

Compact CAR: Low-Overhead Cache Replacement Policy for an ICN Router

Atsushi Ooka^a, Suyong Eum^a, Shingo Ata^b, Masayuki Murata^a

^aGraduate 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

^bGraduate 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

Abstract

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 element. 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 cacheable items 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 high performance ICN element. 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.

Keywords: 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 element to be aware of users' requests as well as their counterpart responses. Thus, individual ICN elements can be turned into caching devices by simply providing physical cache memory for them.

This feature of ICN that all elements 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 ignored since many people believe that the problem has already been investigated intensively in the context of web-caching and content delivery network. However, it is unclear that the conventional cache replacement approaches are suitable for ICN due to following two reasons. First, core ICN elements 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 cacheable items in ICN, namely chunks or segments, changes the traffic access patterns of request messages, which dramatically govern the performance of cache replacement algorithm.

In the light of the observation above, this paper studies the cache replacement problem in the core ICN elements, and so they can support the line-rate operation, which is critical in the design of core ICN elements. First, we analyze the access patterns of contents to understand its relation to cache replacement algorithms. Then, we propose a novel cache replacement algorithm named Compact CLOCK with Adaptive Replacement (Compact CAR) to fulfill the requirements. The proposed algorithm is based on CLOCK that is a classical cache replacement

Email addresses: a-ooka@ist.osaka-u.ac.jp (Atsushi Ooka),
suyong@ist.osaka-u.ac.jp (Suyong Eum),
ata@info.eng.osaka-cu.ac.jp (Shingo Ata),
murata@ist.osaka-u.ac.jp (Masayuki Murata)

policy to achieve low-complexity approximation. 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 cache replacement algorithm for a core element 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 element 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 the 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 cost because of the hardly limited resources. On the other hand, the software cache such as virtual memory of OS commonly adopts LRU and LFU, which are costly to maintain a data structure or/and statistical information (i.e., the number of references to a page).

As researchers uncover problematic access patterns that degrade the performance of the algorithms, many variants of LRU and LFU are devised to overcome the problems. 2Q [7], ARC [5] and LIRS [8] improve the performance by exploiting the advantages of LRU and LFU while their time and space complexities are comparable to that 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 performance. 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 performance improvement and cost reduction.

Since web services became explosively popular, web-cache and CDN-cache are 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 slight 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- and CDN-cache employed in the application-layer, all elements in ICN have caching capability. Because one of the most interesting problems is improvement achieved by through cooperation among ICN elements in the network-layer, many researchers focus on cache placement algorithms [13, 14]. As a cache replacement algorithm taking advantage of ICN, there are also policies that make use of content popularity [15, 16].

To realize ICN, especially an ICN core element, it is required to implement a cache replacement algorithm that can be operated with severe resource constraints instead of the statistical caching algorithms for web and CDN with rich resources. The implementation cost of commonly used approaches such as LRU and LFU are also prohibitive for router hardware, as pointed out by [17, 18]. Looking back at the history of cache replacement algorithms, ICN elements need a hardware implementable approach whose complexity is comparable to that of FIFO or CLOCK. In addition to the cost, this approach should cope with access patterns specific to ICN, where the unit of caching is a fine-grained chunk rather than whole content data. To understand how to satisfy these requirements of cost and performance, we examine the knowledge of caching in computer systems and apply it to in-network caching in the following section.

3. Design Considerations of Cache Replacement Algorithm for ICN

3.1. Access Patterns of Traffic in the Network

An access pattern is the important factor to govern the performance of cache replacement algorithm. It is well known that the popularity of contents follows a Zipf-like distribution: a large number of contents requested only once or just a few times [19]. In addition, many requests generate asynchronous requests for contents, and so temporal locality of network traffic becomes relatively low.

