax}^{1/\nu} - \Phi_\text{amin}^{1/\nu}) \cdot h_p/h_{p_0})^\nu$, where Φ_amax and Φ_amin are the maximum and minimum power consumptions in the C0 state. The constant parameter ν determines the shape of the power consumption curve. The case $\nu = 1$ corresponds to dynamic frequency scaling and $\nu = 2$ corresponds to dynamic voltage scaling. For DVFS, ν is greater than 2.

The author extends this model to multiple multicore CPU devices assuming that the CPU power management is performed as discussed in the previous subsection. The author sets $\Phi_i^{(p)}$ to 0 since the power consumption of the current CPU's idle cores is near-zero independent of other cores [72]. Then, the author applies the model [44] to the least loaded CPU core. The first term of Eq. (6.7) is based on the C-state model in [44]. According to the observations found in empirical measurements below, the author defines the parameters $\sigma_{n_\text{core},p}$ for each P-state p and the number of active CPU cores n_core instead of using a single parameter σ. The constant $\Phi_{n,p}$ is the maximum power consumption of the n-th CPU core for a given P-state p in the case that $n-1$ CPU cores are fully utilized. $\Phi_{n_\text{core},p}$ [W] can be derived as

$$\Phi_{n_\text{core},p} = \left(\Phi_\text{a min}^{\frac{1}{\nu}} + \left(\left(\Phi_{n_\text{core},p_0}\right)^{\frac{1}{\nu}} - \Phi_\text{a min}^{\frac{1}{\nu}}\right) \cdot \frac{h_p}{h_{p_0}}\right)^\nu, \tag{6.8}$$

where $\Phi_\text{a min}$ is the minimum power consumption of the CPU core in the C0 state. The constant

Table 6.1: Parameters of $\phi_{\text{cpu}}(c_{\text{ndn}})$ for E5620 Processor

n_{core}	h_p [GHz]	ν	$\Phi_{\text{a min}}$	$\sigma_{n_{\text{core}},p}$	$\Phi_{n_{\text{core}},p}$
1	1.6	$1.35 \cdot 10^3$	16.86	0.30	24.33
	1.73			0.13	25.08
	1.87			0.15	25.88
	2.0			0.065	26.66
	2.13			0.14	27.49
	2.27			0.074	28.36
	2.4			0.12	29.22
2	2.4	-	-	0.84	5.94
3	2.4	-	-	0.74	5.37
4	2.4	-	-	0.81	6.51

Table 6.2: Parameters of $\phi_{\text{cpu}}(c_{\text{ndn}})$ for E3-1220 Processor

n_{core}	h_p [GHz]	ν	$\Phi_{\text{a min}}$	$\sigma_{n_{\text{core}},p}$	$\Phi_{n_{\text{core}},p}$
1	1.6	$1.89 \cdot 10^3$	3.63	0.46	6.25
	1.8			0.40	6.69
	2.0			0.32	7.16
	2.2			0.38	7.66
	2.4			0.36	8.20
	2.6			0.40	8.78
	2.8			0.38	9.39
	3.1			0.48	10.40
2	3.1	-	-	0.59	5.71
3	3.1	-	-	0.69	6.86
4	3.1	-	-	0.56	5.24

ν is a parameter to determine the maximum power consumption of the CPU core for each P-state. Eq. (6.8) is used to derive $\Phi_{n,p}$ since P-state adaptation is applied only when the number of active CPU cores is one, the author applies Eq. (6.8) for the case of $n_{\text{core}} = 1$ and otherwise the author uses the empirically measured value for Φ_{n,p_0}.

Validation and Parameter Fitting

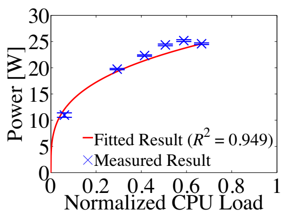
To derive the constants in Eqs. (6.7) and (6.8), the author measures the power consumption by running a program that performs an infinite loop containing only arithmetic operations and a sleep operation to avoid access to any hardware devices other than the CPU. The author varies the CPU load by changing the sleep time. The derived parameters are summarized in Tables 6.1, 6.2, and 6.3.

Table 6.3: Parameters of $\phi_{\mathrm{cpu}}(c_{\mathrm{ndn}})$ for 9520 Processor

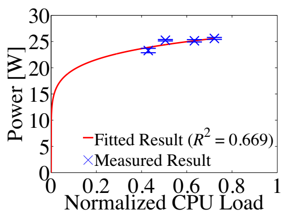
n_{core}	h_p [GHz]	ν	$\Phi_{\mathrm{a\,min}}$	$\sigma_{n_{\mathrm{core}},p}$	$\Phi_{n_{\mathrm{core}},p}$
	1.066			0.68	2.23
1	1.6	2.58	0.00	0.78	6.35
	1.733			0.34	7.80
2	1.733	-	-	0.55	3.92
3	1.733	-	-	0.74	1.39
4	1.733	-	-	0.34	1.21

First, to derive the parameters $\sigma_{n_{\mathrm{core}},p}$ and $\Phi_{n_{\mathrm{core}},p}$ in the case of $n_{\mathrm{core}} = 1$, the author measures the CPU power consumption for each P-state by changing the CPU load. The measured and estimated power consumption of the E5620 processor (server 1) for each operating frequency (P-state) is shown in Fig. 6.1. The horizontal axis shows the offered CPU load normalized to the maximum processing capacity and the vertical axis shows the power consumption. Cross markers and error bars indicate the mean of 20 measured results and 95% confidence intervals. Since the confidence intervals are considerably small for measurements, as shown in Fig. 6.1, the confidence intervals for the CPU power measurements are omitted, hereafter. The author derives $\sigma_{n_{\mathrm{core}},p}$ with least squares approximation. The significance of the regression is measured by the coefficient of determination, $R^2 = 1 - \sum_i(y_i - m_i)/\sum_i(y_i - \bar{y})$, where y_i denotes the sample value with mean $\bar{y}$ whereas m_i is the modeled value, and R^2 is shown in each graph. Due to space limitations, the author omits graphs for the E3-1220 and 9520 processors. The tendencies of the fitting results of the E3-1220 and 9520 processors are the same as that of the E5620 processor. R^2 of the fitting results for the E3-1220 processor in the case of h_p = 1.6, 1.8, 2.0, 2.2, 2.4, 2.6, 2.8, and 3.1 GHz are 0.934, 0.946, 0.856, 0.922, 0.978, 0.981, 0.945, and 0.954, respectively. Those for the 9520 processor in the case of h_p = 1.066, 1.6, and 1.73 GHz are 0.911, 0.899, and 0.919, respectively. Since R^2 of the fitting results are high, the author determines the parameters $\sigma_{n_{\mathrm{core}},p}$ and $\Phi_{n_{\mathrm{core}},p}$ in the case of $n_{\mathrm{core}} = 1$, as shown in Tables 6.1, 6.2, and 6.3.

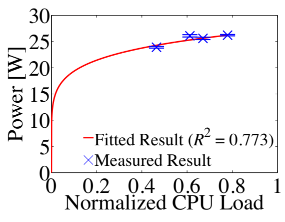
Next, to derive $\sigma_{n_{\mathrm{core}},p}$ in the case that n_{core} is more than 1, the author measures the power consumption of the n-th CPU core by changing the load on the core. During the measurement of the n-th CPU core, the other $n - 1$ CPU cores are fully utilized and the remaining CPU cores are kept idle. Since the P-state of all CPU cores is p_0 when more than one CPU core is active, the author measures the power consumption in the case of the P-state p_0. The measured and fitted results for the E5620 are shown in Fig. 6.2. The meanings of the axes and markers are the same as those in Fig. 6.1. R^2 of the fitting results for the E3-1220 processor in the case of n_{core} = 2, 3, and 4 are 0.891, 0.870,



