Regular Paper

Lightweight Cache Admission Algorithm for Fast NDN Software Routers

Junji Takemasa[1,a) Yuki Koizumi[1,b) Toru Hasegawa[1,c)

Received: April 21, 2018, Accepted: November 7, 2018

Abstract: This paper addresses how to design a cache algorithm for achieving high cache hit rate and high packet forwarding rate of Named Data Networking (NDN) software routers. Although sophisticated cache eviction algorithms like LFU and Adaptive Replacement Cache (ARC) successfully achieve high cache hit rate, they incur heavy computational overheads to choose a victim Data packet from the cache. In contrast, cache admission is expected to achieve high cache hit rate with light-weight computation since it decides simply whether an incoming Data packet should be inserted into the cache or not. In this paper, we design a frequency-based cache admission algorithm, *Filter*, with light-weight computation by simply counting frequencies of incoming Data packets in a fixed time window. A simulation-based evaluation proves that Filter achieves high cache hit rate comparable to sophisticated cache eviction algorithms like ARC. By implementing a prototype of an NDN software router with Filter, we validate that the NDN router with Filter improves packet forwarding rate compared to that with a sophisticated cache eviction algorithm like ARC and even that with a simple one like FIFO.

Keywords: Named Data Networking, caching, router architecture

1. Introduction

Named Data Networking (NDN) [1], which naturally supports caching, mobility and multicasting by adopting name-based routing/forwarding, is promising network architecture. Despite the fact that NDN provides numerous useful functions, time-consuming caching and name-based forwarding raise an issue related to forwarding speed [2], and hence high-speed NDN router implementation has become a hot research topic. While many studies successfully improve name-based forwarding speed focusing on efficient longest prefix matching based on hash tables [3], [4], few studies focus on fast computation of caching, which is computationally heavy due to its per-packet processing.

This paper addresses cache algorithm design to achieve both high packet forwarding rate and high cache hit rate according to the following two steps, whereas most NDN router prototypes [3], [5], [6] adopt a lightweight cache eviction algorithm based on First In First Out (FIFO) at the sacrifice of cache hit rate. We choose candidates of cache algorithms among the many classes of cache algorithms in the first step and then consider how to design high-speed implementations of such algorithms in the second step.

As the first step, we choose Least Frequently Used (LFU) based cache algorithms as the candidates according to the observations in the study of Sun et al. [7]. A key motivation of this study is to answer the following two questions: The first question is whether cache algorithms of routers on a delivery path should cooperate with each other to achieve both high cache hit rate and small capacity of caches on the path or not. The second question is what kinds of cache algorithms each router should independently use if the cooperation is not necessary. Based on simulations for many traffic traces, the study concludes that sophisticated LFU-based cache algorithms executed independently on individual routers reduce total traffic in a whole network and realize sufficiently high cache hit rate compared with cooperative cache algorithms [8], [9], [10]. Reducing the total amount of traffic in a network contributes to reducing the total computation time for forwarding packets at all routers in the network, and we, hence, choose LFU-based cache algorithms, which are independently executed on individual routers, as the candidates.

As the second step, we address how to achieve high speed for LFU-based cache algorithms, of which computation is well known to be heavy. In order to address this issue, we first analyze how individual blocks of a simple FIFO cache eviction algorithm, which is much lighter than LFU-based cache algorithms, spends computation time based on our previous study [11], where we analyzed computation time of individual function blocks of the current state-of-the-art NDN router implementation [3]. This objective is to know how even a simple FIFO cache eviction algorithm spends computation time on a PC based platform.

We have identified the two obstacles by carefully analyzing CPU cycles consumed by individual function blocks. First, the packet processing flow in the case of a cache miss consumes more CPU cycles than that in the case of a cache hit. This is because a router consumes CPU cycles for receiving the Data packet, which

[1] Graduate School of Information Science and Technology, Osaka University, Suita, Osaka 565–0871, Japan
[a) j-takemasa@ist.osaka-u.ac.jp
[b) ykoizumi@ist.osaka-u.ac.jp
[c) t-hasegawa@ist.osaka-u.ac.jp

would not be needed if the Interest packet [*1] hit the cache, in addition to those for forwarding an Interest packet, which is needed regardless of a cache hit. Second, cache insertion of a Data packet is relatively time-consuming among function blocks which are executed when a cache miss occurs. It means that the insertion of unpopular Data packets which are not hit later wastes many CPU cycles.