In particular, ICN is able to identify a chunk (its default size is 4K bytes in CCNx), which enables the chunk level caching in an ICN element. Thus, we conjecture that the distribution of the “chunk popularity” would be more biased than Zipf-like distributions. In this paper, to design an appropriate cache replacement algorithm for ICN under different types of the distributions, we classify access patterns of traffic, which governs the distribution [5, 8, 7, 20], into four categories: SCAN, LOOP, COORELATED REFERENCES, and FICKLE INTEREST as follows:

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

- **FICKLE INTEREST**: rapidly changing sets of requested chunks

First, although the exact access pattern of the chunk level (i.e., network level) traffic in ICN is not known due to the lack of available ICN traffic trace, such one-time used items occupy 60% or more in the network level traffic in IP networks [21]. We conjecture that the highly aggregated network level traffic in ICN would have a large number of one-time used chunks, which correspond to SCAN access pattern.

Second, ICN is originally designed to efficiently disseminate multimedia traffic which generally occupies high network bandwidth and is requested repeatedly. Thus, we also conjecture that the chunk level traffic in ICN will have LOOP access pattern. As mentioned previously, LOOP is highly correlated to SCAN: SCAN and LOOP are generated by unpopular and popular contents, respectively.

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

For the reason above, the cache replacement algorithm for ICN should be able to deal with the access patterns described above. We here focus on the first access pattern, SCAN, in the design of the cache replacement algorithm for ICN since it is the major traffic that occupies the network bandwidth. Among the conventional cache replacement algorithms, CAR is able to efficiently deal with SCAN traffic access pattern [6] due to its dual lists which enable to distinguish popular and non-popular contents. Our proposal is based on CAR to inherit this feature.

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 the ICN element. The other is the cost that manages the table in the memory according to a cache replacement algorithm, e.g., prioritizing cached items. We call the former and the latter as computational cost and 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 cache replacement algorithm, especially when it is applied for a high speed core ICN element.

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, 17, 18]. For example, LFU has much higher overhead to keep statistics of each cached item. In LRU using double-linked-list, this cost is prohibitive due to the maintenance of double pointers to other cached items.

To reduce the computational and memory costs in conventional approaches, **CLOCK** was introduced, which has a memory link list having a shape of a clock. It searches for a cached item that needs to be replaced following a clockwise. While searching for a candidate for replacement, it refers to one bit corresponding to the candidate. When the bit is set to off, the cached item is discarded. Otherwise, the searching process keeps on going until it finds a cached item whose bit is off. Then, all bits skipped over during the searching process are set to off. Thus, **CLOCK** requires only a single bit per chunk and few repetitions of the searching process. Our proposed mechanism also adopts this mechanism in **CLOCK** to reduce the computational and memory costs.

3.3. Adaptable Parameter Tuning

Some cache replacement algorithms need to tune parameters statically and dynamically according to workloads offered to the cached items in cache. 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 workload, 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 optimal parameters, 2) invalidity of optimal parameters in the change of workload, 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 cache replacement algorithm for ICN.

4. Compact CLOCK with Adaptive Replacement (Compact CAR)

4.1. Key Ideas of the Proposed Algorithm

The algorithm we propose is based on Clock with Adaptive Replacement (CAR) [6]. CAR is known to be robust against the access patterns of SCAN, **CORRELATED REFERENCES** and **FICKLE INTEREST**.

CAR maintains two **CLOCK** lists: one is for cached items that have been accessed only once, and the other one is for the rest. **CLOCK** is a dominant algorithm in the design of page replacement in computer operating system [6, 22]. Basically, it aims to achieve some of the benefits of LRU replacement but without heavy computational and memory costs in the manipulation of LRU operation - we discuss the operation in detail in the next section.

This two-stack approach can keep frequently requested contents from being replaced by one-time requested content. In addition, CAR can dynamically adapt to the access patterns by adjusting its lengths of **CLOCK** lists as autonomously tuning parameter. Due to the adaptive parameter tuning, CAR can also function well in an unsteady environment. However, the computational cost of CAR is similar to that of LRU, which is prohibitive in the use of the algorithm for the design of high speed ICN core element.