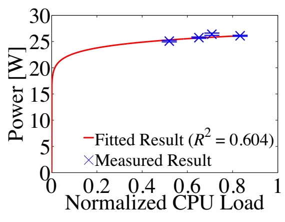
(a) 1.6 GHz



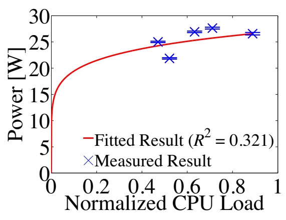
(b) 1.73 GHz



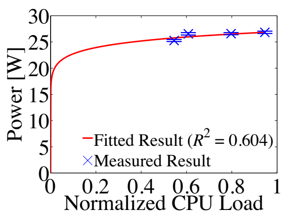
(c) 1.87 GHz



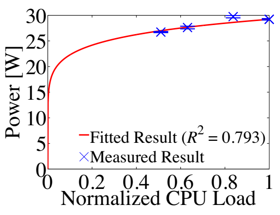
(d) 2.0 GHz



(e) 2.13 GHz

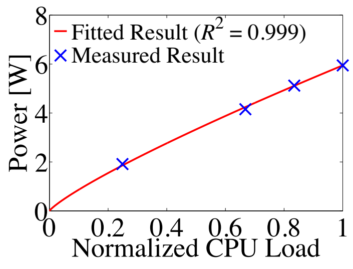


(f) 2.27 GHz

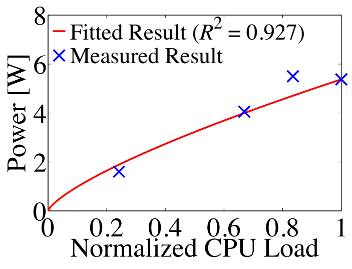


(g) 2.4 GHz

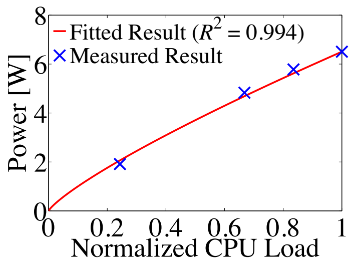
Figure 6.1: Power Consumed by CPU (E5620) Cores at Each P-state



(a) $n_{\text{core}} = 2$



(b) $n_{\text{core}} = 3$



(c) $n_{\text{core}} = 4$

Figure 6.2: Power Consumed by CPU (E5620) in the Case of $n_{\text{core}} \geq 2$

and 0.699, and those for the 9520 processor are 0.929, 0.998, and 0.913.

To derive the parameters ν and $\Phi_{\text{a min}}$, the author measures the power consumption for each P-state when a CPU core is fully utilized and the other CPU cores are kept idle. The measured power and the fitted model are shown in Fig. 6.3. The horizontal axis shows the operating frequency normalized to the maximum frequency of each CPU core. R^2 of each regression is shown in each graph. Though the error between fitted and measured power consumption of the E5620 processor with operating frequency of 2.13 and 2.27 GHz is large, R^2 is high. R^2 of the fitting results for the E3-1220 and 9520 processors are 0.951 and 0.995, respectively. Therefore, the author derives ν for the E5620, E3-1220, and 9520 processors as $1.35 \cdot 10^3$, $1.89 \cdot 10^3$, and 4.22 and $\Phi_{\text{a min}}$ for the E5620, E3-1220, and 9520 processors as 16.86, 3.63, and 0.00, respectively.

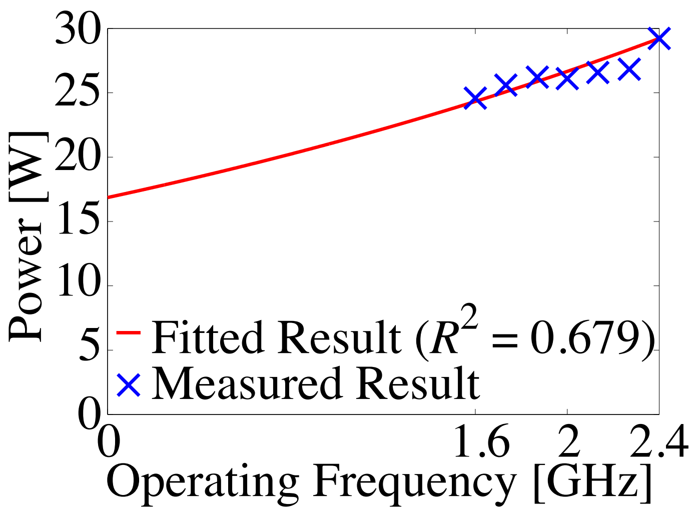
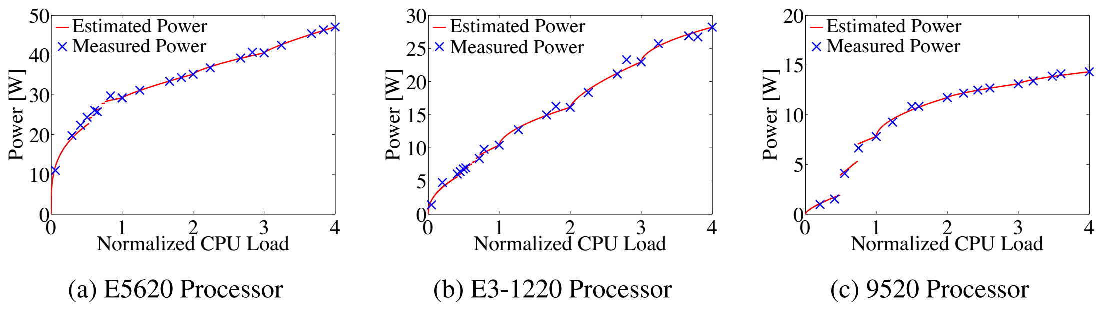


Figure 6.3: Maximum Power Consumed by one CPU Core (E5620) in Each P-state



(a) E5620 Processor (b) E3-1220 Processor (c) 9520 Processor

Figure 6.4: Power Consumption by CPU Device

Finally, the author shows the CPU power consumption of the E5620, E3-1220, and 9520 processors as estimated by the model of this thesis in Fig. 6.4. The horizontal axis shows the offered CPU load normalized to the maximum processing capacity, c_{ndn}/h_{p_0}, of each CPU core and the vertical axis shows the power consumption. The significance of the estimation has been shown as the coefficients of determination, R^2, of the fitting results in Figs. 6.1 to 6.3. Since modern CPU devices employ several power management states, they are almost energy-proportional to their loads in the case that two or more CPU cores are active. That is, the power consumption of the CPU devices are approximately modeled as a linear function to their loads. The author derives a coefficient and an intercept of the linear function by using linear regression. The power consumption of the E5620 processor is approximately modeled as $\phi_{\mathrm{cpu}}(c_{\mathrm{ndn}}) = 5.88 \cdot c_{\mathrm{ndn}}/h_{p_0} + 23.58$ and R^2 is 0.997. In the same way, the power consumption of the E3-1220 and 9520 processors are modeled as $\phi_{\text{cpu}}(c_{\text{ndn}}) = 5.85 \cdot c_{\text{ndn}}/h_{p_0} + 5.43$ with $R^2 = 0.979$ and $\phi_{\text{cpu}}(c_{\text{ndn}}) = 1.65 \cdot c_{\text{ndn}}/h_{p_0} + 8.15$ with $R^2 = 0.943$, respectively.

6.3.4 Power Consumed by Memory Device

The author formulates the power consumed by accessing data in the DDR3 device $\phi_{\text{mem}}(r_{\text{mem}})$ [W] as a function of the average number of bytes accessed per second r_{mem} [byte/s] as follows:

$$\phi_{\text{mem}}(r_{\text{mem}}) = \Psi_{\text{mem}} \cdot r_{\text{mem}}, \tag{6.9}$$

where Ψ_{mem} is the energy [joule] consumed to read/write one byte of data from/to the DDR3 device [joule/byte]. Since currently available DDR3 devices do not have DVFS, their power consumption is proportional to the access rate to the devices [73]. Hence, the author models the power consumption of the DDR3 device as a linear function to the access rate r_{mem}. Note that the power consumed constantly by the DDR3 device, such as the power for refreshing registers, is included in the power consumed by the chassis Φ_{chassis}.