This implies that cache algorithms for NDN routers need to increase cache hit rate and reduce unpopular Data packet insertion rate simultaneously without incurring computation overheads. We propose cache admission as a better candidate than cache eviction for the two reasons: First, cache admission obviously resolves the second 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. Sophisticated cache eviction algorithms like Least Frequently Used (LFU) and Adaptive Replacement Cache (ARC) [12] incur computational overheads for choosing sophisticatedly a victim Data packet, which will be evicted from the cache. 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 contrary, 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.

The goal of this paper is to develop a cache admission algorithm with low computation but with high cache hit rate. In our previous study [11], we have developed a lightweight cache admission algorithm, *Filter*, which consumes about only 100 CPU cycles, and have compared Filter with another cache algorithms, the TinyLFU admission algorithm [13] and the FIFO eviction algorithm, in terms of a part of computation time which is specific to cache admission or eviction. In this paper, we extend the study [11] from the following three aspects: First, we quantitatively evaluate computation time of cache admission and cache eviction algorithms by analyzing packet processing flows of an NDN router with both cache eviction and admission algorithms to understand their influences on total computation time of packet processing. Second, we prove the efficiency of Filter by comparing it with a well-known cache admission algorithm in terms of both computation time and cache hit rate to show that Filter is one of the lightest cache admission algorithms. Third, we empirically evaluate the overall packet forwarding speed of an NDN router with Filter based on its prototype implementation, whereas we have evaluated only computation time for processing packets in our previous study.

The contributions of the paper are three-fold:

- As far as we know, this is the first study which sheds light on wasteful computation of handling packets of unpopular content, whereas some studies focus on its negative effects on cache hit rate [8], [9].
- We carefully design a lightweight cache admission algorithm, *Filter* on the basis of the results of the empirical measurement of an NDN software platform. As a result of the careful design, Filter consumes a few tens of CPU cycles on average while providing as high cache hit rate as ARC.
- We implement a proof-of-concept prototype of an NDN router with Filter, which haa all functionalities including name-based forwarding and caching on a commercial off-the-shelf computer, and empirically prove that Filter improves forwarding speed of the NDN router.

The rest of this paper is organized as follows. Section 2 overviews the related work. We introduce a problem of wasteful caching computation and propose an NDN packet forwarding scheme with the caching admission based on Filter in Section 3 and Section 4, respectively. Section 5 analytically investigates cache hit rate of caching with Filter. Section 6 evaluates performance of Filter and the proposed NDN packet forwarding scheme with Filter. Finally, Section 7 concludes this paper.

2. Related Work

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 [14]. To achieve high-speed NDN packet processing, many researches attack issues of heavy NDN functions, focusing on efficient longest name prefix matching [3], [4] and efficient caching [3].

To cope with the issue of time-consuming longest prefix matching, Yuan and Crowley propose a longest prefix matching algorithm based on binary search of hash tables, which reduces the worst case computation complexity [4]. So et al. design an NDN forwarding engine with a) fast name lookup via hash tables with fast collision-resistant hash computation, b) efficient FIB lookup algorithm that provides good average and worst case FIB lookup time, and c) multi-threaded forwarding that exploits computing capabilities of multi-core CPUs [3].

In contrast to longest prefix matching, only a few studies focus on fast computation of caching. Although 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 [15], it requires heavy computation with multiple CPU cores.

In this paper, we propose to use cache admission according to the observation that handling Data packets of unpopular content is wasteful. At first, cache admission algorithms [8], [9], [10] are designed by leveraging the cooperation of caches of routers in a network so that the same Data packet is not likely to be inserted into different caches. ProbCache [8] and Betweenness [10] insert Data packets equally into caches on a delivery path, whereas Leave Copy Down (LCD) [9] focuses on off-path caching. Nevertheless, we use 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. [7] that the LFU-based cache algorithms without such cooperative cache admission achieves higher traffic reduction in

[*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.

a whole network as well as sufficiently a similar cache hit rate compared with the cache admission algorithms with cooperation. The reason why we choose these algorithms is described in Section 1. A cache admission algorithm without such cooperation is simply referred to as a cache admission algorithm, hereafter.

Precisely, we adopt a frequency-based cache admission algorithm which is executed independently from the other caches. TinyLFU [13] is a frequency-based cache admission algorithm and inserts Data packets which are received often recently into a cache. Although our proposed Filter is also a frequency-based cache admission, Filter focuses on fast computation of cache admission itself compared to the TinyLFU, as we will discuss later in Section 4.4.

3. Computation Time Analysis of Cache Algorithms

In this section, we analyze computation time of two types of cache algorithms, i.e., cache eviction and admission algorithms, to show that cache admission is better than cache eviction by analyzing the packet processing flows of the NDN software proposed in Ref. [6]. The software is chosen among several platforms [3], [6], [16] since it is highly optimized.