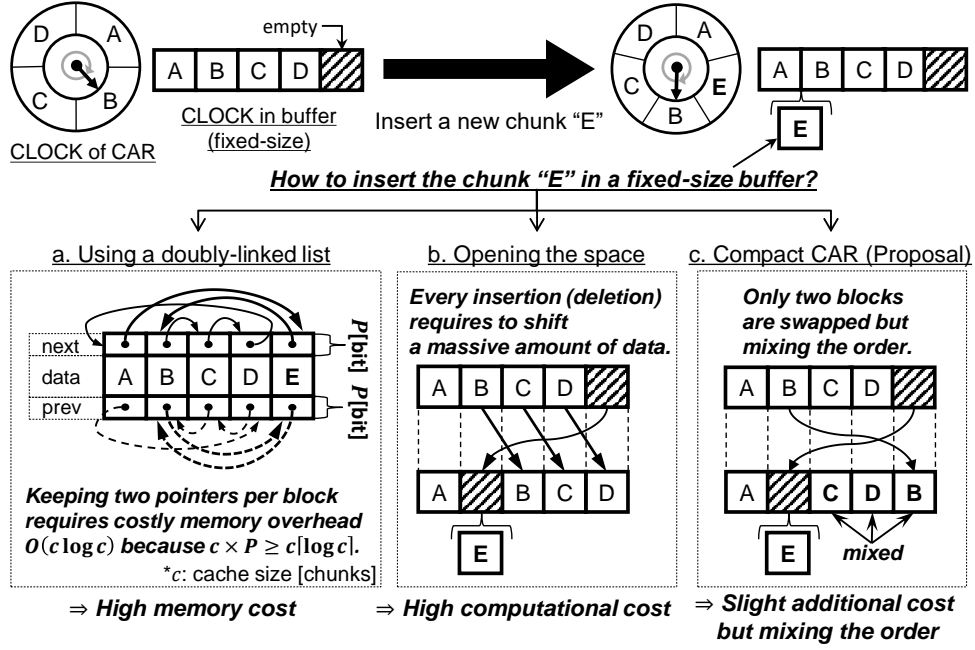


Figure 1: Illustration of Computational and Memory Costs in the Inserting Operation in the Different Data Structures

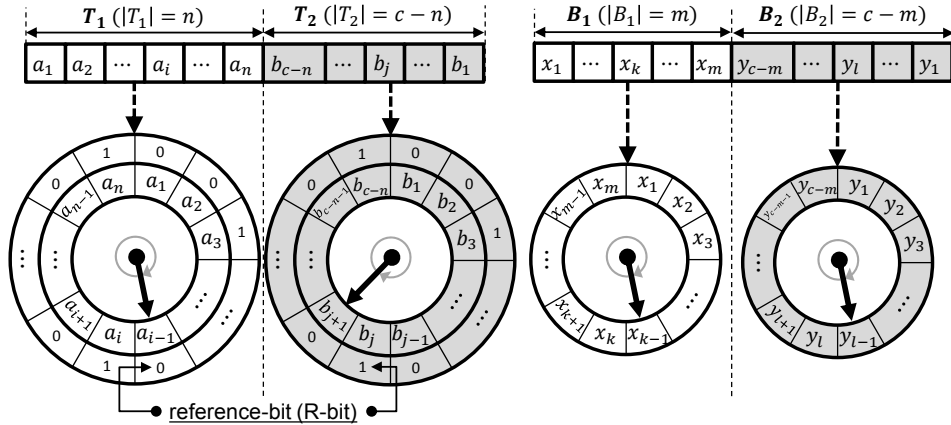


Figure 2: Data Structure of Compact CAR

4.2. Design of Compact CAR

Compact CAR is designed to further reduce the memory cost of CAR while maintaining its inherent advantages. The reason that CAR has high memory cost is that the sizes of CLOCK lists used in CAR have variable-size. Thus, Compact CAR has the two CLOCK lists in the fixed-size buffer to overcome the problem.