To validate that the power consumed by accessing the DDR3 device is proportional to the rate of accessing it and to derive the constant Ψ_{mem} in Eq. (6.9), the author runs a program that repeatedly reads 8 byte data from the array allocated by the malloc function. In general, the DDR3 power consumption for writing and reading data is slightly different but the difference is small. For instance, writing a page to a DDR3 device operating at 1333 MHz consumes 61 nano-joule while reading does 56 nano-joule [73]. Since the author measures the power consumed for reading the DDR3 device and estimate Ψ_{mem} based on the measured results, the model of this thesis may underestimate the power consumed by the DDR3 device. However, the error is at most about 8% according to [73]. Furthermore, the DDR3 device accounts for the small proportion of the total power consumption of the servers. Therefore, the error might be much smaller. The average number of bytes accessed in the DDR3 device per second is measured by the Intel® Performance Counter Monitor. Since all three servers 1, 2, and 3 have DDR3 devices, the author uses server 2 for this measurement. The author derives the power consumed by accessing the DDR3 device by subtracting the power consumed by one active CPU core (10.40 [W]) from the measured power.

Figure 6.5 shows the power consumed by the DDR3 device at various byte access rates [byte/s]. The author derives the constants Ψ_{mem} by using the least squares approximation. Since the confidence intervals of each measured result are small and R^2 is close to 1, the author decides on Ψ_{mem} to be $0.61 \cdot 10^{-9}$ [joule/byte].

6.3.5 Power Consumed by NIC

The author formulates the power consumed by the NIC $\phi_{\text{nic}}(\lambda_{\text{ip}})$ [W] as a function of the IP packet forwarding rate λ_{ip} [packet/s] as

$$\phi_{\text{nic}}(\lambda_{\text{ip}}) = \Psi_{\text{nic}} \cdot \lambda_{\text{ip}}, \tag{6.10}$$

where Ψ_{nic} is the energy [joule] consumed by the NIC for forwarding one IP packet [joule/packet]. Though the power consumed by a currently available NIC may not be energy-proportional, its power consumption in general is much smaller than that of a CPU device [74]. Even though the author models it as a linear function to its load, the error between the estimated and measured power consumption might be small enough, which is validated by empirical measurements below.

The author measures the power consumed by the NIC at various rates in the following way. The three computers are connected by 10 Gbps Ethernet links. One computer is used as an IP router and the other two act as a client and server, respectively. The client sends UDP packets at various rates by running a simple program that switches between sending a UDP packet and sleeping. The author measures the power by choosing 1500 bytes as the size of the IP packets.

Figure 6.6 shows the power consumed by the NIC. The horizontal axis shows the packet forwarding rate [packet/s]. Cross markers and error bars indicate the mean and 95% confidence intervals of measured results. The author derives the constant Ψ_{nic} as $1.26 \cdot 10^{-5}$ [joule/packet] using the least squares approximation. The power does not appear to be exactly proportional to the IP packet forwarding rate since the power consumed by the NIC is small and thus the fluctuations in the measured values become relatively large. However, the author assumes that the power is proportional to the forwarding rate λ_{ip}. This is because the confidence intervals of each measured result are small and R^2 is close to one, and thus errors between the actual and estimated values would be negligible.

6.4 Packet Forwarding Analysis

This section describes the method of calculating values of the three parameters c_{ndn}, r_{mem} and λ_{ip} in order to derive the average power consumed by a multicore NDN software router built on computers. The author describes how average values of the three parameters are calculated from the average input Interest packet rate λ_{ndn} [packet/s], cache hit rate π_{hit} and cache insertion rate $\pi_{\text{insertion}}$ of the NDN router. Since the objective of this chapter is to compare power consumption of two routers with and without Filter and prefetch algorithms, the rest of this section provides the calculation methods for the two routers. For notation simplicity, hereinafter, routers with and without prefetch algorithms

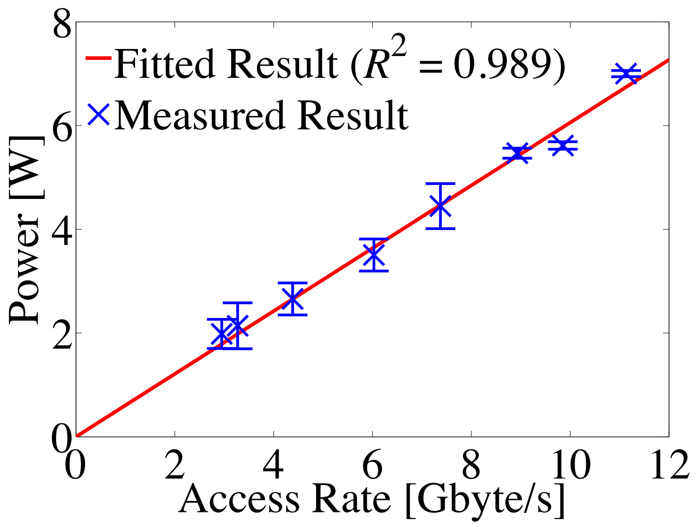


Figure 6.5: Power Consumed by DDR3 Device

and Filter are simply referred to as an optimized router and an naive router, respectively.

6.4.1 Calculation Method for Naive Router

CPU Cycles

The average power consumed by the CPU device is determined by the average number of consumed CPU cycles per second [cycles/s] c_{ndn}. c_{ndn} is calculated by the product of the average Interest packet rate λ_{ndn} [packet/s] and the average number of CPU cycles spent for processing a pair of an Interest and a Data packet [cycles/packet]. Please note that the CPU cycle number for the naive router is defined as a function of cache hit rate π_{hit} and it is denoted by $C_{\mathrm{ndn}}^{\mathrm{Naive}}(\pi_{\mathrm{hit}})$.

$C_{\mathrm{ndn}}^{\mathrm{Naive}}(\pi_{\mathrm{hit}})$ is calculated based on packet processing flows of the naive router illustrated in Fig. 6.7. Please note that the author omits a description of each block B_i in this figure since the description is already provided in 5.1 of Chapter 5. The calculation of $C_{\mathrm{ndn}}^{\mathrm{Naive}}(\pi_{\mathrm{hit}})$ is conducted with the flows in the following steps. First, the author groups the blocks so that $C_{\mathrm{ndn}}^{\mathrm{Naive}}(\pi_{\mathrm{hit}})$ is calculated with cache hit rate π_{hit}, as below:

- G_1 is the set of blocks executed always for each Interest packet: B_1, B_2, B_3 and B_{11}.
- G_2 is the block executed at a cache hit: B_4.
- G_3 is the set of blocks executed at a cache miss except for the blocks for cache eviction, i.e., B_{12} and B_{10}: B_5, B_6, B_7, B_8 and B_9.

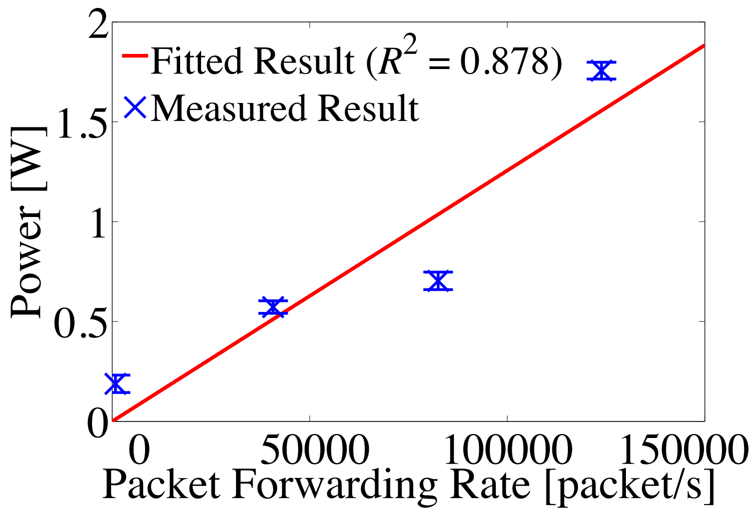


Figure 6.6: Power Consumed by NIC

Next, $C_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}})$ is defined by Eq. (6.11) as

$$C_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}}) = C_{g,1}+\pi_{\text{hit}}C_{g,2} + (1 - \pi_{\text{hit}})(C_{g,3} + C_{b,12} + C_{b,10}), \tag{6.11}$$

where $C_{b,i}$ is the CPU cycles of block B_i, and $C_{g,i}$ is the CPU cycles of group G_i.