3.1 Named Data Networking

NDN realizes request/response communication of named content between consumers and producers. The communication is realized by two NDN packet types, Interest and Data packets. Both types of packets carry a name, which uniquely identifies a piece of content, and only a Data packet carries the piece of content. A consumer requests a piece of content by generating an Interest packet with its name and a producer responds to the Interest packet by generating a Data packet with the piece of content and its name. An intermediate NDN node, i.e., an NDN router, delivers Interest and Data packets by forwarding them to a next hop router based on their names. Interfaces of next hop routers for Interest packets and those for Data ones are stored in data structures of Forwarding Information Base (FIB) and Pending Interest Table (PIT), respectively. Received Data packets are stored in the cache called Content Store (CS). Hereafter, we use CS and cache, interchangeably.

3.2 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 both cache eviction and cache admission algorithms by carefully analyzing how individual function blocks consume CPU cycles of the flows.

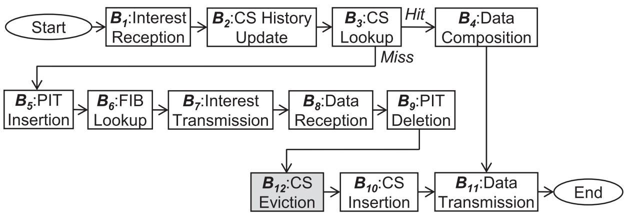


Fig. 1 Packet processing flow with cache eviction.

3.2.1 Packet Processing Flows with Cache Eviction

This section describes the request/reply NDN packet processing flows with cache eviction in the cases of both cache hits and misses according to the diagram illustrated in **Fig. 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.

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.

We group 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. (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.

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

3.2.2 Packet Processing Flows with Cache Admission

The packet processing flows with cache admission is illustrated by the diagram in **Fig. 2**. The flow in the case of a cash hit is the same as the flow with cache eviction illustrated in Fig. 1. Precisely, the CPU cycles of B_2 for the cache eviction and admission algorithms 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

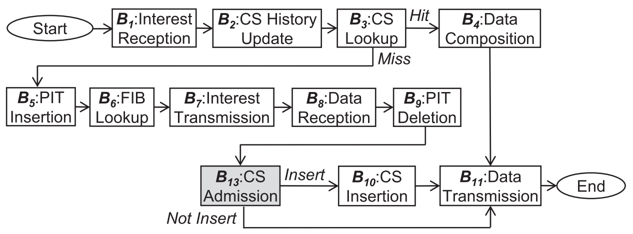


Fig. 2 Packet processing flow with cache admission.

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 $C^{\text{Avg}}_{\text{Adm}}$ are defined by Eq. (2), where $p^{\text{Insertion}}$ is the cache insertion rate. $p^{\text{Insertion}}$ is defined as the probability that Data packets are inserted into the cache by block B_{13}.

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

3.3 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., $C^{\text{Avg}}_{\text{Evi}} - C^{\text{Avg}}_{\text{Adm}}$, is $(1 - p^{\text{Hit}})((C_{b,12} - C_{b,13}) + p^{\text{Insertion}} C_{b,10})$. If the difference is positive, the average CPU cycles $C^{\text{Avg}}_{\text{Adm}}$ is smaller than $C^{\text{Avg}}_{\text{Evi}}$, which means that cache admission is faster than cache eviction.

To validate the above claim preliminarily, we estimate 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 6. Precisely, we do 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 $p^{\text{Insertion}}$ of TinyLFU is 10%. The measurement results in Section 6 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 $C^{\text{Avg}}_{\text{Evi}}$ of ARC and those of cache admission $C^{\text{Avg}}_{\text{Adm}}$ of TinyLFU are 1,817 and 1,705 CPU cycles, respectively, and $C^{\text{Avg}}_{\text{Adm}}$ is 112 CPU cycles smaller than $C^{\text{Avg}}_{\text{Evi}}$.

We have 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 paper, we focus on a lighter cache admission algorithm than TinyLFU, whereas it realizes the similar cache hit rate to TinyLFU.

4. Design of Cache Admission Algorithm

4.1 Overview

The goal of the cache admission algorithm is to identify unpopular Data packets for filtering them out in order to increase the cache hit rate and reduce the cache insertion rate. Obviously, an increase in cache hit rate contributes to improving forwarding speed of NDN routers. A few cache admission algorithms have been proposed [8], [9], [13]; however, computation time for filtering unpopular Data packets out is not carefully considered. We design a simpler cache admission algorithm, named *Filter*. In the rest of this section, we firstly describe the design rationale behind Filter, and we then design Filter according to the design rationale. Finally, we compare Filter with a well-known cache admission algorithm, TinyLFU [13], in terms of cache hit rate and computation time.

