INTERNATIONAL JOURNAL OF COMMUNICATION SYSTEMS
Int. J. Commun. Syst. 0000; **00**:1–26
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/dac

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

Atsushi Ooka[1]*, Suyong Eum[1], Shingo Ata[2] and Masayuki Murata[1]

[1] *Graduate School of Information Science and Technology, Osaka University,*
1-5 Yamadaoka, Suita, Osaka, 565-0871 Japan
Tel.: +81-6-6879-4542, Fax: +81-6-6879-4544
[2] *Graduate School of Engineering, Osaka City University,*
3-3-138 Sugimoto, Sumiyoshi-ku, Osaka-shi, Osaka 558-8585, Japan
Tel.: +81-6-6605-2191, Fax: +81-6-6690-5382

SUMMARY

Information-centric networking (ICN) has gained attention from network research communities due to its capability of efficient content dissemination. In-network caching function in ICN plays an important role to achieve the design motivation. However, many researchers on in-network caching have focused on where to cache rather than how to cache: the former is known as contents deployment in the network and the latter is known as cache replacement in an ICN router. Although the cache replacement has been intensively researched in the context of web-caching and content delivery network previously, the conventional approaches cannot be directly applied to ICN due to the fine granularity of chunks in ICN, which eventually changes the access patterns.
In this paper, we argue that ICN requires a novel cache replacement algorithm to fulfill the requirements in the design of a high performance ICN router. Then, we propose a novel cache replacement algorithm to satisfy the requirements named Compact CLOCK with Adaptive Replacement (Compact CAR), which can reduce the consumption of cache memory to one-tenth compared to conventional approaches. Copyright © 0000 John Wiley & Sons, Ltd.

Received ...

KEY WORDS: Information-centric networking; Content-centric networking; Cache replacement algorithm; In-network caching; Router

1. INTRODUCTION

Information-centric networking (ICN) was introduced as a future network architecture which is optimized for content dissemination. ICN is built on the idea of name-based routing which enables each ICN router to be aware of users' requests as well as their counterpart responses. Thus,

*Correspondence to: Graduate School of Information Science and Technology, Osaka University, 1-5 Yamadaoka, Suita, Osaka, 565-0871 Japan. E-mail: a-ooka@ist.osaka-u.ac.jp

Contract/grant sponsor: MIC/SCOPE #165007007

individual ICN routers can be turned into caching devices by simply providing physical cache memory for them.

This feature of ICN that all network devices have caching capability is called in-network caching function, and several ICN architectures, CCNx[1], NDN[2], SAIL[3] and PURSUIT[4], have already suggested utilizing the function to take several advantages of caching system such as reducing network access latency, alleviating network traffic, balancing network load, and achieving robustness against a single failure scenario. In this sense, ICN can be considered as a largely distributed caching architecture whose performance depends on mainly two factors: where to cache and how to cache contents. The former and the latter are known as content placement and cache replacement problems, respectively.

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

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

This paper is organized as follows. In Section 2, we review related research works. In Section 3, we describe the design considerations of a cache replacement algorithm for a core router of ICN. This is followed by a detail description of our proposed method Compact CAR in Section 4. In Section 5, we evaluate our protocol through extensive simulations. Then, we discuss on some implementation issues of our proposal, especially for the design of high performance of ICN core routers in Section 6. Finally, we conclude this article in Section 7.

2. RELATED WORKS

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

Replacement algorithms are developed originally for the purpose of paging in the computer system [5, 6]. The bottleneck of the systems is the latency of fetching pages from slow auxiliary memory to fast cache memory. On the one hand, the hardware cache such as CPU commonly used First-in, first-out (FIFO) and Not Recently Used (NRU) to reduce the memory and computational cost because of the hardly limited resources. On the other hand, the software cache such as virtual memory in OS commonly adopts LRU and LFU, which increase cache hit rate with cost maintaining a data structure or/and statistical information (i.e., the number of references to a page).

As researchers uncover problematic access patterns that degrade the cache hit rate of the algorithms, many variants of LRU and LFU are devised to overcome the problems. 2Q [7], ARC [5] and LIRS [8] improve the hit rate by exploiting the advantages of LRU and LFU while their time and space complexity are comparable to those of LRU. In contrast to them, CLOCK [9] reduces the complexity of LRU by approximating its behavior with a fixed circular buffer while keeping the hit rate. The complexity of CLOCK is comparable to that of NRU which has a low computational cost. CAR [6] combines CLOCK with ARC to achieve both hit rate improvement and cost reduction.

Since web services became explosively popular, web-cache and CDN-cache were researched intensively to improve the performance of them in terms of bottleneck, latency, overload and robustness [10, 11, 12]. Because the resource constraints of them are more moderate than that of computer systems, the cache replacement algorithms in a web and a CDN utilize statistical information including not only recency and frequency but also several others including size, latency, and URI [11]. However, the improvement was trivial or specific to particular environments in spite of an abundance of caching algorithms [12].

In recent years, ICN has revived research on caching algorithms because ICN provides inherent in-network caching feature. Unlike web-cache and CDN-cache employed in the application-layer, all network devices in ICN have caching capability. Thus, caching related researches in ICN have been carried out intensively, especially concerning the locations of content placements. Also, a few of cache replacement algorithms were introduced in [13, 14].

Previous papers on caching use only LRU [15, 16] or claim that the effect on performance of cache replacement is minimal [17]. However, the papers use only blunt cache replacement policies and ignore the suitability for network traffic. In fact, there are studies that exhibit the capability to improve the performance of a network [13, 14]. Cache replcement policy based on content popularity (CCP) [13] can significantly decrease the server load and increase cache hit rate compared to that of LRU and LFU. The work in [14] analyzed the effects of chunking and proposed Highest cost item caching (HECTIC), which uses a utility-based replacement algorithm and outperforms existing polices including LRU. Their statistical approaches are too expensive to be employed in an ICN core router due to computational and memory costs. However, we propose a low-overhead cache replacement policy that outperforms LRU-based and simple replacement policies by coping with access patterns specific to ICN.

To realize ICN, especially an ICN core router, it is also required to implement a cache replacement algorithm that can be operated with severe resource constraints instead of the statistical caching algorithms for a web and a CDN with rich resources. The implementation cost of commonly used approaches such as LRU and LFU is also prohibitive for router hardware, as pointed out in [18, 17].

Looking back at the history of cache replacement algorithms, ICN core routers need a hardware implementable approach whose complexity is comparable to that of FIFO or CLOCK.

3. DESIGN CONSIDERATIONS OF CACHE REPLACEMENT ALGORITHM FOR ICN

3.1. Access Patterns of Traffic in the Network

To understand cache replacement strategy suitable for network traffic without expensive mechanisms (e.g, LFU and statistical approaches), we focus on an access pattern. An access pattern is the important factor to govern the performance of cache replacement algorithms [5, 8, 7, 19]. It is well known that the popularity of contents follows a Zipf-like distribution: a large number of contents are requested only once or just a few times [20]. Although the exact access pattern of the network level traffic in ICN is not known due to the lack of available ICN traffic trace, such one-time used contents occupy 60% or more in the network level traffic in IP networks [21].

A sequence of requests to such one-time used contents forms an access pattern called SCAN [5, 6]. SCAN makes the performance of an LRU-based approach much poor because such unpopular contents occupy the whole cache. In particular, ICN is able to identify a chunk (its default size is 4K bytes in CCNx), which enables the chunk level caching in an ICN router. Thus, we conjecture that the distribution of the "chunk popularity" would have heavier tail than Zipf-like distributions, which renders the effect of SCAN more serious.