Access Rate to DDR3 Device in Bytes

The average power consumed by the DDR3 device is determined by the average number of bytes accessed (read or written) per second r_{mem} [bytes/s]. r_{mem} is calculated by the product of the average Interest packet rate λ_{ndn} [packet/s] and the average number of bytes accessed for processing a pair of an Interest and a Data packet $M_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}})$ [bytes/packet], which is defined as a function of cache hit rate π_{hit}.

$M_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}})$ is calculated based on packet processing flows in Fig. 6.7 in the same way as that of calculating $C_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}})$. $M_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}})$ is defined by Eq. (6.12) as

$$M_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}}) = M_{g,1}+\pi_{\text{hit}}M_{g,2} + (1 - \pi_{\text{hit}})(M_{g,3} + M_{b,12} + M_{b,10}), \tag{6.12}$$

where $M_{b,i}$ is the number of bytes accessed in block B_i, and $M_{g,i}$ is the number of bytes accessed in blocks which are included in group G_i.

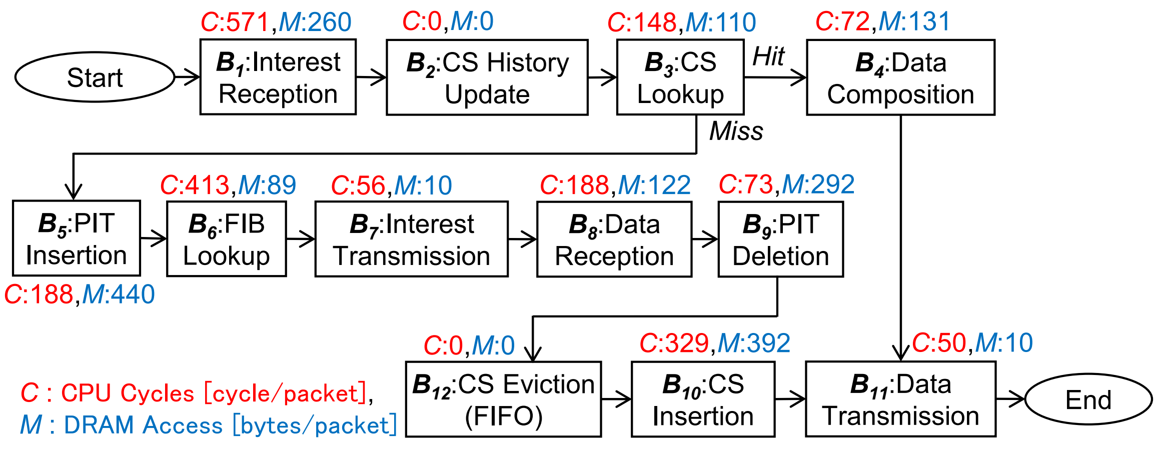


Figure 6.7: Software block diagram of naive router (without prefetch algorithms and Filter) and the consumption of CPU cycles and DRAM accesses

6.4.2 IP Packet Forwarding Rate

The average IP packet forwarding rate λ_{ip} [packet/s] is calculated by the following function of λ_{ndn} and π_{hit}:

$$\lambda_{\mathrm{ip}}(\lambda_{\mathrm{ndn}}, \pi_{\mathrm{hit}}) = \lambda_{\mathrm{ndn}} \cdot (2 - \pi_{\mathrm{hit}}). \tag{6.13}$$

λ_{ip} is calculated assuming that Interest and Data packets are encapsulated by IP packets. Since Interest and Data packets are one-for-one and their flow balances are strictly maintained in NDN [75], the author can calculate the average number of Interest and Data packets received and sent per second as follows. The router receives λ_{ndn} Interest packets from downstream routers per second. $\lambda_{\mathrm{ndn}} \cdot \pi_{\mathrm{hit}}$ Data packets are sent back to them per second when Interest packets hit those in the CS. Otherwise, $\lambda_{\mathrm{ndn}} \cdot (1 - \pi_{\mathrm{hit}})$ Interest packets are sent (forwarded) to upstream routers per second. Then $\lambda_{\mathrm{ndn}} \cdot (1 - \pi_{\mathrm{hit}})$ Data packets are received from the upstream routers and then they are sent back to the downstream routers per second. Since either an Interest or a Data packet is encapsulated by an IP packet, $\lambda_{\mathrm{ndn}} \cdot (2 - \pi_{\mathrm{hit}})$ IP packets are sent and the same number of IP packets are received per second. It means that $\lambda_{\mathrm{ndn}} \cdot (2 - \pi_{\mathrm{hit}})$ IP packets are forwarded. Thus, $\lambda_{\mathrm{ip}}(\lambda_{\mathrm{ndn}}, \pi_{\mathrm{hit}})$ is $\lambda_{\mathrm{ndn}} \cdot (2 - \pi_{\mathrm{hit}})$.

6.4.3 Calculation Method for Optimized Router

CPU Cycles

In a similar way to the calculation of the number of CPU cycles for the naive router, c_{ndn} in the case of the optimized router is calculated by the product of the average interest packet rate λ_{ndn} [packet/s]

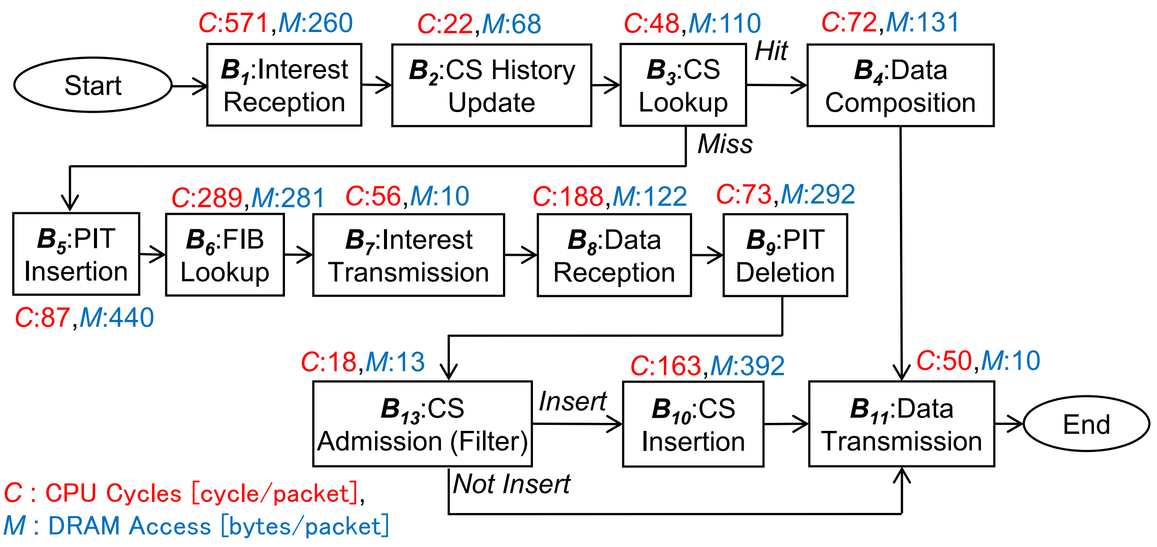


Figure 6.8: Software block diagram of optimized router (with prefetch algorithms and Filter) and the consumption of CPU cycles and DRAM accesses

and the average number of CPU cycles spent for processing a pair of an Interest and a Data packet [cycles/packet]. Please note that the CPU cycle number for the optimized router is defined as a function of cache hit rate π_{hit} and cache insertion rate $\pi_{\text{insertion}}$ and it is denoted by $C_{\text{ndn}}^{\text{Opt}}(\pi_{\text{hit}}, \pi_{\text{insertion}})$.

$C_{\text{ndn}}^{\text{Opt}}(\pi_{\text{hit}}, \pi_{\text{insertion}})$ is calculated based on the packet processing flow of the optimized router, which is illustrated in Fig. 6.8, as

$$C_{\text{ndn}}^{\text{Opt}}(\pi_{\text{hit}}, \pi_{\text{insertion}}) = C_{g,1}+\pi_{\text{hit}}C_{g,2} + (1-\pi_{\text{hit}})(C_{g,3} + C_{b,13} + \pi_{\text{insertion}}C_{b,10}). \tag{6.14}$$

Note that the author omits a description of each block B_i in this figure since the description is already provided in Section 5.1 of Chapter 5.