4.2 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. Our previous study [6], for instance, empirically revealed that one DRAM access spends about 11% of the average computation time of entire NDN packet processing.

We therefore do not adopt TinyLFU [13] because it is difficult to eliminate DRAM accesses from the 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, we will develop an analytical model of Filter in Section 5 to clarify a guideline of choosing the threshold.

4.3 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, we design 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. We refer to the FIFO queue and the hash table as *history queue* and *counter hash*

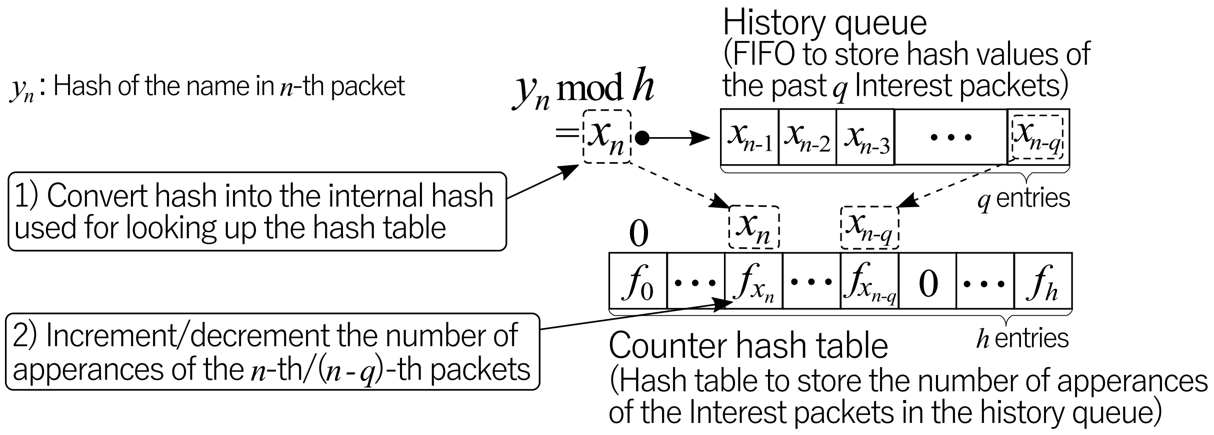

y_n: Hash of the name in n-th packet
$y_n \bmod h = x_n$
History queue
(FIFO to store hash values of the past q Interest packets)
x_{n-1} x_{n-2} x_{n-3} ... x_{n-q}
q entries
1) Convert hash into the internal hash used for looking up the hash table
0 x_n x_{n-q}
f_0 ... f_{x_n} ... $f_{x_{n-q}}$ 0 ...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

Fig. 3 Schematic of data structures of Filter.

table, respectively. The schematic of Filter is shown in **Fig. 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 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 Fig. 2), Filter computes the hash of its content name, x_n, and updates the history queue and the counter hash table, as shown in Fig. 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 Fig. 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 [17]. 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, we omit 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, we conclude that p^{FP} is sufficiently low.

4.4 Comparison with TinyLFU

Finally, we discuss the advantage and the disadvantage of our proposed Filter for existing cache admission algorithms focusing on cache hit rate and computation time. As a reference for Filter, we choose TinyLFU [13] among several cache admission algorithms [8], [9], [13] because its implementation is highly optimized, whereas other algorithms [8], [9] have focused on only high cache hit rate.

The key advantage of Filter is that its computation time is shorter than that of TinyLFU. We implement 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 Fig. 2. Please note that the measurement conditions are the same as those summarized in Section 6.1. 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 4.2.

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, we select the threshold value for Filter based on its characteristic analysis, and then in Section 6, we evaluate cache hit rates of Filter with the selected threshold and TinyLFU.

5. Characteristic Analysis of Filter

5.1 Overview

In this section, we analyze how parameters of Filter, i.e., the threshold and the history length affect the performance of Filter. We develop an analytical model of Filter, focusing on two performance metrics: *cache hit rate* and *cache insertion rate*. Cache hit rate, which is defined as the 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, we refer to cache hit rate and cache insertion rate as *hit rate* and *insertion rate*, respectively.