Figure 1 illustrates an operation of chunk insertion with different approaches from the viewpoint of computational and memory costs. First, CAR uses a doubly-linked 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 a doubly-linked list. It involves with two operational costs: com-

putational 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 block.

Second, it illustrates a case where a doubly-linked list is not used but memory blocks are shifted when a chunk is inserted. It does not require high memory cost because the order of chunks in physical memory can be used directly without creating virtual order created in the first scenario. However, this scenario demonstrates high computational cost caused by the shift of memory blocks.

Third, the operation of chunk insertion in Compact CAR is

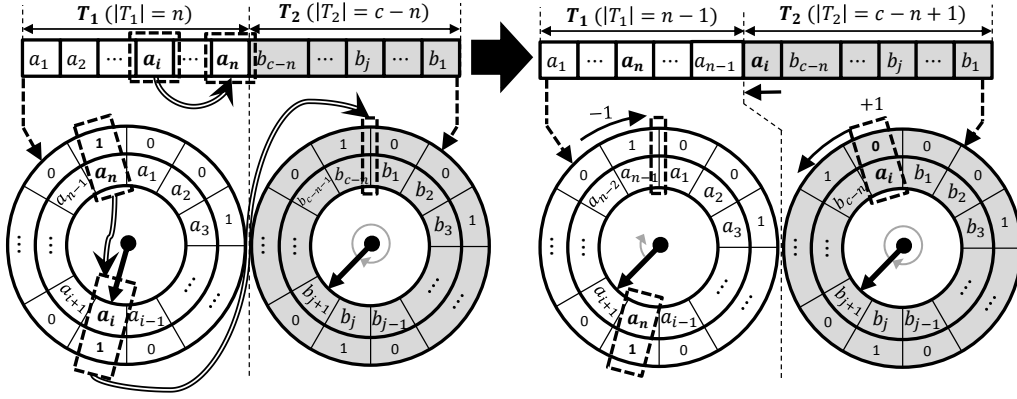


Figure 3: Example of Moving a Chunk a_i from T_1 to T_2 by Replacing the Edge Chunk a_n

illustrated. To insert a new chunk at the position of the chunk 'B', Compact CAR moves the chunk 'B' to the end. Then, the newly inserting chunk is inserted to the location. It does not use a doubly-linked list to create virtual order of chunks in the memory space and so the memory cost can be reduced dramatically. At the same time, it does not involve the shift of memory blocks simultaneously, which reduces a computational cost as well. Readers may concern that the mixed order degrades the performance but it is not that serious: we will address the issue in Section 5.

Figure 2 illustrates the data structure of Compact CAR which maintains four variable-size CLOCK lists. Let us say $T_* = T_1 \cup T_2$ and $B_* = B_1 \cup B_2$. T_* are used for caching data, and B_* remember the record of evicted chunks. Each of T_* and B_* is arranged in physically contiguous memory. The n chunks in T_1 are arranged in one side and the $(c - n)$ chunks in T_2 fill the other side. To realize the searching process described in Section 3.2, T_* and B_* have a hand as the starting position to start for searching process, and T_* assigns each chunk a reference bit (R -bit). The hand of T_1 moves rightward on the array in the figure and that of T_2 moves leftward so as to fill the free space in the direction that the hand moves (i.e., if cache is not full, there are empty blocks between T_1 and T_2). The same principle applies to B_1 and B_2 , except that they do not hold data of chunks and R -bits. Compared to the operation of CAR in the related work, Compact CAR does not require a memory space to keep pointers as discussed above. This is a reason why Compact CAR can use memory more efficiently in a compact manner than CAR.