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

For the reason above, the cache replacement algorithm for ICN should be able to deal with the access patterns described above. Among the conventional cache replacement algorithms, CAR is able to efficiently deal with the access pattern [6]. CAR is resistant to SCAN traffic access pattern due to its dual lists which enable to distinguish popular and non-popular contents. CAR is also resistant to the volatile popularity because of strategy based on limited-frequency. Our proposal is based on CAR to inherit these features.

3.2. Computational Power and Memory Limitations

In the design of the cache replacement algorithm, two of the performance metrics should be considered. One is the cost that updates the table holding the information of cached items in an ICN router. The other is the cost that holds the table in the memory according to a cache replacement algorithm, e.g., prioritizing cached items. We call the former and the latter as a computational cost and a memory cost, respectively.

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

The memory cost increases as the number of cached items increases due to the increase of control information for the maintenance of the table [22, 6, 18, 17]. For example, LFU has much higher overhead to keep statistics of each cached item. In LRU using a doubly-linked list, this cost is prohibitive due to the maintenance of double pointers to other cached items.

To reduce the computational and memory costs in conventional approaches, CLOCK was introduced. CLOCK is a low-overhead approximation of LRU as mentioned in Section 2. We briefly describe its operation although the detail operation will be explained in Section 4.2. CLOCK has a circular buffer having a shape of a clock. It assigns each entry in a clock list with one-bit, which is called a reference-bit and is set whenever the entry is accessed. CLOCK searches for a cached item that needs to be replaced following a clockwise. While searching for a candidate for replacement, it refers to the reference-bit. When the bit is unset, the cached item is discarded. Otherwise, the searching process keeps on going because an entry with the bit is recently accessed. All bits skipped over during the searching process are unset. Thus, CLOCK requires only a single bit per chunk and a few repetitions of the searching process. Our proposed mechanism also adopts this mechanism of CLOCK to reduce the computational and memory costs.

3.3. Adaptable Parameter Tuning

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

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

4. COMPACT CLOCK WITH ADAPTIVE REPLACEMENT (COMPACT CAR)

4.1. Data Structure of Compact CAR

Compact CAR has two stacks, denoted by L_1 (unshaded) and L_2 (shaded) as shown in Figure 1. Each stack L_i is partitioned into two lists: a left list and a right list in the figure, denoted by T_i and B_i, respectively. Each list is implemented as a CLOCK list, which is known as the low-overhead LRU. T_1 consists of the entries of chunks that are initially cached. T_2 consists of the entries of chunks that have at least one cache hit as well as the entries of chunks are from B_1 and B_2. B_1 and B_2 act as "the losers bracket" providing an opportunity for discarded chunks to be re-cached. Entries in T_i hold information to point to cached chunks, and entries in B_i hold information to keep only the record of chunks discarded from T_i (i.e., the chunks do not exist in the cache memory).

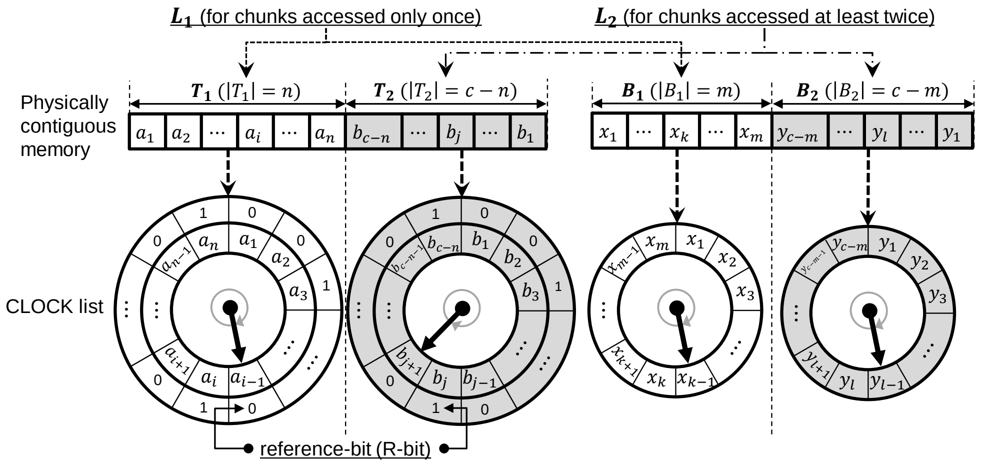

L_1 (for chunks accessed only once)
L_2 (for chunks accessed at least twice)
T_1 ($|T_1| = n$)
T_2 ($|T_2| = c - n$)
B_1 ($|B_1| = m$)
B_2 ($|B_2| = c - m$)
Physically contiguous memory
a_1 a_2 ⋯ a_i ⋯ a_n b_{c-n} ⋯ b_j ⋯ b_1
x_1 ⋯ x_k ⋯ x_m y_{c-m} ⋯ y_l ⋯ y_1
CLOCK list
reference-bit (R-bit)

Figure 1. Data Structure of Compact CAR

The records in B_* are essential to adapt to the variety and dynamism of access patterns as explained later.

T_1 and T_2 are arranged in physically contiguous memory. B_1 and B_2 are arranged in the same manner (upper rectangles in Figure 1). Let T_* denote $T_1 \cup T_2$ and B_* denote $B_1 \cup B_2$ for explanation. T_* has the fixed size of c, where c denotes the number of cacheable chunks in the memory. T_1 and T_2 have n entries and $(c - n)$ entries, respectively. In the same manner, B_1 and B_2 have m entries and $(c - m)$ entries, respectively. Thus, the size of $L_1 \cup L_2$ is $2c$. In addition, we define the maximum size of L_1 as c (i.e., $n + m \leq c$). a_i, b_j, x_k, and y_l represent the entries in T_1, T_2, B_1 and B_2, respectively.

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

4.2. *Operation of Compact CAR*

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

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

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

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

To adaptively control the sizes of T_1 and T_2, Compact CAR defines the target size for T_1. The target size for T_1 is represented as p ($0 < p \leq c$). When the current size of T_1, which is n, is larger or equal to the target size p ($n \geq p$), an entry in T_1 is discarded; otherwise, an entry in T_2 is discarded. By adjusting p, the sizes of the CLOCK lists of T_1 and T_2 vary adaptively. In summary, the target size p governs the behavior of Compact CAR. We will explain how the target size p is dynamically adapted according to traffic access patterns in Section 4.4.

4.3. Design of Low-overhead Variable-sized CLOCK List

As mentioned in the previous section, Compact CAR varies each size of CLOCK list to adapt the change in traffic access patterns. A typical CLOCK list, whose size is fixed, is known as low-overhead because it simply replaces an old entry with new one, which does not change its size. However, the variable-sized CLOCK list in Compact CAR is costly because it changes its size to insert or delete an entry. For instance, when an entry of a new chunk needs to be inserted in T_1, the hand keeps moving to search for an entry with $R = 0$, which will be replaced with the new entry. When the hand encounters an entry with $R = 1$, the entry is removed from T_1 and inserted to T_2. This operation is expensive from the view point of computational and memory costs because it varies the size of CLOCK lists.