5.2 Analysis Model

Firstly, we compute 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, we assume 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, we define 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

$$\begin{aligned} p_c^{\text{Insertion}} &= \mathbb{P}[X_c \geq \theta - 1] = 1 - \mathbb{P}[X_c \leq \theta - 2] \\ &= 1 - \left(\exp\left(-q\lambda'_c\right) \sum_{k=0}^{\theta-2} \frac{(q\lambda'_c)^k}{k!} \right). \end{aligned} \tag{3}$$

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

$$p^{\text{Insertion}} = \frac{\sum_{c\in G} \lambda_c (1 - p_c^{\text{Hit}}) p_c^{\text{Insertion}}}{\sum_{c\in G} \lambda_c (1 - p_c^{\text{Hit}})}, \tag{4}$$

which is the average of insertion rates 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 we select in Section 6.1, we use 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 [18] and the insertion rate for Data packet c. In Ref. [18], 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 [18], we assume 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

$$p_c^{\text{Hit}} = \frac{\lambda_c p_c^{\text{Insertion}} \tau}{1 + \lambda_c p_c^{\text{Insertion}} \tau}. \tag{5}$$

We can obtain τ for a cache of size C by solving

$$C = \sum_{c\in G} \frac{\lambda_c p_c^{\text{Insertion}} \tau}{1 + \lambda_c p_c^{\text{Insertion}} \tau}. \tag{6}$$

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

$$p^{\text{Hit}} = \frac{\sum_{c\in G} \lambda_c p_c^{\text{Hit}}}{\sum_{c\in G} \lambda_c}, \tag{7}$$

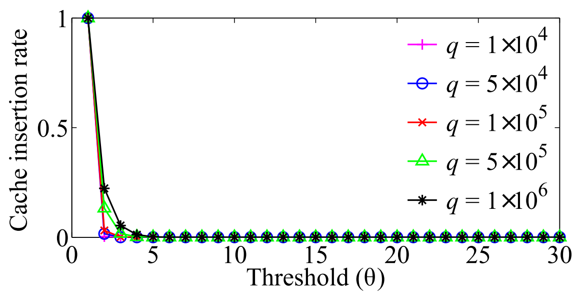
which is the average of the cache hit rate for all Data packets weighted by arrival rates of Interest packets requesting for them.

5.3 Threshold Value

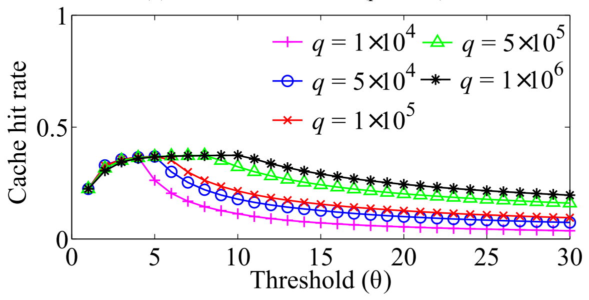
Determining the optimal value for the threshold θ, which realizes both high cache hit rate and low cache insertion rate, is not trivial because it depends on many factors, such as a cache size and traffic patterns. Instead of determining the optimal value, this section determines it according to the 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 [19] 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 [20]. More precisely, Gill et al. [19] 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. [20] 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, we 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 6, we will validate the above claim through simulations.



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



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

Fig. 4 Effects of the length of the history queue q and the threshold θ on insertion rates, and hit rates.

5.4 Analysis Results

We analyze how the threshold θ and the history length q affects cache hit rate and cache insertion rate of Filter. Please note that we employ the same analysis conditions as those in Section 6.1, 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 4** shows the cache insertion rate $p^{\mathrm{Insertion}}$ 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 4 (a) indicates that $p^{\mathrm{Insertion}}$ is sufficiently low when θ is larger than 2 under all settings of q. In Fig. 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 paper, for small memory consumption of Filter, we set $1 \times C$ to the history length q, where C is the cache size.

6. Performance Evaluation

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

6.1 Evaluation Conditions

6.1.1 Experiment Platform

For experiments, we use 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 our 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, we use the results of the analysis on HTTP requests and responses of the IRCache traces [21] conducted in Ref. [3]. 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 Ref. [3]. 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 [22], and they are encapsulated by IP packets so that sizes of an Interest packet and a Data packet are 121 bytes and 1,143 bytes, respectively.