As mentioned in Section 4.1, Compact CAR inherits advantages of two-stack approach of CAR. $L_1 = T_1 \cup B_1$ (unshaded) and $L_2 = T_2 \cup B_2$ (shaded) are assumed to be two CLOCK lists. L_1 is for contents that have been accessed only once. L_2 is for contents that have been accessed at least twice. By adjusting the target size p , which is the parameter representing the target size of T_1 ($0 < p \leq c$), the sizes of the lists $|T_1|$, $|B_1|$, $|T_2|$, and $|B_2|$ vary adaptively. A cache hit to L_1 (L_2) dynamically increases the target size for T_1 (T_2), and simultaneously decreases the target size for the other list. In other words, the size of T_1

grows when the recency of contents governs the performance of the cache replacement algorithm, whose behavior becomes similar to *LRU*. On the other hand, the size of T_2 grows when the frequency of content access governs the performance of the cache replacement algorithm, whose behavior becomes similar to *LFU*. For example, the size of T_2 grows when SCAN occurs. This feature enables Compact CAR to adaptively deal with the change of traffic access patterns.

Figure 3 gives an example of moving a chunk a_i within T_* to realize the swap operation illustrated in Fig. 1(c). Compact CAR swaps a_i with the one at the boundary between T_1 and T_2 , and then just shifts the boundary to the left hand side. This simple swap operation enables the migration of a chunk between T_1 and T_2 . However, the original CAR algorithm requires to keep a doubly linked list or to shift an enormous amount of chunks ($a_{i+1}, \dots, a_n, b_{c-n}, \dots, b_{j+1}$) leftward to insert a_i into the position pointed by the hand of T_2 as illustrated in Figure 1.

4.3. Replacement Algorithm of Compact CAR

Algorithms 1, 2, 3, and 4 show pseudocode of the cache replacement algorithm of Compact CAR. The replacement process in cache replacement starts with Algorithm 1. If an accessed chunk x is hit, then the process sets the R -bit of x and terminates (lines 2–4). If there is a cache miss, then line 5 checks whether x is in a ghost cache. If B_* contains x , the history of x is discarded and the parameter p , which is the target size for T_1 , is updated (lines 7–13). This process of tuning p makes CAR adaptive to changes in access patterns. If x is not in B_* and the ghost cache is full, then a chunk in B_* is discarded (lines 16–18). In this, i stands for the index of the list into which x is to be cached; therefore, i is set to 2 when there is a ghost hit and is set to 1 otherwise. After ensuring that there is room in the ghost cache, a chunk in T_* is discarded if T_* is full (lines 20–23). A victim chunk is selected from T_1 if the size of T_1 is not less than the target size p ; otherwise, a chunk in T_2 is replaced. Finally, x is cached at a position s_t in T_i , which has been ensured to be free (line 27).

Algorithms 2, 3, and 4 describe how to make room for a chunk or its historical information. Hand_{T_i} indicates the loca-

Algorithm 1 Compact CAR Replacement Algorithm

```

1: procedure CACHEREPLACEMENT( $x$ )  $\triangleright x$  is an accessed chunk.
2:   if  $x \in T_*$  then  $\triangleright$  cache hit
3:      $x.R\text{-bit} \leftarrow 1$ 
4:     return
5:   else if  $x \in B_*$  then  $\triangleright$  ghost hit
6:      $i \leftarrow 2$   $\triangleright$  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:       DiscardBtm( $1, x$ )
10:    else  $\triangleright x \in B_2$ 
11:       $\delta \leftarrow \max(1, \frac{|B_1|}{|B_2|}); p \leftarrow \max(0, p - \delta)$ 
12:      DiscardBtm( $2, x$ )
13:    end if
14:  else  $\triangleright$  cache miss
15:     $i \leftarrow 1$   $\triangleright$  to cache  $x$  in  $T_1$ 
16:    if Full( $L_1$ ) &  $|B_1| > 0$  then ReplaceBtm( $1$ )
17:    else if Full( $L$ ) &  $|B_2| > 0$  then ReplaceBtm( $2$ )
18:    end if
19:  end if
20:  if Full( $T_*$ ) then
21:    if  $|T_1| \geq \max(p, 1)$  then  $s_t \leftarrow \text{ReplaceTop}(1)$ 
22:    else  $s_t \leftarrow \text{ReplaceTop}(2)$ 
23:    end if
24:  else  $\triangleright T_*$  is not full.
25:     $s_t \leftarrow$  an available address in  $T_i$ 
26:  end if
27:   $T_i[s_t] \leftarrow x$   $\triangleright x$  is cached as a chunk in  $T_i$ .
28: end procedure