Access Rate to DDR3 Device in Bytes

Similarly to the calculation of access rate for the naive router, r_{mem} is calculated by the product of the average Interest packet rate λ_{ndn} [packet/s] and the average number of bytes accessed for processing a pair of an Interest and a Data packets $M_{\text{ndn}}^{\text{Naive}}(\pi_{\text{hit}}, \pi_{\text{insertion}})$ [bytes/packet], which is also defined as a function of cache hit rate π_{hit} and cache insertion rate $\pi_{\text{insertion}}$.

$M_{\text{ndn}}^{\text{Opt}}(\pi_{\text{hit}}, \pi_{\text{insertion}})$ is calculated based on packet processing flows in Fig. 6.8 in the same way as calculating $C_{\mathrm{ndn}}^{\mathrm{Opt}}(\pi_{\mathrm{hit}}, \pi_{\mathrm{insertion}})$. $M_{\mathrm{ndn}}^{\mathrm{Opt}}(\pi_{\mathrm{hit}}, \pi_{\mathrm{insertion}})$ and it is defined as

$$M_{\mathrm{ndn}}^{\mathrm{Opt}}(\pi_{\mathrm{hit}}, \pi_{\mathrm{insertion}}) = M_{g,1}+\pi_{\mathrm{hit}}M_{g,2} + (1-\pi_{\mathrm{hit}})(M_{g,3} + M_{b,13} + \pi_{\mathrm{insertion}}M_{b,10}). \tag{6.15}$$

IP Packet Forwarding Rate

In the case of the optimized router, the average IP packet forwarding rate λ_{ip} [packet/s] is calculated by Eq. (6.13) in the similar way to the case of the naive router.

6.4.4 Measured Numbers of CPU Cycles and Accesses Bytes

In this section, the author provides the measured number of CPU cycles of each block $C_{b,i}$ and that of bytes accessed in each block $M_{b,i}$ in order to calculate $C_{\mathrm{ndn}}^{\mathrm{Naive}}(\pi_{\mathrm{hit}})$, $M_{\mathrm{ndn}}^{\mathrm{Naive}}(\pi_{\mathrm{hit}})$, $C_{\mathrm{ndn}}^{\mathrm{Opt}}(\pi_{\mathrm{hit}}, \pi_{\mathrm{insertion}})$ and $M_{\mathrm{ndn}}^{\mathrm{Opt}}(\pi_{\mathrm{hit}}, \pi_{\mathrm{insertion}})$. Measurement conditions are same as those used in Chapter 5. The numbers for the naive and optimized routers are summarized in Figs. 6.7 and 6.8, respectively. In Figs. 6.7 and 6.8, the value after the character "C" and that after the character "M" represent the CPU cycles spent at each block and the number of bytes accessed in each block, respectively. Please note that in Fig. 6.8, $B_{b,2}$ and $B_{b,12}$ do not incur any CPU cycles and accessed bytes because the FIFO cache eviction does not require any processing at $B_{b,2}$ and $B_{b,12}$.

In Fig. 6.8, the number of each $M_{b,i}$ in the case of the optimized router is estimated based on the measured number of each $M_{b,i}$ in the case of the naive router, as shown in Fig. 6.7. This is because the prefetch algorithms of the optimized router make the accurate measurement of $M_{b,i}$ difficult by reordering execution orders of instructions to fetch data pieces. The author estimates $M_{b,i}$ as follows: First, the author assumes that in the case of the optimized router, extra data pieces are fetched from the DRAM device, i.e., all possible hash buckets for a given name prefix are prefetched for the FIB lookup of the name prefix regardless of whether they are used or not, based on the observation obtained in Section 4.3.2 of Chapter 4. This assumption means that the numbers of $M_{b,6}$ in the naive and optimized routers are different, whereas the other numbers of $M_{b,i}$ are same between two routers. Second, the number of prefetched hash buckets in the optimized router is equivalent to the length of the prefix according to the behavior of a FIB lookup described in Sections 4.3.2, whereas the number of fetched hash buckets in the naive router is equivalent to the matching prefix length in longest prefix matching of FIB lookup. This means that extra data pieces fetched in the optimized router are hash buckets and that their number is derived by the difference between the prefix length and the matching prefix length. The former and the latter lengths are 7 and 4, respectively, in measurement conditions of this chapter, and thus the difference is 3. Third, since the size of one hash bucket is 64 bytes, which corresponds to the size of one cache line, the number of extra bytes fetched in the optimized router is $192 = 3 \times 64$ bytes. Therefore, the number of $M_{b,6}$ in the optimized router is $281 = 89 + 192$ [bytes/packet].

In the next section, with measured numbers of CPU cycles and accessed bytes, the author compares power consumption of the optimized and naive routers in order to evaluate how CPU cycle reduction of prefetch algorithms and Filter contributes to power reduction of an NDN software router.

6.5 Case Study

This section evaluates how the prefetch algorithms and Filter designed in Chapters 4 and 5 contribute to power reduction of a NDN software router. For this evaluation, the author compares power consumption of the naive and the optimized NDN software routers. This comparison is conducted by using the power consumption model described in Section 6.3 and the calculation method of CPU cycle and accessed byte described in Section 6.4.

6.5.1 Server Platform

In the rest of this section, the author chooses a low-energy server computer with an E3-1220 processor (server 2) among the three server platforms, which are described in Section 6.2, because this server is the most power efficient among the three servers. To evaluate power consumption in the case of high traffic load, the author assumes that the server 2 has two Intel Ethernet Converged Network Adapter XL710 (dual 40 Gbps Ethernet ports) NIC devices instead of one Intel X540-T2 NIC device (single 10 Gbps Ethernet ports), which is used for experiments in Section 6.3. The author calculates the power consumption of the XL 710 NIC device, assuming that it is also proportional to the input IP packet forwarding rate in the same way as the X540-T2 NIC device. The coefficient Ψ_{nic}, which is the energy consumed by the NIC for forwarding one IP packet Ψ_{nic} [joule/packet], is estimated from the maximum packet forwarding rate [76] and the maximum power consumption [77] of the XL 710 NIC device and it is derived as $\Psi_{\mathrm{nic}} = 2.3 \times 10^{-7}$.

6.5.2 Scenario

A consumer, who sends Interest packets, and a producer, who sends back Data packets to the consumer, are connected to an NDN software router. The producer stores 1×10^{7} unique web objects, where one web object corresponds to one Data packet and the consumer generates Interest packets for these Data packets. Interest packets for each Data packet are generated according to the Poisson process and the rates of Interest packets for Data packets are determined according to the Zipf distribution with the parameter $\alpha = 0.8$ [67]. The size of one Data packet is 1024 bytes. The NDN software router can store 1×10^5 Data packets in its cache. Regarding cache algorithms, the naive router employs the FIFO cache eviction algorithm, whereas the optimized router does Filter, the cache admission algorithm designed in Chapter 5, in addition to the FIFO cache eviction algorithm. Cache hit and insertion rate of both the two routers are obtained from their simulation results in Section 5.4 of Chapter 5. The cache hit rates of the naive and the optimized routers are 0.238 and 0.341, respectively, whereas the cache insertion rates of the naive and the optimized routers are 1.0 and 0.03, respectively.

6.5.3 Power Reduction Analysis

First, the author compares the total power consumption of the naive and the optimized routers. Figure 6.9 shows the power consumption of the naive and the optimized routers. The total data retrieval rate from the producer, which is defined as the total amount of Data packet size per one second received by the consumer, is varied from 0 to about 58 Gbps. The high data retrieval rate means that the load on the router is high. The optimized router reduces the power consumption by 11.3% compared with the naive router in the case that the total data retrieval rate is 58 Gbps.

In order to analyze the power consumption in more detail, the power consumption of individual hardware devices of the NDN software router is analyzed. Figure 6.10 shows the power consumption of the individual devices of the naive and optimized routers in the case that total data retrieval rate is 58 Gbps. The following observations are found from the results in Fig. 6.10. First, the power reduction of the entire NDN software router is mostly come from the CPU device since it consumes the most power among the NIC, the CPU and the memory devices. This means that the CPU cycle reduction derived by the prefetch algorithms and Filter contributes to the reduction in the total power consumption of an NDN software router. Second, the power consumption of the NIC device and that of the memory device are relatively small compared with that of the CPU device. This implies that the extra data fetches caused by the prefetch algorithms, which increase the number of accessed bytes for the memory device, do not have much impact on the total power consumption of a NDN software router.