6.1.2 Traffic Loads

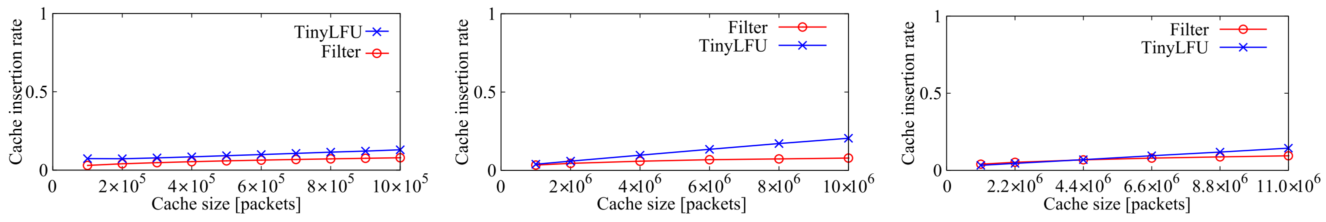
We use traffic loads generated on the basis of two typical popular applications in the Internet, i.e., web and video, because they account for a large portion of the current Internet traffic [23]. We assume that web objects are small enough so that each object consists of one Data packet. In contrast, a video is divided into multiple Data packets. We choose 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 uses traffic loads according to it.

We create 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 $\alpha = 0.8$ [23]. The second workload is video traffic. The producer stores 1×10^4 videos where the size of each video is 10 Mbytes [19] and each video consists of 10^4 Data packets with 1,024 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. We set 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 Ref. [23].

We evaluate 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. [7] 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, we focus on validating that the performance of Filter is close to that of LFU at an edge router.

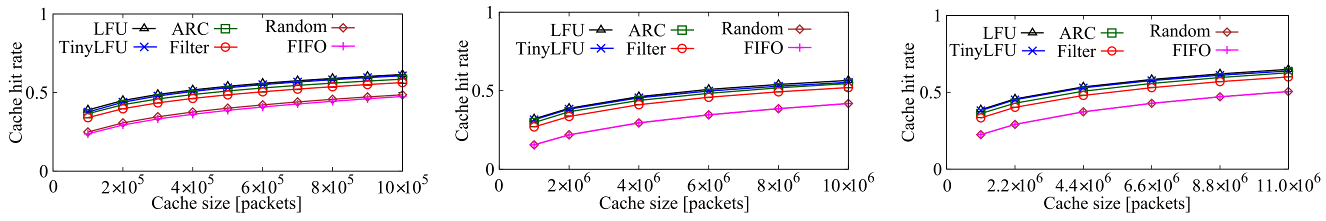
6.1.3 Reference Cache Algorithms

As a reference cache eviction algorithm for Filter, we select ARC [12] among existing cache eviction algorithms [12], [24] 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 [24] 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 Einziger et al. inves-



(a) Workload of web traffic (b) Workload of video traffic (c) Workload including web and video traffic

Fig. 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

Fig. 6 Comparisons of cache hit rate for FIFO eviction, Random eviction, LFU eviction, ARC eviction, Filter admission and TinyLFU admission.

tigated in Ref. [13]. In this evaluation, we implement the ARC eviction based on data structures and algorithms which the authors of ARC have proposed in Ref. [25]. Please note that we use the FIFO eviction algorithm behind the Filter admission algorithm.

6.2 Cache Insertion and Hit Rates

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

Figure 5 shows the cache insertion rate $p^{\text{Insertion}}$ with Filter under three workloads of web traffic, video traffic and traffic including both web traffic and video traffic. In Fig. 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, we also plot the cache insertion rate with TinyLFU, which was chosen as a reference cache admission algorithm for Filter in Section 4.4. As Einziger et al. [13] selected for the TinyLFU admission, we select 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 indicates that the cache insertion rate with Filter is lower than that with TinyLFU under all of the three workloads.

Next, we show 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, we also plot the cache hit rates of the FIFO eviction, the Random eviction, the LFU eviction, the ARC eviction, and the TinyLFU admission. **Figure 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 we obtain similar results under all of three workloads, we explain the result under the workload of web traffic of Fig. 6 (a). In the case where the cache size is 1×10^5, the cache hit rates of FIFO, Random,

Table 1 CPU cycles spent for processing blocks of packet processing shown in Figs. 1 and 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 2 CPU cycles spent for processing blocks of cache eviction and admission algorithms shown in Figs. 1 and 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

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 [23]. The results in Fig. 5 and Fig. 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.

6.3 CPU Cycles Spent for NDN Packet Processing

In this subsection, we estimate the average CPU cycles of the NDN software with Filter, ARC, FIFO, Random, and TinyLFU on the PC platform. We conduct the estimation with the workload of web traffic in the following three steps:

First, we measure the CPU cycles of individual function blocks illustrated in Fig. 1 and Fig. 2. **Table 1** and **Table 2** summarize the measured CPU cycles of the blocks of the NDN software. Table 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, we estimate the CPU cycles for various cache sizes by