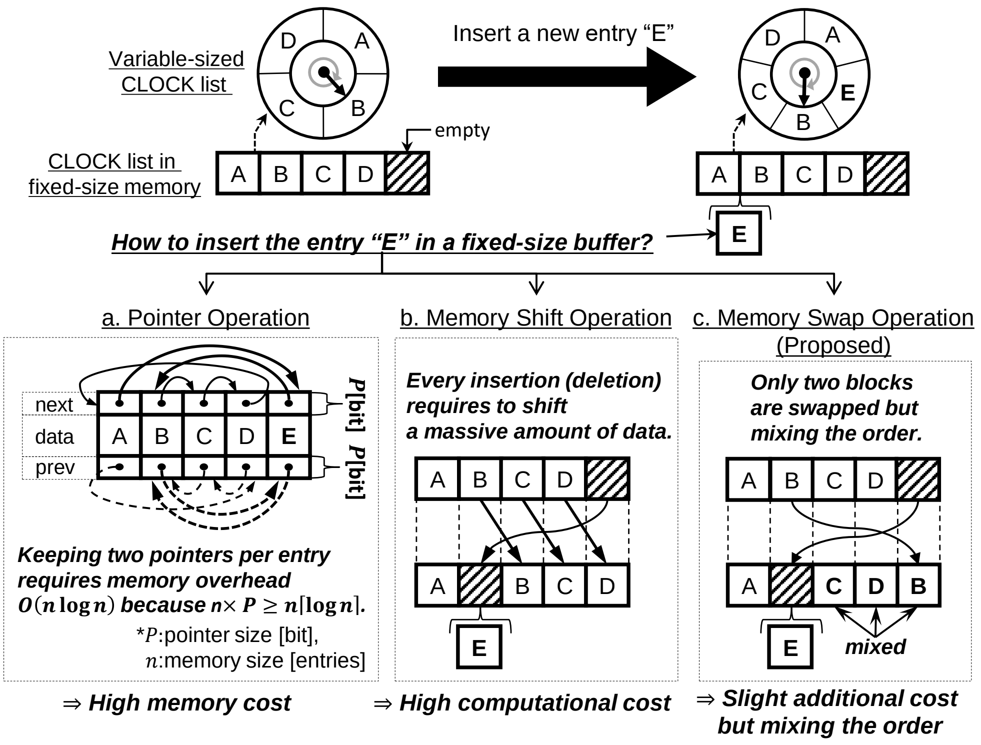

Variable-sized CLOCK list
Insert a new entry "E"
empty
CLOCK list in fixed-size memory
How to insert the entry "E" in a fixed-size buffer?
a. Pointer Operation
b. Memory Shift Operation
c. Memory Swap Operation (Proposed)
next
data
prev
P[bit]
P[bit]
Keeping two pointers per entry requires memory overhead O(n log n) because n× P ≥ n⌈log n⌉.
*P:pointer size [bit], n:memory size [entries]
Every insertion (deletion) requires to shift a massive amount of data.
Only two blocks are swapped but mixing the order.
mixed
⇒ High memory cost
⇒ High computational cost
⇒ Slight additional cost but mixing the order

Figure 2. Illustration of Computational and Memory Costs in the Inserting Operation in the Different Data Structures

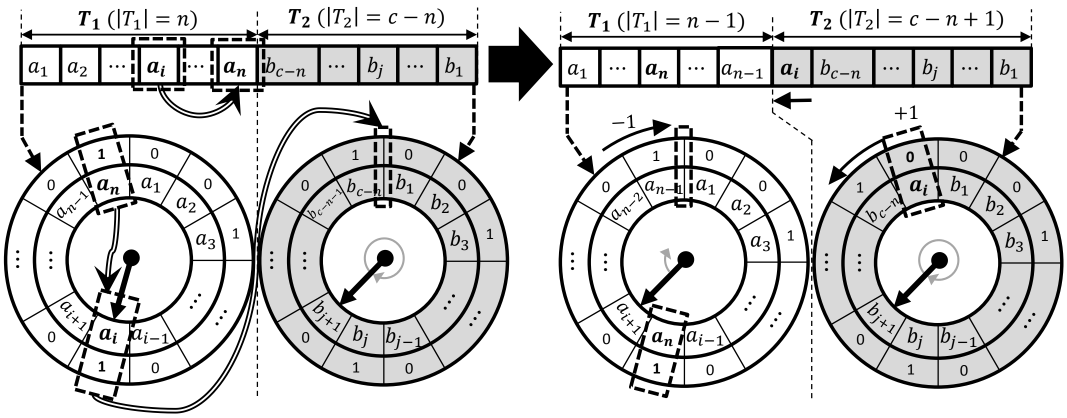


Figure 3. Example of Moving a Chunk a_i from T_1 to T_2 by Replacing the Edge Chunk a_n

To understand the cost of a variable-sized CLOCK list, Figure 2 illustrates the operation of entry insertion in a CLOCK list. Initially, as an example, the CLOCK list is capable of holding five entries and only four entries are currently occupied, which are A, B, C, and D. These entries are stored in fixed-size memory as shown bottom of the CLOCK list in the figure. The hand points to the entry B.

Consider what happens when a new entry E is inserted to the CLOCK list. The entry E is supposed to be inserted between A and B because the position is farthest from the hand. However, we must make a space for E in the physically contiguous memory, which currently four entries occupy. In the bottom of Figure 2, we show three different approaches to insert the entry into the memory from the viewpoint of computational and memory costs.

First, the most left case (a) illustrates a pointer operation with a doubly-linked list. The doubly-linked list introduces additional pointers which manage the order of entries in a CLOCK list. When a chunk is inserted in the middle of memory space, the chunk is inserted physically at the end of the memory space. Then, the order of the chunks in the memory is arranged virtually using the doubly-linked list. It involves two operational costs: computational cost which involves the rearrangement of pointers in the doubly-linked list, and memory cost which involves the memory space accommodating the doubly-linked list. Computational cost is not that expensive. However, it consumes a decent amount of memory space to maintain the order by keeping two pointers per entry (see also Section 6.1 for detail). The original CAR algorithm [6] adopts the pointer operation to insert a new entry.

Second, the case (b) illustrates a case of a memory shift operation. In this case, a doubly-linked list is not used but memory blocks are shifted when an entry is inserted. It does not require high memory cost to maintain pointers because the order of entries in a CLOCK list is realized in physical memory directly without a doubly-linked list in the first scenario. However, this scenario introduces high computational cost caused by the shift of memory blocks (see also Section A.2 for more detail).

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

It may be concerned that the operation changes the order of recency, which may degrade the cache performance of the proposal: the operation shortens a lifetime of a popular entry or vise versa. If “B” is unpopular in this example, the unpopular entry will undesirably occupy the space for a longer time. Thus, the operation mixing the order of a CLOCK lists may make its behavior close to random replacement. However, the influence is not that serious: we will address the issue in Section 5.

Figure 3 gives an example of moving an entry a_i within T_* to realize the swap operation illustrated in Fig. 2(c). Remember the example explained in Section 4.2, where an entry a_i with $R = 1$ is moved from T_1 to T_2. We realize the memory swap operation by exploiting the constancy of the size of T_* when the sizes of T_1 and T_2 change. Compact CAR swaps a_i with the entry a_n at the boundary between T_1 and T_2 in the physically contiguous memory. Then, Compact CAR just shifts the boundary leftward to make a_i be in T_2. This simple swap operation enables the movement of an entry between T_1 and T_2 without computational and memory costs.

4.4. Replacement Algorithm of Compact CAR

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

In the case (2), first, the process discards x from B_* and updates the parameter p, which represents the target size for T_1 as mentioned in Section 4.2 (lines 7–13). At line 6, i stands for the index of the list T_i into which x is to be cached; therefore, i is set to 2 as explained in Section 4.2. The process of tuning p is important to make Compact CAR adaptive to changes in access patterns as discussed in [5, 6]. This process needs to determine (a) whether to increase or decrease p, and (b) the amount of increase or decrease in p.

In respect to (a), when x is in B_1, p increases; otherwise, p decreases. As described in Section 4.2, T_1 and T_2 are devised to effectively cache recently accessed chunks and frequently accessed chunks, respectively. Since B_1 and B_2 correspond to T_1 and T_2, respectively, we can adaptively control the behavior of Compact CAR by changing p according to accesses to B_1 and B_2. The tuning process increases p when x is in B_1 because the access indicates recently accessed chunks are becoming important. On the other hand, when x is in B_2, p decreases to place importance on frequency.