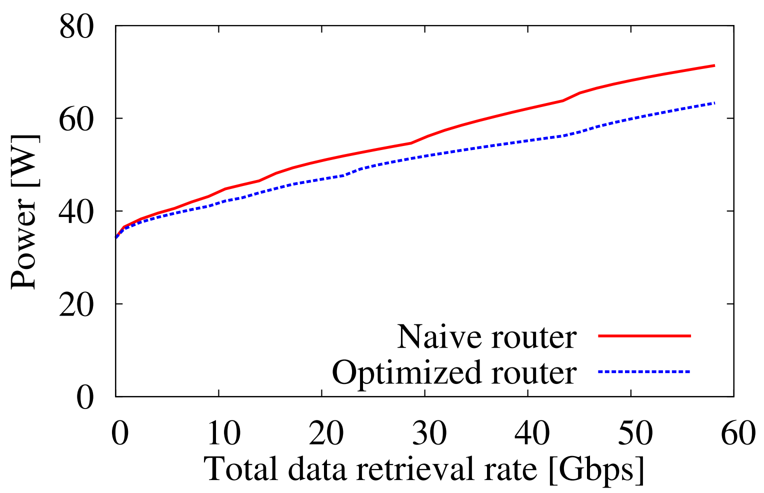


Figure 6.9: The power consumption of the naive and the optimized routers

6.5.4 Lessons Learned

This section summarizes important lessons learned from the evaluation.

An important requirement for realizing power-efficient NDN software routers is that the power consumption of devices constituting routers need to be proportional to their load. Among the devices, the CPU device is a key to reducing the power consumption because its power consumption dominates the total power consumption of an NDN software router. Hence, the prefetch algorithms and Filter are beneficial to reduce the power consumption because they reduce the number of CPU cycles for forwarding packets, thereby reducing the power consumption of the CPU device.

The power consumption model proposed in this thesis is useful for evaluating the power consumption of an NDN network because it derives the power consumption of an NDN software router by using just a few parameters. Furthermore, it is important to understand how NDN packet forwarding consumes power and to improve forwarding algorithms precisely. Our power consumption model can play an important role in this task because it can derive the power consumption of the CPU device in a function-by-function manner. For instance, the power consumption of the CPU devices caused by each function can be derived by using the CPU power consumption model (Eq. (6.7)), the calculation methods for CPU cycles (Eq. (6.11) and Eq. (6.14)), and the cache hit rate and insertion rate analysis models (Chapter 5).

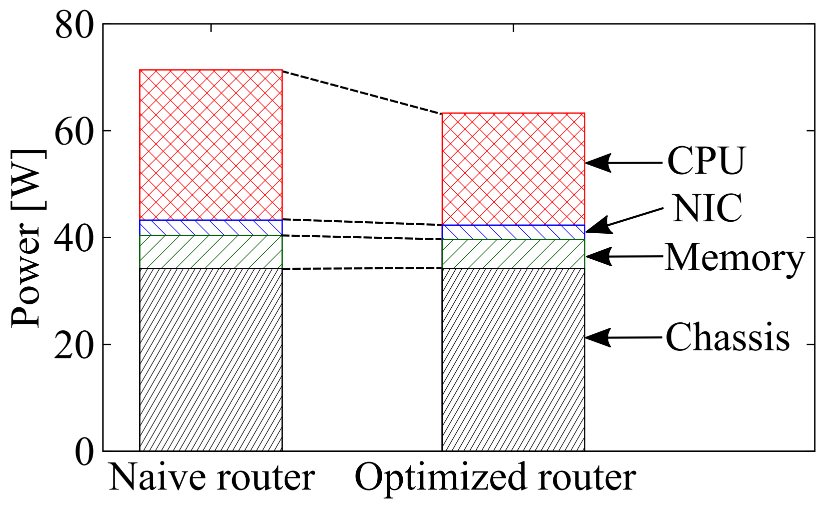


Figure 6.10: The power consumption of individual devices of the naive and the optimized routers

6.6 Conclusion

This chapter has developed a power consumption model of an NDN software router by carefully modeling how hardware devices consume the power and how software components incur the load on the hardware devices. First, the author has modeled the power consumption of a CPU, a memory, and a NIC devices as the power consumption model of hardware devices. Next, the author model the number of CPU cycles spent by CPU devices, the number of bytes accessed to DRAM devices and the packet forwarding rate of NIC devices as the load of the devices caused by the software. The author has incorporated the NDN software platform designed through Chapters 3, 4 and 5 into the power consumption model. The author learned several lessons to realize a power-efficient NDN software router through the analyses conducted with the developed power consumption model. According to the analyses, the power consumption of CPU devices dominates the total power consumption of an NDN software router and hence the CPU cycle reduction due to the prefetch algorithms and Filter, which are designed in Chapters 4 and 5, contribute to reducing the power consumption as well as improving the forwarding speed of an NDN software router, which is discussed in Chapters 4 and 5. Finally, the author believes that the developed power consumption model of an NDN software router will play an important role in developing name-based forwarding and caching algorithms.

Chapter 7

Conclusion

In this thesis, the author presents what a high-speed NDN forwarding engine on a computer is supposed to be. As the first step, the author identifies bottlenecks for high speed NDN forwarding by carefully analyzing the existing NDN forwarding engine in terms of hardware and software constraints. First of all, the author analyzed what is the most significant constraint among the constraints on the computation capability of a CPU device, bandwidths of DRAM devices and bandwidths of I/O devices on a current state-of-the art NDN software router implementation and revealed that computation on a CPU is the most significant constraint. Based on the analysis result, the author further analyzed the behavior of NDN packet processing at the level of CPU pipelines and instructions. The analysis identifies two bottlenecks. The first bottleneck is the stall to wait data to be fetched from the DRAM device and the other one is the wasteful computation of cache eviction.

To hide the DRAM access latency, the author conducted a detailed study of the existing techniques for high-speed NDN packet forwarding and compiled them into the design rationale toward the realization of an ideal NDN software router. The author compared existing data structures and algorithms both for name-based forwarding and caching from the viewpoint of the ease of hiding DRAM access latency and chose a hash table-based algorithm and a frequency-based cache admission algorithm for name-based forwarding and caching, respectively. Then, the author proposed two prefetch-friendly packet processing techniques to hide the unhidden DRAM access latency. Finally, the prototype implemented according to the rationale and the prefetch-friendly packet processing techniques demonstrated that a forwarding speed exceeding 40 million packets per second (MPPS) could be achieved on a single computer.

To resolve wasteful cache eviction computation, the author proposed to use cache admission for achieving both high cache hit rate and fast packet forwarding speed in NDN software routers.

Cache admission avoids wasteful cache computations for unpopular Data packets by filtering their unnecessary cache insertions, whereas cache eviction always inserts all of incoming Data packets into the cache. As a reference cache admission algorithm, the author designed Filter so that it is implemented by a lightweight code of consuming a few tens of CPU cycles. A simulation-based evaluation proves that Filter achieves high cache hit rate comparable to sophisticated cache eviction algorithms such as ARC. Empirical evaluations with a prototype implementation of an NDN software router with Filter validated that the NDN router with Filter improves forwarding speed compared to that with sophisticated cache eviction like ARC and even that with simple cache eviction like FIFO.