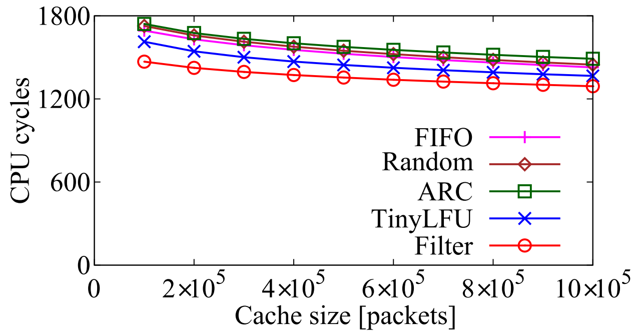


Fig. 7 Comparison of the average CPU cycles for NDN packet processing with Filter, TinyLFU, FIFO, Random and ARC under various cache sizes.

assigning the measured CPU cycles of the blocks and the cache hit and the insertion rates estimated with simulations, shown in Fig. 5 and Fig. 6, to Eqs. (1) and (2). **Figure 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 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.

We obtain the following observations from Fig. 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 6.2. 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, we compare 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. 7. From these results, we conclude that Filter provides fast computation in return for a small sacrifice, i.e., the small cache hit rate degradation.

6.4 Forwarding Speed

Finally, we measure the forwarding speed of NDN software routers with Filter, FIFO, and ARC. The software is executed in a single-threaded environment. We measure 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 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

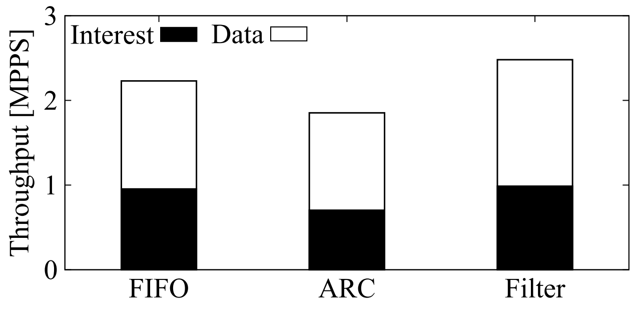


Fig. 8 Packet forwarding speeds of NDN routers with Filter, ARC and FIFO.

1,143 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 a similar cache hit rate to ARC.

7. Conclusion

In this paper, we propose 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. We design 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. We implement 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.

Acknowledgments This work has been supported by KAKENHI Grant Number JP17H01733 and Grant-in-Aid for JSPS Fellows Grant Number 17J07276.

References

[1] Zhang, L., Afanasyev, A., Burke, J., Jacobson, V., Claffy, K., Crowley, P., Papadopoulos, C., Wang, L. and Zhang, B.: Named Data Networking, *ACM SIGCOMM Computer Communication Review*, Vol.44, No.3, pp.66–73 (2014).
[2] Yuan, H., Song, T. and Crowley, P.: Scalable NDN Forwarding: Concepts, Issues and Principles, *Proc. IEEE ICCCN*, pp.1–9 (2012).
[3] So, W., Narayanan, A. and Oran, D.: Named Data Networking on a Router: Fast and DoS-resistant Forwarding with Hash Tables, *Proc. ACM/IEEE ANCS*, pp.215–226 (2013).
[4] Yuan, H. and Crowley, P.: Reliably scalable name prefix lookup, *Proc. ACM/IEEE ANCS*, pp.111–121 (2015).
[5] Kirchner, D., Ferdous, R., Cigno, R.L., Maccari, L., Gallo, M., Perino, D. and Saino, L.: Augustus: A CCN router for programmable networks, *Proc. ACM ICN*, pp.31–39 (2016).
[6] Takemasa, J., Koizumi, Y. and Hasegawa, T.: Toward an Ideal NDN Router on a Commercial Off-the-shelf Computer, *Proc. ACM ICN*, pp.43–53 (2017).
[7] Sun, Y., Fayazand, S.K., Guo, Y., Sekar, V., Jin, Y., Kaafar, M.A. and Uhlig, S.: Trace-Driven Analysis of ICN Caching Algorithms on Video-on-Demand Workloads, *Proc. ACM CoNEXT*, pp.363–376 (2014).
[8] Psaras, I., Chai, W.K. and Pavlou, G.: Probabilistic In-network Caching for Information-centric Networks, *Proc. ACM SIGCOMM Workshop on Information-Centric Networking*, pp.55–60 (2012).
[9] Rossini, G. and Rossi, D.: Coupling Caching and Forwarding: Ben-