In respect to (b), we determine the amount of increase (decrease) δ according to the ratio of $|B_1|$ to $|B_2|$, where $|B_i|$ represents the size of B_i, to adapt the changes of access patterns rapidly. To explain the reason, let us consider the case where there is an access to B_1 when $|B_2|$ is larger than $|B_1|$. Intuitively, the fact that $|B_2| > |B_1|$ indicates frequently accessed chunks were more important than recently accessed chunks until now; however, the current access to B_1 indicates recently accessed chunks are becoming important. Thus, when $|B_2| > |B_1|$, the tuning process sets $\delta = |B_2|/|B_1|$ to rapidly adapt the access patterns where recency is more important; otherwise, $\delta = 1$.

In the case (3), line 15 sets i to 1 to cache the new entry x in T_1. Lines 16–18 ensure that there is a room in B_* because an existing entry in T_1 (T_2) is replaced by x and is moved to B_1 (B_2) at lines 21–23. If the lists in Compact CAR are not full (i.e., $|L_1| < c$ and $|L_1 \cup L_2| < 2c$ as defined in Section 4.1), the entry is simply inserted into B_*. If L_1 is full (i.e., $|L_1| = c$), the replacement process discards an entry in B_1 to insert the entry. If L_1 is not full and $L_1 \cup L_2$ is full (i.e., $|L_1 \cup L_2| = 2c$), the process discards an entry in B_2.

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

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

Algorithm 1 Compact CAR Replacement Algorithm

```
1: procedure CACHEREPLACEMENT($x$)                          ▷ $x$ is an accessed entry.
2:     if $x \in T_*$ then                                  ▷ Cache hit
3:         $x$.R-bit$\leftarrow 1$
4:         return
5:     else if $x \in B_*$ then                             ▷ Record of discard remains
6:         $i \leftarrow 2$                                 ▷ To cache $x$ in $T_2$
7:         if $x \in B_1$ then
8:             $\delta \leftarrow \max(1, \frac{|B_2|}{|B_1|})$; $p \leftarrow \min(c, p + \delta)$
9:             DiscardBottom(1,$x$)
10:        else                                             ▷ $x \in B_2$
11:            $\delta \leftarrow \max(1, \frac{|B_1|}{|B_2|})$; $p \leftarrow \max(0, p - \delta)$
12:            DiscardBottom(2,$x$)
13:        end if
14:    else                                                 ▷ Cache miss
15:        $i \leftarrow 1$                                 ▷ To cache $x$ in $T_1$
16:        if Full($L_1$) & $|B_1| > 0$ then ReplaceBottom(1)
17:        else if Full($L_1 \cup L_2$) & $|B_2| > 0$ then ReplaceBottom(2)
18:        end if
19:    end if
20:    if Full($T_*$) then
21:        if $|T_1| \geq \max(p, 1)$ then $s_t \leftarrow$ ReplaceTop(1)
22:        else $s_t \leftarrow$ ReplaceTop(2)
23:        end if
24:    else                                                 ▷ $T_*$ is not full.
25:        $s_t \leftarrow$ an available address in $T_i$
26:    end if
27:    $T_i[s_t] \leftarrow x$                              ▷ $x$ is cached as an entry in $T_i$.
28: end procedure
```

Algorithm 2 DiscardBottom() for Compact CAR

```
1: procedure DISCARDBOTTOM($i, x$)
2:     Swap($x, B_i$.EdgeEntry)
3:     Discard $x$ (at the edge of $B_i$)    ▷ Ensuring that the address next to the edge of $B_i$ is free
4: end procedure
```

Algorithm 3 ReplaceBottom() for Compact CAR

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

5. PERFORMANCE EVALUATION

In this section, we evaluate the performance of Compact CAR compared to OPT (off-line optimal algorithm with a priori knowledge of the stream of requests: absolute upper bound on the achievable cache hit rate), FIFO, CLOCK, and CAR in various scenarios to demonstrate the fulfillment of the design considerations discussed previously.

Algorithm 4 ReplaceTop() for Compact CAR

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

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

5.1. Simulation Setup and Configuration

Two types of workloads are used in this simulation study: artificial workloads that follow a Zipf distribution and real traffic traces of Video-on-Demand (VoD), e.g., YouTube, DailyMotion, and NicoVideo, which are collected from a network gateway at Osaka University campus. The former and the latter are denoted by $A_{Zipf(\alpha)}$ and by A_{Real}, respectively. In addition, their superscript C and P, e.g., $A^{C}_{Zipf(a)}$ and $A^{P}_{Zipf(\alpha)}$ represent the workloads in units of content and chunks, respectively.

The popularity of Internet content (e.g., VoD, web pages, file sharing, and user generated traffic) has been reported to follow the Zipf distribution with $0.6 \leq \alpha \leq 1.2$ [23, 21]. Thus, we use these values to generate synthetic traffic requests from the Zipf distribution for this simulation study.

To justify the results using synthetic traffic, we also use the real traffic traces. The traces are gathered from July 26th 2013 to February 26th 2015. The number of unique contents is 2,428,880; the number of contents requested at least twice is 918,545; and the number of total accesses is 13,004,868. The popularity distribution of the real traffic trace follows the Zipf-like distribution, as depicted in Fig. 4. We also show the statistics of the real traffic traces in units of chunks in Table II.

As stated in Section 3.1, the fine granularity of chunks in ICN, namely chunks or segments, changes the access patterns of request message, which dramatically governs the performance of cache replacement algorithm. Unfortunately, ICN traffic traces are not available yet. Thus, we generate synthetic requests for chunks, which simulates the access pattern of ICN in the following manner. We determines the inter-arrival time between requests to content according to our observed

Table I. Number of Chunks Per Second [pck/s]

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

Table II. Statistics of Workloads in Units of Chunks

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

real trace. On the other hand, the inter-arrival time for chunks is constant according to Table I, which is determined by the statistics of our observed real traffic. The generated requests are superimposed to simulate the aggregation of request messages in the network.

This simulation studies the cache performance of ideal-cooperative caching and non-cooperative caching[†] by using two topologies: the one-node topology and the line topology. One is the topology in which there is only one ICN router between clients and a server. The other is the line topology that includes ten ICN routers between them. As mentioned previously, the cache hit rate is governed by two factors: one is a cache replacement algorithm (how to cache), and the other is a cooperative caching algorithm (where to cache). Because the purpose of our simulation is to evaluate the performance of the former, we do not investigate the performance of the latter in detail but rather we consider its best and worst case performance to make the analysis complete.