The main contributions of this thesis are summarized as follows: First, as far as the author knows, this is the first study that summarizes observations found in existing studies to build high-speed NDN routers on computers and compiles the observations into a design guideline. According to the guideline, the author reveals that a hash table-based algorithm and a frequency-based cache admission algorithm are suitable for name-based forwarding and caching, respectively. Second, the author designs a prefetch algorithm for hiding most of the DRAM access latency for both name-based forwarding and caching. The prefetch algorithm reorders instruction executions for a pair of packets so that the DRAM access latency due to data fetches is successfully hidden by execution time of instructions independent of the data fetches. Third, the author carefully designs a lightweight frequency-based cache admission algorithm, Filter, on the basis of the results of the empirical measurement of an NDN software platform. As the results of the careful design, Filter consumes a few tens of CPU cycles in average with providing as high cache hit rate as ARC, which is one of the sophisticated cache eviction algorithms. Fourth, the author implements a proof-of-concept prototype of an NDN softwarerouter with the proposed two techniques and empirically proves that forwarding speed with the two techniques increases linearly as the number of CPU cores increases, whereas that without the two techniques does not increase so. A detailed analysis at the level of CPU pipelines and instructions reveal that most DRAM access latency are successfully hidden by the prefetch algorithm and that the number of CPU cycles of NDN packet processing is reduced by the Filter admission algorithm. Fifth, the author sheds light on power efficiency of NDN software routers as a positive effect of CPU cycle reduction for NDN forwarding speed improvement. In order to validate the power reduction effect, the author, first of all, models how NDN packet processing consumes power of hardware devices on an NDN software router and then conducts a case study of comparing power consumption of naive NDN software routers and those which use the proposed techniques for high speed forwarding. The case study reveals that reduction of the number of CPU cycles due to the proposed techniques contributes to power reduction of modern computers which are energy proportional to their load.

Bibliography

[1] C. Partridge, P. P. Carvey, E. Burgess, I. Castineyra, T. Clarke, L. Graham, M. Hathaway, P. Herman, A. King, S. Kohalmi, T. Ma, J. Mcallen, T. Mendez, W. C. Milliken, R. Pettyjohn, J. Rokosz, J. Seeger, M. Sollins, S. Storch, B. Tober, G. D. Troxel, D. Waitzman, and S. Winterble, “A 50-Gb/s IP router,” *IEEE/ACM Transactions on Networking*, vol. 6, pp. 237–248, June 1998.

[2] A. Asthana, C. Delph, H. V. Jagadish, and P. Krzyzanowski, “Towards a gigabit IP router,” *Journal of High Speed Networks*, vol. 1, pp. 281–288, Oct. 1992.

[3] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy, “RouteBricks: Exploiting parallelism to scale software routers,” in *Proceedings of ACM SOSP*, pp. 15–28, Oct. 2009.

[4] S. Han, K. Jang, K. Park, and S. Moon, “PacketShader: A GPU-accelerated software router,” in *Proceedings of ACM SIGCOMM*, pp. 195–206, Aug. 2010.

[5] M. Degermark, A. Brodnik, S. Carlsson, and S. Pink, “Small forwarding tables for fast routing lookups,” in *Proceedings of ACM SIGCOMM*, pp. 3–14, Sept. 1997.

[6] “CIDR report.” http://www.cidr-report.org/as2.0/.

[7] L. Zhang, A. Afanasyev, J. Burke, V. Jacobson, kc claffy, P. Crowley, C. Papadopoulos, L. Wang, and B. Zhang, “Named data networking,” *ACM SIGCOMM Computer Communication Review*, vol. 44, pp. 66–73, July 2014.

[8] “December 2017 web server survey.” https://news.netcraft.com/archives/2017/12/26/december-2017-web-server-survey.html.

[9] H. Dai, J. Lu, Y. Wang, and B. Liu, “BFAST: Unified and scalable index for NDN forwarding architecture,” in *Proceedings of IEEE INFOCOM*, pp. 2290–2298, Apr. 2015.

[10] W. So, A. Narayanan, and D. Oran, “Named data networking on a router: Fast and DoS-resistant forwarding with hash tables,” in *Proceedings of ACM/IEEE ANCS*, pp. 215–226, Oct. 2013.

[11] T. Song, H. Yuan, P. Crowley, and B. Zhang, “Scalable name-based packet forwarding: From millions to billions,” in *Proceedings of ACM ICN*, pp. 19–28, Sept. 2015.

[12] H. Yuan and P. Crowley, “Reliably scalable name prefix lookup,” in *Proceedings of ACM/IEEE ANCS*, pp. 111–121, May 2015.

[13] N. Megiddo and D. S. Modha, “ARC: A self-tuning, low overhead replacement cache,” in *Proceedings of USENIX FAST*, pp. 115–130, Mar. 2003.

[14] U. Lee, I. Rimac, D. Kilper, and V. Hilt, “Toward energy-efficient content dissemination,” *IEEE Network*, vol. 25, pp. 14–19, Mar. 2011.

[15] U. Lee, I. Rimac, and V. Hilt, “Greening the internet with content-centric networking,” in *Proceedings of ACM e-Energy*, pp. 179–182, Apr. 2010.

[16] N. Choi, K. Guan, D. C. Kilper, and G. Atkinson, “In-network caching effect on optimal energy consumption in content-centric networking,” in *Proceedings of IEEE ICC*, pp. 2889–2894, June 2012.

[17] S. Imai, K. Leibnitz, and M. Murata, “Energy efficient data caching for content dissemination networks,” *Journal of High Speed Networks*, vol. 19, pp. 215–235, Oct. 2013.

[18] D. Perino and M. Varvello, “A reality check for content centric networking,” in *Proceedings of ACM SIGCOMM Workshop on Information-Centric Networking*, pp. 44–49, Aug. 2011.

[19] M. Fukushima, A. Tagami, and T. Hasegawa, “Efficient lookup scheme for non-aggregatable name prefixes and its evaluation,” *IEICE Transactions on Communications*, vol. E96-B, pp. 2953–2963, Dec. 2013.

[20] A. Afanasyev, J. Shi, B. Zhang, L. Zhang, I. Moiseenko, Y. Yu, W. Shang, Y. Huang, J. P. Abraham, S. DiBenedetto, C. Fan, C. Papadopoulos, D. Pesavento, G. Grassi, G. Pau, H. Zhang, T. Song, H. Yuan, H. B. Abraham, P. Crowley, S. O. Amin, V. Lehman, and L. Wang., “NFD developer’s guide,” tech. rep., NFD Team, Aug. 2014. http://named-data.net/wp-content/uploads/2014/07/NFD-developer-guide.pdf.

[21] W. So, T. Chung, H. Yuan, D. Oran, and M. Stapp, "Toward terabyte-scale caching with SSD in a named data networking router," in *Proceedings of ACM/IEEE ANCS*, pp. 241–242, Oct. 2014.

[22] R. B. Mansilha, L. Saino, M. P. Barcellos, M. Gallo, E. Leonardi, D. Perino, and D. Rossi, "Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation," in *Proceedings of ACM ICN*, pp. 59–68, Sept. 2015.

[23] I. Psaras, W. K. Chai, and G. Pavlou, "Probabilistic in-network caching for information-centric networks," in *Proceedings of ACM SIGCOMM Workshop on Information-Centric Networking*, pp. 55–60, Aug. 2012.

[24] G. Rossini and D. Rossi, "Coupling caching and forwarding: Benefits, analysis, and implementation," in *Proceedings of ACM ICN*, pp. 127–136, Sept. 2014.

[25] W. K. Chai, I. Psaras, and G. Pavlou, "Cache "less for more" in information-centric networks," in *Proceedings of IFIP TC 6*, pp. 27–40, May 2012.

[26] Y. Sun, S. K. Fayazand, Y. Guo, V. Sekar, Y. Jin, M. A. Kaafar, and S. Uhlig, "Trace-driven analysis of icn caching algorithms on video-on-demand workloads," in *Proceedings of ACM CoNEXT*, pp. 363–376, Dec. 2014.

[27] G. Einziger, R. Friedman, and B. Manes, "TinyLFU: A highly efficient cache admission policy," *ACM Transactions on Storage*, vol. 13, pp. 35:1–35:31, Dec. 2017.

[28] D. Kirchner, R. Ferdous, R. L. Cigno, L. Maccari, M. Gallo, D. Perino, and L. Saino, "Augustus: a CCN router for programmable networks," in *Proceedings of ACM ICN*, pp. 31–39, Sept. 2016.

[29] Intel Corporation, "Data plane development kit (DPDK)," 2017.

[30] The Fast Data Project (FD.io), "Vector packet processing," 2016.

[31] S. Li, H. Lim, V. W. Lee, JungHoAhn, A. Kalia, M. Kaminsky, D. G. Andersen, S. O, S. Lee, and P. Dubey, "Achieving one billion key-value requests per second on a single server," *IEEE Micro*, vol. 36, pp. 94–104, May 2016.