efits, Analysis, and Implementation, *Proc. ACM ICN*, pp.127–136 (2014).
[10] Chai, W.K., Psaras, I. and Pavlou, G.: Cache "less for more" in information-centric networks, *Proc. IFIP TC 6*, pp.27–40 (2012).
[11] Takemasa, J., Taniguchi, K., Koizumi, Y. and Hasegawa, T.: Identifying Highly Popular Content Improves Forwarding Speed of NDN Software Router, *Proc. IEEE GLOBECOM Workshop on Information Centric Networking Solutions for Real World Applications*, pp.1–6 (2016).
[12] Megiddo, N. and Modha, D.S.: ARC: A Self-Tuning, Low Overhead Replacement Cache, *Proc. USENIX FAST*, pp.115–130 (2003).
[13] Einziger, G., Friedman, R. and Manes, B.: TinyLFU: A highly efficient cache admission policy, *ACM Transactions on Storage*, Vol.13, No.4, pp.35:1–35:31 (2017).
[14] Perino, D. and Varvello, M.: A Reality Check for Content Centric Networking, *Proc. ACM SIGCOMM Workshop on Information-Centric Networking*, pp.44–49 (2011).
[15] Mansilha, R.B., Saino, L., Barcellos, M.P., Gallo, M., Leonardi, E., Perino, D. and Rossi, D.: Hierarchical Content Stores in High-speed ICN Routers: Emulation and Prototype Implementation, *Proc. ACM ICN*, pp.59–68 (2015).
[16] NDN Project: NFD - Named Data Networking Forwarding Daemon (2014).
[17] Broder, A. and Mitzenmacher, M.: Network Applications of Bloom Filters: A Survey, *Internet Mathematics*, Vol.1, No.4, pp.485–509 (2003).
[18] Martina, V., Garetto, M. and Leonardi, E.: A unified approach to the performance analysis of caching systems, *Proc. IEEE INFOCOM*, pp.2040–2048 (2014).
[19] Gill, P., Arlitt, M., Li, Z. and Mahanti, A.: YouTube Traffic Characterization: A View From the Edge, *Proc. ACM IMC*, pp.15–28 (2007).
[20] Imai, S., Leibnitz, K. and Murata, M.: Statistical Approximation of Efficient Caching Mechanisms for One-Timers, *IEEE Trans. Network and Service Management*, Vol.12, No.4, pp.595–604 (2015).
[21] IRCache: IRCache Project (1995).
[22] The Named Data Networking (NDN) project: NDN Packet Format Specification (2014).
[23] Fricker, C., Robert, P., Roberts, J. and Sbihi, N.: Impact of traffic mix on caching performance in a content-centric network, *Proc. IEEE NOMEN*, pp.310–315 (2012).
[24] Karakostas, G. and Serpanos, D.: Exploitation of different types of locality for Web caches, *Proc. IEEE ISCC*, pp.207–212 (2002).
[25] Megiddo, N. and Modha, D.S.: One up on LRU;login, *The Magazine of the USENIX Association*, Vol.4, No.18, pp.7–11 (2003).

Junji Takemasa received his Bachelor and Master of Information Science degrees from Osaka University, Japan, in 2014 and 2016, respectively. He is currently a Japan Society for the Promotion of Science (JSPS) research fellow and is pursuing the Ph.D. degree at the Graduate School of Information Science and Technology, Osaka University. His research interests include Information Centric Networking, high-speed packet switching and green networking. He is a member of IEEE, IEICE and IPSJ.

Yuki Koizumi is an associate professor of Graduate School of Information Science and Technology, Osaka University, Japan. He received his Master of Information Science and Ph.D. of Information Science degrees from Osaka University, Japan, in 2006 and 2009, respectively. His research interests include Information Centric Networking and mobile networking. He is a member of IEEE, ACM, and IEICE.

Toru Hasegawa is a professor of Graduate school of Information and Science, Osaka University. He received his B.E., M.E. and Dr. Informatics degrees in information engineering from Kyoto University, Japan, in 1982, 1984 and 2000, respectively. After receiving the master degree, he worked as a research engineer at KDDI R&D labs. (former KDD R&D Labs.) for 29 years and moved to Osaka University. His current interests are future Internet, Information Centric Networking, mobile computing and so on. He has published over 100 papers in peer-reviewed journals and international conference proceedings including MobiCom, ICNP, IEEE/ACM Transactions on Networking, Computer Communications. He has served on the program or organization committees of several networking conferences such as ICNP, P2P, ICN, CloudNet, ICC, Globecom etc, and as TPC co-chair of Testcom/Fates 2008, ICNP 2010, P2P 2011 and Global Internet Symposium 2014. He received the Meritorious Award on Radio of ARIB in 2003, the best tutorial paper award in 2014 from IEICE and the best paper award in 2015 from IEICE. He is a fellow of IPSJ and IEICE.