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

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

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

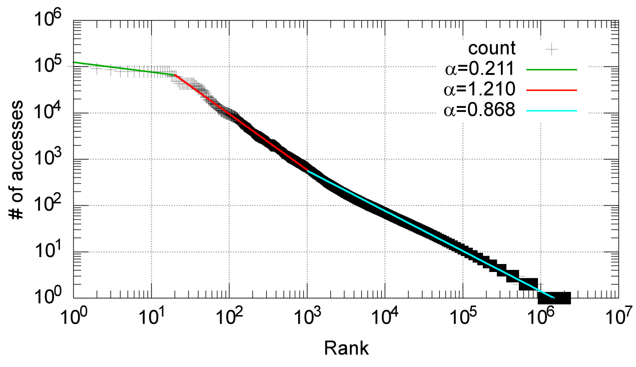
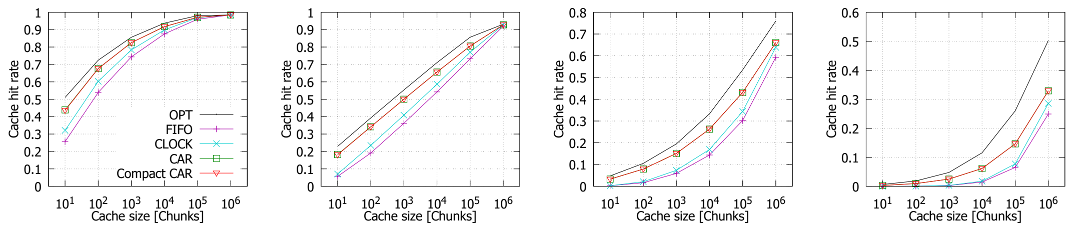
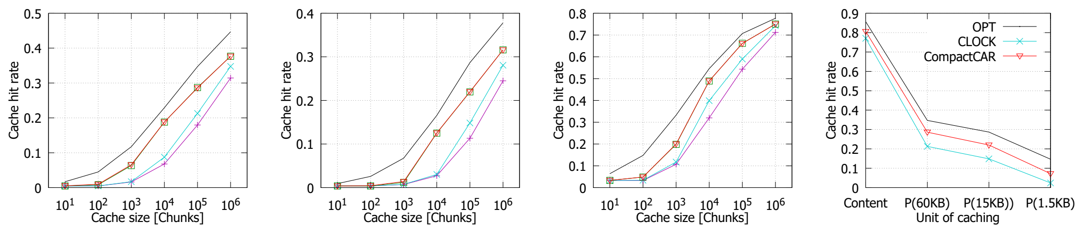


Figure 4. Popularity Distribution of Real Trace



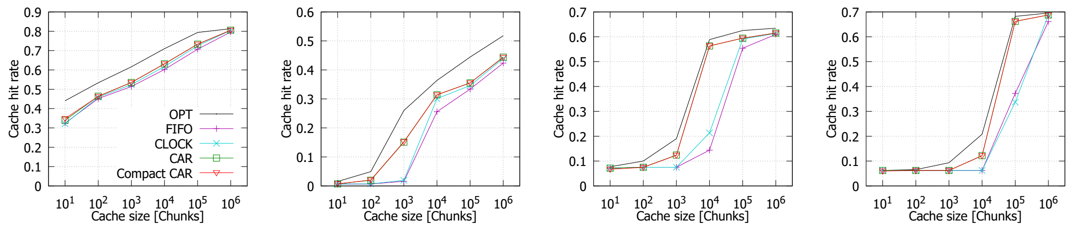
(a) $A^{C}_{\mathrm{Zipf}(1.2)}$ (b) $A^{C}_{\mathrm{Zipf}(1.0)}$ (c) $A^{C}_{\mathrm{Zipf}(0.8)}$ (d) $A^{C}_{\mathrm{Zipf}(0.6)}$

Figure 5. Results for Synthetic Traffic in Units of Content



(a) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ (b) $A^{P(15\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ (c) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.2)}$ (d) $A_{\mathrm{Zipf}(1.0)}$, $c = 10^5$

Figure 6. Results for Synthetic Traffic in Units of Chunks

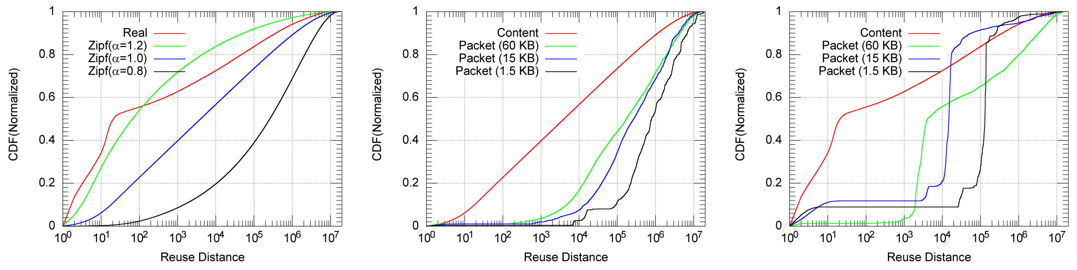


(a) A^{C}_{Real} (b) $A^{P(60\mathrm{KB})}_{\mathrm{Real}}$ (c) $A^{P(15\mathrm{KB})}_{\mathrm{Real}}$ (d) $A^{P(1.5\mathrm{KB})}_{\mathrm{Real}}$

Figure 7. Results for Real Traffic Trace

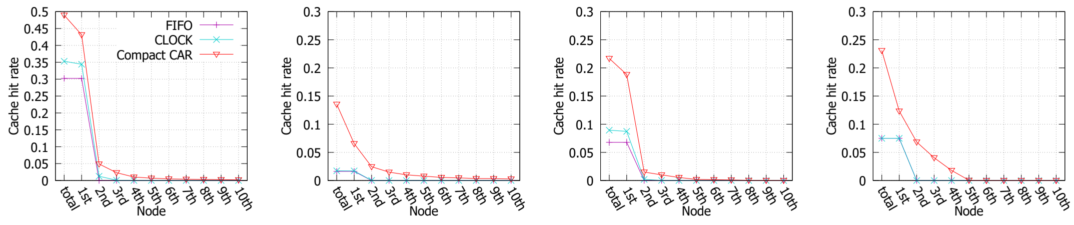
5.2. Cache Hit Rate with a SCAN Access Pattern using Synthetic Traffic

Figure 5 depicts the cache hit rate of each cache replacement policy in the one-node topology with synthetic traffic described previously: $A^{C}_{\mathrm{Zipf}(\alpha)}$ changing α from 0.6 to 1.2. Our proposal, Compact



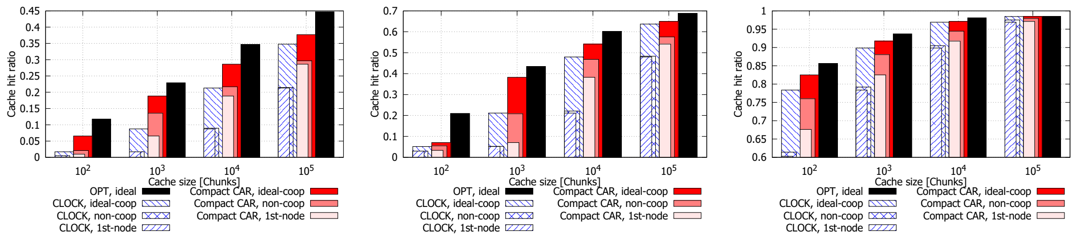
(a) Different Types of Workloads in Units of Content (b) Artificial Workloads ($\alpha = 1.0$) in Different Units (c) Real Traces in Different Units

Figure 8. CDF of RD in Various Workloads



(a) $A^{C}_{\mathrm{Zipf}(0.8)}$ $(c = 10^5)$ (b) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ $(c = 10^3)$ (c) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ $(c = 10^4)$ (d) $A^{P(15\mathrm{KB})}_{\mathrm{Real}}$ $(c = 10^3)$

Figure 9. Results for Simulation with the Line Topology



(a) $A^{P(15\mathrm{KB})}_{\mathrm{Zipf}(1.0)}$ (b) $A^{P(60\mathrm{KB})}_{\mathrm{Zipf}(1.2)}$ (c) A^{C}_{Real}

Figure 10. Comparison between Non-cooperative Caching and Ideally-cooperative Caching

CAR, achieves a hit rate comparable to that of CAR, which is contrary to our speculation. We conjectured that the operation mixing the order in the Compact CAR would degrade its performance because the operation makes its behavior close to random replacement. The result is promising because we can achieve the performance as good as CAR even with much less memory cost. The memory cost of Compact CAR including several others is theoretically analyzed in Section 5.6 in detail. In addition, the results show that Compact CAR can achieve the same cache hit ratio with one-tenth of cache size compared to simple cache replacement algorithms such as FIFO and CLOCK in the best case. This is because the two-stack approach of Compact CAR prevent popular content from being removed from the cache by SCAN.