[32] B. Fan, D. G. Andersen, and M. Kaminsky, "Memc3: Compact and concurrent memcache with dumber caching and smarter hashing," in *Proceedings of USENIX NSDI*, pp. 371–384, Apr. 2013.

[33] B. Pfaff, J. Pettit, T. Koponen, E. Jackson, A. Zhou, J. Rajahalme, J. Gross, A. Wang, J. Stringer, P. Shelar, K. Amidon, and M. Casado, "The design and implementation of open vswitch," in *Proceedings of USENIX NSDI*, pp. 117–130, 2015.

[34] P. Graziano, "Speed up your web site with varnish," *Linux Journal*, vol. 2013, Mar. 2013.

[35] J. L. Hennessy and D. A. Patterson, "A new golden age for computer architecture: Domain-specific hardware/software co-design, enhanced security, open instruction sets, and agile chip development," in *Turing Award Lecture given at ISCA*, 2018. Available at http://iscaconf.org/isca2018/turing_lecture.html.

[36] 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 ACM SIGCOMM*, pp. 99–110, Aug. 2013.

[37] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proceedings of ACM SIGCOMM*, pp. 15–28, Aug. 2017.

[38] B. Networks, "Barefoot tofino," 2018.

[39] R. Miguel, S. Signorello, and F. M. V. Ramos, "Named data networking with programmable switches," in *IEEE ICNP*, pp. 400–405, Sept. 2018.

[40] D. Kim, Y. Zhu, C. Kim, J. Lee, and S. Seshan, "Generic external memory for switch data planes," in *Proceedings of ACM SIGCOMM Workshop on Hot Topics in Networks*, pp. 1–7, Nov. 2018.

[41] T. Vogelsang, "Understanding the energy consumption of dynamic random access memories," in *Proceedings of IEEE/ACM MICRO*, pp. 363–374, Dec. 2010.

[42] Hewlett-Packard Company, "DDR3 memory technology." http://h20000.www2.hp.com/bc/docs/support/SupportManual/c02126499/c02126499.pdf.

[43] R. Bolla, R. Bruschi, and A. Ranieri, "Performance and power consumption modeling for green COTS software router," in *Proceedings of COMSNETS*, pp. 1–8, Jan. 2009.

[44] R. Bolla, R. Bruschi, A. Carrega, F. Davoli, D. Suino, C. Vassilakis, and A. Zafeiropoulos, "Cutting the energy bills of internet service providers and telecoms through power management: An impact analysis," *Computer Networks*, vol. 56, pp. 2320–2342, July 2012.

[45] Intel Corporation, "Intel® Xeon® processor E5 v4 family," 2016.

[46] Intel Corporation, "Intel® data direct I/O technology (Intel® DDIO): A primer," 2012.

[47] Intel Corporation, "Intel® 64 and IA-32 architectures optimization reference manual," 2016.

[48] L. Saino, I. Psaras, and G. Pavlou, "Understanding sharded caching systems," in *Proceedings of IEEE INFOCOM*, pp. 1–9, Apr. 2016.

[49] A. Yasin, "A top-down method for performance analysis and counters architecture," in *Proceedings of IEEE ISPASS*, pp. 35–44, Mar. 2014.

[50] H. Yuan, T. Song, and P. Crowley, "Scalable NDN forwarding: Concepts, issues and principles," in *Proceedings of IEEE ICCCN*, pp. 1–9, Aug. 2012.

[51] The Named Data Networking (NDN) project, "NDN packet format specification," 2014.

[52] IRCache, "IRCache Project," 1995.

[53] W. M. Holt, "1.1. moore's law: A path going forward," in *Proceedings of IEEE ISSCC*, pp. 8–13, Jan. 2016.

[54] 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 IEEE INFOCOM Mini Conference*, pp. 95–99, Apr. 2013.

[55] G. Karakostas and D. Serpanos, "Exploitation of different types of locality for web caches," in *Proceedings of IEEE ISCC*, pp. 207–212, July 2002.

[56] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, pp. 3–11, Mar. 2000.

[57] S. Jiang and X. Zhang, "LIRS: An efficient low inter-reference recency set replacement policy to improve buffer cache performance," in *Proceedings of ACM SIGMETRICS*, pp. 31–42, June 2002.

[58] E. I. Organick, *The Multics System: An Examination of Its Structure*. MIT Press, 1972.

[59] S. Bansal and D. S. Modha, "CAR: Clock with adaptive replacement," in *Proceedings of USENIX FAST*, pp. 187–200, Mar. 2004.

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

[61] T. Johnson and D. Shasha, "2Q: A low overhead high performance buffer management replacement algorithm," in *Proceedings of VLDB*, pp. 439–450, Sept. 1994.

[62] Fujitsu, "White paper: Memory performance of Xeon E5-2600 v4 (Broadwell-EP) based system." http://docs.ts.fujitsu.com/dl.aspx?id=8f372445-ee63-4369-8683-da9557673357.

[63] A. Broder and M. Mitzenmacher, "Network applications of bloom filters: A survey," *Internet Mathematics*, vol. 1, no. 4, pp. 485–509, 2003.

[64] V. Martina, M. Garetto, and E. Leonardi, "A unified approach to the performance analysis of caching systems," in *Proceedings of IEEE INFOCOM*, pp. 2040–2048, Apr. 2014.

[65] P. Gill, M. Arlitt, Z. Li, and A. Mahanti, "Youtube traffic characterization: A view from the edge," in *Proceedings of ACM IMC*, pp. 15–28, Oct. 2007.

[66] S. Imai, K. Leibnitz, and M. Murata, "Statistical approximation of efficient caching mechanisms for one-timers," *IEEE Transactions on Network and Service Management*, vol. 12, pp. 595–604, Dec. 2015.

[67] C. Fricker, P. Robert, J. Roberts, and N. Sbihi, "Impact of traffic mix on caching performance in a content-centric network," in *Proceedings of IEEE NOMEN*, pp. 310–315, Mar. 2012.

[68] N. Megiddo and D. S. Modha, "One up on LRU," *;login: - The Magazine of the USENIX Association*, vol. 4, pp. 7–11, Aug. 2003.

[69] S. K. Fayazbakhsh, Y. Lin, A. Tootoonchian, A. Ghodsi, T. Koponen, B. Maggs, K. C. Ng, V. Sekar, and S. Shenker, "Less pain, most of the gain: Incrementally deployable icn," in *Proceedings of ACM SIGCOMM*, pp. 147–158, Aug. 2013.

[70] Intel Corporation, "Intel® 64 and ia-32 architectures optimization reference manual," Sept. 2014. available at https://www-ssl.intel.com/content/www/us/en/architecture-and-technology/64-ia-32-architectures-optimization-manual.html.

[71] "The Linux kernel archives." https://www.kernel.org.

[72] Intel Corporation, “Intel ® Xeon® processor 5600/5500 series platforms for embedded computing.” available at http://www.intel.com/content/dam/www/public/us/en/documents/platform-briefs/xeon-5500-5600-platform-brief.pdf.

[73] H. David, C. Fallin, E. Gorbatov, U. R. Hanebutte, and O. Mutlu, “Memory power management via dynamic voltage/frequency scaling,” in *Proceedings of the ACM ICAC*, pp. 31–40, June 2011.

[74] J. Mogul, “Improving energy efficiency for networked applications.” Keynote presentation at ACM/IEEE ANCS, Dec. 2007. available at http://www.cse.wustl.edu/ANCS/2007/slides/ANCS2007KeynoteMogul.pdf.

[75] V. Jacobson, D. K. Smetters, J. D. Thornton, M. Plass, N. Briggs, and R. L. Braynard, “Networking named content,” in *Proceedings of ACM CoNEXT*, pp. 1–12, Dec. 2009.

[76] P. Emmerich, S. Gallenmüller, D. Raumer, F. Wohlfart, and G. Carle, “MoonGen: A scriptable high-speed packet generator,” in *Proceedings of ACM IMC*, pp. 275–287, Oct. 2015.

[77] Intel Corporation, “Intel ® ethernet converged network adapter XL710.” available at https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/ethernet-xl710-brief.pdf.