In addition to the simulation using traces in units of contents, Figure 6 shows the cases when the sizes of chunks change from 60 KB to 1.5 KB with the parameters of the Zipf distribution α at 1.0 and 1.2, which are denoted by, e.g., $A^{P(60KB)}_{Zipf(0.6)}$ to $A^{P(1.5KB)}_{Zipf(0.6)}$. As the value of α increases, the hit rate increases. This means that a high popularity bias results in a high hit rate as known in the previous

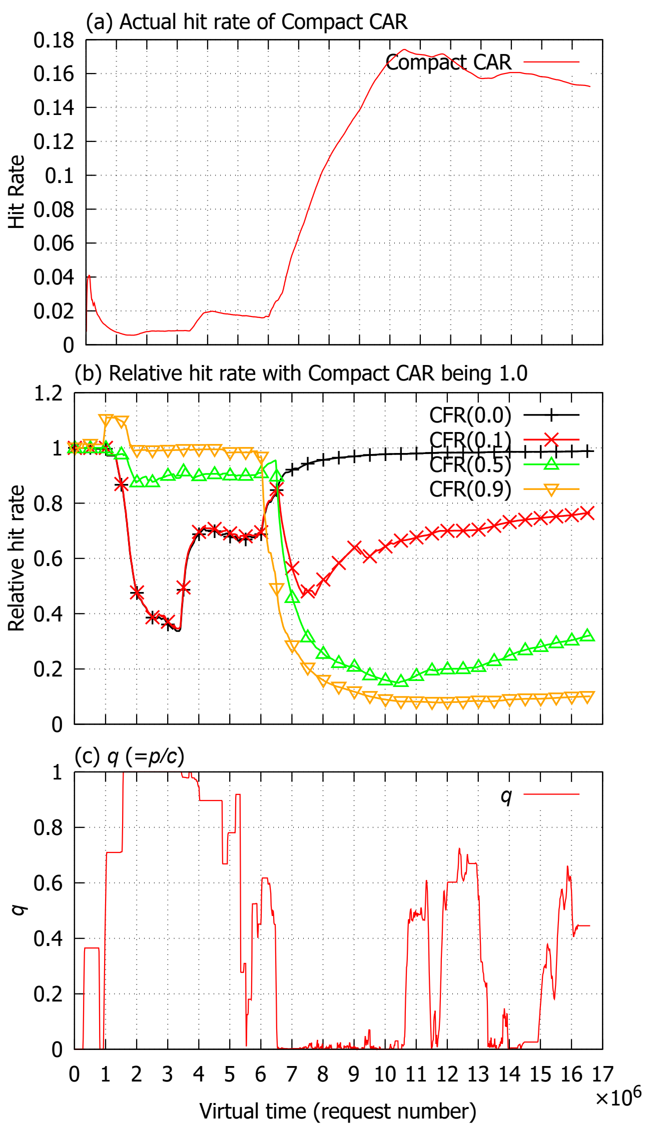


Figure 11. Dynamics of Hit Rate of $\mathrm{CFR}(q)$and Adaptive Parameter q

studies. As depicted in Fig.6(d), we observe that the cache hit rate decreases substantially as the size of chunks becomes small, e.g., from a whole content to chunks.

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

Figure 7 presents the simulation results in the one-node topology with real Video-on-demand (VoD) traffic which was collected at Osaka University. Because the original traces are in units of content, we divide each content into small sized chunks to simulate the traces in units of chunks in ICN networks. The cache hit rate in Fig. 7 are similar to those in Fig. 5 and Fig. 6. In Fig. 7, one interesting observation is that the cache hit rate of our proposed algorithm suddenly soars, e.g., when cache size is 10^4 in Fig. 7(c) compared to conventional cache replacement algorithms:

the performance becomes outstanding. This phenomenon correlates to the Reuse Distance (RD); therefore, we discuss it below.

Figure 8 plots the cumulative distribution functions (CDFs) of RD. RD represents the number of requests between two consequtive requests for the same chunk. For example, consider what happens when the value of RD is larger than the size[‡] of cache. The cache of the chunk requested by the first request has a high probability to be discarded from the cache before the second request arrives. If this case keeps happening due to a large amount of one-time contents (e.g., SCAN), only non-popular contents remain in the cache. This situation is called cache pollution that non-popular contents occupy whole cache causing low-cache hit rate. Thus, a cache hit almost occurs when RD is smaller than cache size, and vice versa.

As mentioned in Section 4, Compact CAR maintains two link lists: one for non-popular contents, and the other for popular contents. Thus, the cache pollution only affects to the link list that maintains non-popular contents. In other words, Compact CAR is robust to the cache pollution scenario caused by a large amount of non-popular contents.

5.4. *Simulation with the Line Topology*

Figure 9 presents the cache hit rate of individual nodes on the line topology. We omit the graph of CAR because the hit rate of CAR are almost identical to Compact CAR. Compact CAR improves the hit rate in the second and succeeding routers, whereas the hit rate of FIFO and CLOCK decreased to approximately zero. Figure 10 shows the upper and lower bound of the performance achieved by cooperative caching. The performance of ideally cooperative caching is denoted by “ideal-coop”, which specifies the upper bound. The result denoted by “non-coop” means the total cache hit rate of nodes in the line topology, which is the performance of non-cooperative caching and specifies the lower bound. We also show the hit rate of the only first node of the line topology as “1st-node” to understand how CLOCK is inappropriate for the environment without cooperative caching. There is less difference between the upper bound and the lower bound of Compact CAR than that of CLOCK. This result indicates Compact CAR can exploit resources in a network by reducing redundant caches caused by the cooperation failure.

It is interesting to analyze the performance under an environment with a certain cooperation or a cache decision algorithm; however, we do not show the analysis because the main purpose of this paper is proposing the cache replacement algorithm that is feasible and appropriate for an ICN router. In future, we will investigate the effects of various cache placement and decision algorithms on a network and communication quality.

5.5. *Dynamic Parameter Tuning*

As explained in Section 4.3, Compact CAR dynamically adapts to changing traffic access patterns by varying the parameter p. There is no one-size-fits-all parameter and it is necessary that the parameter should be tuned to maximize cache hit rate under any circumstances.

Here we evaluate the parameter tuning strategy for the proposed Compact CAR whose parameter p represents the target size for T_1. The parameter p ranges from zero to the cache size c. As the

[‡]Its unit is the number of chunks

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

Thus, depending on the variation of access patterns, the parameter p should be tuned. To compare the difference between dynamical tuning and statical tuning, we introduce Clock with Fixed Replacement (CFR) algorithm which corresponds to our proposal Compact Clock with Adaptive Replacement (Compact CAR). CFR has the fixed value of $q = p/c$ $(0 \leq q \leq 1)$ which is determined in advance.

Figure 11 shows that Compact CAR adaptively changes the parameter: the trends of q and the cache hit rate of CFR(q). The x-axis shows the virtual time t, which is equivalent to the total number of requests. The cache hit rate of CFR(q) are shown as relative value with that of Compact CAR being 1.0 in Fig. 11 (b). When $0 < t < 6 \times 10^6$, CFR(q) with high q achieves the high hit rate, and vice versa. When $t = 6 \times 10^6$, we can observe the rapid increase in the cache hit rate of Compact CAR. This increase is due to an arrival of many popular contents. Thus, the value of q decreases to adopt the access patterns, where frequently accessed content becomes important, and the corresponding hit rate of CFR(q) increases. The results show that Compact CAR can adaptively change the parameter. In addition, q of Compact CAR continues to follow the optimal value at any time as evidenced by the fact that the best relative hit rate among CFR(q) are at most nearly 1.0. By contrast, the relative hit rate of the parameter fixed algorithms become at worst nearly 0.1. Thus, we can confirm that the parameter tuning algorithm of Compact CAR are necessary and greatly adaptive.

5.6. Analysis on Space and Time Complexities of CAR and Compact CAR

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

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

Table III. Time Complexity of Cache Replacement Algorithm's Overhead

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

Table IV. Space Complexity of Cache Replacement Algorithm's Overhead

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

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

5.6.1. Space Complexity First, we analyze the space complexity of CAR and Compact CAR. Compact CAR maintains four CLOCK lists shown in Fig. 1. Our simple swapping renders Compact CAR free from the additional costs of memory or process for maintaining the order of sweeping the CLOCK list. Furthermore, B_1 and B_2 do not need R-bits and the total length of the other two CLOCK lists, T_1 and T_2, is n. Thus, Compact CAR consumes $(n + 9P)$ bits for the two normal CLOCK lists whose total length is n, the two CLOCK lists without a R-bit, four information of the size of the lists, and a parameter of a target size for T_1.

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

5.6.2. Time Complexity Then, we elaborate the time complexity of CAR and Compact CAR. Since many of the analysis is overlapped, we first elaborate the time complexity of CAR, followed by that of Compact CAR. CAR as well as CLOCK incurs $t_w + \delta$ complexity at a cache hit since it requires only to update R-bit. The worst-case complexity at a cache miss is $O(n)$ because the hand must move n times to go around the clock in the worst case where R-bit of all entries in CLOCK is set.

The average number of hand movements at a cache miss ω is represented as n/s, where s is the number of cache misses during n hand movements. Because we aim to calculate the order of ω, our analysis can be simplified by considering the extreme case where ω is maximized in the steady state. Therefore, we consider two cases where n is maximized, and where s is minimized. For brevity, we do not show how to maximize n and minimize s here, which is obtained by the same calculation as CLOCK discussed in Appendix A.3. The difference between CLOCK and CAR is that we must count not only the first and second accesses to a chunk but also the third accesses should to maximize n since the accesses turn on R-bits of entries in T_2. According to the calculation, ω satisfies the following inequality:

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

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

6. DISCUSSION ON THE IMPLEMENTATION OF COMPACT CAR FOR HIGH PERFORMANCE ICN CORE ROUTER

6.1. Feasibility of Hardware Implementation

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

Figure 12 shows a memory overhead of CAR and Compact CAR. As explained in Section 5.6.1, CAR using a doubly-linked list consumes $(4Pn + n + 9P)$ bits. Assuming a router holds 20 million cache entries, the memory cost of CAR becomes 2 Gbit to hold 20 million entries because $P \geq \lceil \log n \rceil$. This cost is prohibitive according to the constraint of SRAM, whose available size is

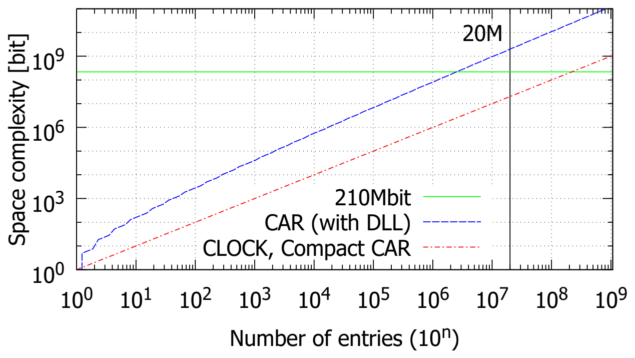


Figure 12. Space Complexities of CAR and Our Proposal(Compact CAR)

210 Mbit [24]. On the other hand, Compact CAR requires a memory overhead of one bit per entry. Compact CAR consumes 20 Mbit; therefore the memory cost of Compact CAR is feasible.

6.2. *Computational Overhead of Variants of CLOCK*

In Section 5.6.2, we analyzed the computational cost of Compact CAR, which provides the complexity of $O(1/(1-\beta))$ in terms of β. It may be arguable that the complexity could be extremely large as the parameter β becomes close to 1.0. In fact, the β values of content-level and packet-level workloads used in our simulation ranges from 0.38 to 0.71 and from 0.08 to 0.22, respectively. $\frac{1+2\beta}{1-\beta}$ showing average-case time complexity of Compact CAR is less than only 2.0 when $\beta < 0.2$. $\frac{1+2\beta}{1-\beta}$ grows 6.0, which is the computational cost of LRU, when β becomes 0.625. $\frac{1+2\beta}{1-\beta} < 8.0$ even if $\beta < 0.7$. Although the space complexity of CAR can be reduced by using a memory shift operation instead of a doubly-linked list, the memory shift operation makes the time complexity prohibitive as illustrated in Fig. 2(b).

If SRAM access time is 0.45 ns [24], the router can handle about 278 million accesses per second even if eight hand movements per access are required as discussed above. Assuming 64-byte data packets, this throughput is 142 Gbps. In conclusion, the computational cost of Compact CAR is acceptable in the design of high performance ICN core router.

However, the data of the chunks must be kept in a scalable memory, such as dynamic RAM or a solid-state disk. Since such memory is slow, we plan to consider a hierarchically structured cache memory and a pipelined process to ensure a high average speed for read/write accesses. We will eventually evaluate the router performance in a hardware implementation of the router, combining Compact CAR and a name lookup entity [25], to demonstrate the feasibility of the router.

7. CONCLUSIONS

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

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

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

ACKNOWLEDGEMENT

This research and development work was supported by the MIC/SCOPE #165007007.

REFERENCES

1. Jacobson V, Smetters DK, Thornton JD, Plass MF, Briggs NH, Braynard RL. Networking named content. *Proceedings of the ACM CoNEXT 2009*, 2009; 1–12.
2. Zhang L, Estrin D, Burke J, Jacobson V, Thornton JD, Smetters DK, Zhang B, Tsudik G, Claffy K, Krioukov D, *et al.*. Named data networking (NDN) project October 2010. URL http://named-data.net/techreport/TR001ndn-proj.pdf.
3. Levä T, Gonçalves J, Ferreira RJ, *et al.*. Description of project wide scenarios and use cases February 2011. URL http://www.sail-project.eu/wp-content/uploads/2011/02/SAIL_D21_Project_wide_Scenarios_and_Use_cases_Public_Final.pdf.
4. Fotiou N, Nikander P, Trossen D, Polyzos GC. Developing information networking further: From PSIRP to PURSUIT. *Proceedings of the 7th International ICST Conference on Broadband Communications,Networks, and Systems*, 2010; 1–13.
5. Megiddo N, Modha DS. ARC: a self-tuning, low overhead replacement cache. *Proceedings of the 2Nd USENIX Conference on File and Storage Technologies*, 2003; 115–130.
6. Bansal S, Modha DS. CAR: Clock with adaptive replacement. *Proceedings of the 3rd USENIX Conference on File and Storage Technologies*, 2004; 187–200.
7. Johnson T, Shasha D. 2Q: a low overhead high performance buffer management replacement algorithm. *Proceedings of the 20th International Conference on Very Large Data Bases*, 1994; 439–450.
8. Jiang S, Zhang X. Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance. *IEEE Transactions on Computers* August 2005; **54**(8):939–952.
9. Corbato FJ. A paging experiment with the Multics system. *Technical Report*, DTIC Document May 1968.
10. Wang J. A survey of web caching schemes for the Internet. *ACM SIGCOMM Computer Communication Review* October 1999; **29**(5):36–46.
11. Wong KY. Web cache replacement policies: a pragmatic approach. *IEEE Network* January 2006; **20**(1):28–34.
12. Pathan AMK, Buyya R. A taxonomy and survey of content delivery networks. *Technical Report*, University of Melbourne Grid Computing and Distributed Systems Laboratory February 2007.

13. Ran J, Lv N, Zhang D, Ma Y, Xie Z. On performance of cache policies in named data networking. *Proceedings of the International Conference on Advanced Computer Science and Electronics Information 2013*, 2013; 668–671.
14. Wang L, Bayhan S, Kangasharju J. Optimal chunking and partial caching in information-centric networks. *Computer Communications* May 2015; **61**(1):48–57.
15. Chai WK, He D, Psaras I, Pavlou G. Cache "less for more" in information-centric networks. *Computer Communications* May 2012; **36**(7):758–770.
16. Safari Khatouni A, Mellia M, Venturini L, Perino D, Gallo M. Performance comparison and optimization of ICN prototypes. *Proceedings of 2016 IEEE GLOBECOM*, 2016; 1–6.
17. Rossi D, Rossini G. Caching performance of content centric networks under multi-path routing (and more). *Technical Report*, Telecom ParisTech July 2011.
18. Arianfar S, Nikander P, Ott J. Packet-level caching for information-centric networking. *Technical Report*, Finnish ICT SHOK June 2010.
19. Jaleel A, Theobald KB, Steely SC Jr, Emer J. High performance cache replacement using re-reference interval prediction (RRIP). *ACM SIGARCH Computer Architecture News* June 2010; **38**(3):60–71.
20. Breslau L, Cao P, Fan L, Phillips G, Shenker S. Web caching and Zipf-like distributions: evidence and implications. *Proceedings of IEEE INFOCOM'99*, vol. 1, 1999; 126–134.
21. Guillemin F, Kauffmann B, Moteau S, Simonian A. Experimental analysis of caching efficiency for YouTube traffic in an ISP network. *Proceedings of the 25th International Teletraffic Congress*, 2013; 1–9.
22. Jiang S, Chen F, Zhang X. CLOCK-Pro: An effective improvement of the CLOCK replacement. *Proceedings of the USENIX 2005*, 2005; 323–336.
23. Fricker C, Robert P, Roberts J, Sbihi N. Impact of traffic mix on caching performance in a content-centric network. *Proceedings of the IEEE Conference on Computer Communications 2012*, 2012; 310–315.
24. Perino D, Varvello M. A reality check for Content Centric Networking. *Proceedings of the ACM SIGCOMM workshop on Information-centric networking*, 2011; 44–49.
25. Ooka A, Ata S, Inoue K, Murata M. High-speed design of conflict-less name lookup and efficient selective cache on CCN router. *IEICE Transactions on Communications* April 2015; **E98-B**(04):607–620.

A. TIME AND SPACE COMPLEXITY OF THE REMAINING POLICIES

We analyze the time and space complexity of policies which are skipped in Section 5.6.

We analyze them in the same manner as described in Section 5.6. In addition to the notations in Section 5.6, we define the following notations and variables. m is the number of records of discarded chunks in LIRS. Statistical policies assign C-bit information (as a counter used in LFU) to every entry.

A.1. Complexity of FIFO

In FIFO, only a P-bit pointer to remember the head of the queue is required. When a cache hit occurs, no additional operations are necessary (except for common operations such as reading the accessed chunk). When a cache miss occurs, there are two additional operations: reading the pointer to discard the entry at the head of the queue and updating it. Thus, the space complexity is P bits. The time complexity at a cache hit and miss are δ and $(t_r + t_w + \delta)$, respectively.

A.2. Complexity of LRU

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

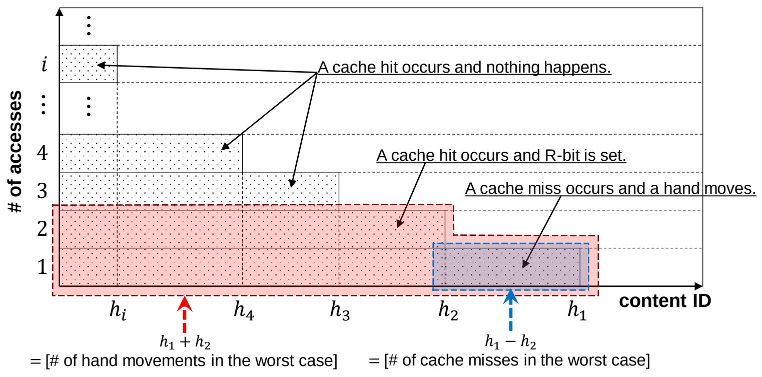
Let e_i denote the i-th most recently accessed entry in LRU$_{DLL}$ $(i = 1, 2, \cdots, n)$, that is, smaller i means that the entry is more recent. In addition, p_i^{prev} and p_i^{next} denote the pointers that point the previous and next entry, respectively. If e_i is accessed, e_i is moved to the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

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

However, LRU$_S$ requires shifting a large amount of entries to insert or move an entry just like the algorithm described in Section 4.3. If e_i is accessed, all entries from e_1 to e_{i-1} must be shifted. If there is a cache miss, it is required to shift entries from e_1 to e_{n-1} and write a new entry at the top of the stack. In the worst case, n entries are moved. On average, $n/2$ entries are moved at a cache hit if all entries are uniformly referenced. Thus, time complexity of LRU$_S$ is $O(n)$. This process in a small-scale computer system is typically supported by special hardware for the shifting operation; however, it is infeasible for use in an ICN router because of an excessive amount of entries.

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

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



# of accesses
i
4
3
2
1
content ID
A cache hit occurs and nothing happens.
A cache hit occurs and R-bit is set.
A cache miss occurs and a hand moves.
h_i
h_4
h_3
h_2
h_1
$h_1 + h_2$
= [# of hand movements in the worst case]
$h_1 - h_2$
= [# of cache misses in the worst case]

Figure 13. Description of calculating ω of CLOCK

A.3. *Complexity of CLOCK*

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

Let s denote the average number of cache misses during one cycle of a hand (i.e., n hand movements) to calculate the average-case time complexity $\omega = n/s$, which can be defined as the number of hand movements per cache miss on average. Fig. 13 gives an intuitive understanding of how to calculate n and s according to h_i defined in the time interval $[1, n]$ during n hand movements.

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

First, we discuss the case where n is maximized. It is obvious that the first access to a chunk causes a cache miss and rotation of a hand. A cache hit by the second access to a chunk set R-bit of the accessed entry. This entry whose R-bit is set causes a movement of a hand because the hand ignores the entry only resetting the R-bit. Even if a chunk is accessed three or more times per cycle, the accesses do not cause a hand movement. Therefore, the number of hand movements to go around CLOCK's circular list is at most $h_1 + h_2$ as illustrated in Fig. 13 (a red area).

Second, we determine the minimum number of cache misses s. It is clear that $s = 1$ at the minimum in the worst case where $(h_1 - 1)$ chunks have been already accessed and their R-bits are set before our considering time interval $[1, n]$. However, assuming the steady state where the popularity distribution of chunks (i.e. the distribution of h_i) is stable, there is at most h_2 chunks that is accessed before the beginning of the interval. Therefore, the number of cache misses is at least $h_1 - h_2$ as illustrated in Fig. 13 (a blue area).

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

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

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

A.4. *Complexity of LFU$_H$*

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

A.5. Complexity of ARC

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

A.6. Complexity of LIRS

LIRS uses two LRU lists which are called LRU stack S and Q. The maximum size of LRU S and Q is $(n + m)$ and n, respectively. In addition, two bits are assigned to each entry to mark a hot chunk[§] and a record of a discarded chunk. Thus, the space complexity is $(4Pn + 2n + 2Pm + 4P)$. m is practically smaller than $4n$ [8] although the length of m, which is determined by the length of a sequence of one-time content such as SCAN, is theoretically unlimited.

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

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

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