```

tion pointed to by the hand of T_i , and B_i is analogous. Every algorithm assures a free space at *EdgeAddr*, which is the address of the boundary of two lists, either T_1 and T_2 or B_1 and B_2 . Whenever x is not located at the boundary, it is swapped with the *EdgeChunk* of the list, which is located at the boundary. The reason to do so is that this frees an address located next to the edge address of both lists. For example, consider what happens when caching a new chunk in T_1 when a chunk in T_2 is discarded. If the cache is full and the swap is not performed, there is no free space contiguous with the area of T_1 unless the discarded chunk in T_2 is adjacent to T_1 . Otherwise, it will be necessary to swap two chunks or shift an enormous amount of chunks in order to cache the new chunk in T_1 . Thus, the swap process is essential to ensure a vacant address that is contiguous with both of the lists (not only T_1 and T_2 , but also B_1 and B_2).

The DiscardBtm procedure (shown in Algorithm 2) evicts the historical information of x in B_i when x is a ghost hit. The ReplaceBtm procedure (shown in Algorithm 3) evicts the history information pointed to by the hand of B_i when there is a cache miss and B_i is full. The ReplaceTop procedure (shown in Algorithm 4) removes the chunk pointed by the hand of T_i when the cache is full. If a chunk whose *R*-bit is set is found in T_1 , the chunk is moved to T_2 in the manner described in Fig.5. Lines 11–12 store the evicted chunk as history information without losing the contiguousness of the lists because an address next to the edge of B_i has been ensured to be free, as discussed above.

Algorithm 2 DiscardBtm() for Compact CAR

```

1: procedure DISCARDBTM( $i, x$ )
2:   Swap( $x, B_i.\text{EdgeChunk}$ )
3:   Discard  $x$  (at the edge of  $B_i$ )
4:    $\triangleright$  ensuring that an address next to the edge of  $B_i$  is free
5: end procedure

```

Algorithm 3 ReplaceBtm() for Compact CAR

```

1: procedure REPLACEBTM( $i$ )
2:   Swap( $B_i[\text{Hand}_{B_i}], B_i.\text{EdgeChunk}$ )
3:   Discard  $B_i.\text{EdgeChunk}$ 
4:   Rotate  $\text{Hand}_{B_i}$ 
5:    $\triangleright$  ensuring that an address next to the edge of  $B_i$  is free
6: end procedure

```

Algorithm 4 ReplaceTop() for Compact CAR

```

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

```

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.

First, the performance of the proposed algorithm is evaluated with various access patterns including synthetic traffic as well as real traffic trace in different types of topologies. Then, adaptability of our proposal to changing access traffic patterns is demonstrated by comparing to the same approach without tuning a parameter. Finally, computational and memory costs of the proposal are theoretically analyzed to present its efficient memory usage which is critical in the design of a high performance ICN core element.

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} . In addition, their superscript C and P , e.g., $A_{Zipf(\alpha)}^C$ and $A_{Zipf(\alpha)}^P$ represent the sizes of content and chunk, 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.

At the same time, the real traffic 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 2.

As stated in Section 3.1, the fine granularity of cacheable items 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 assume that the inter arrival time between requests to content items is similar to the one in the current Internet, which follows the Zipf distribution. However, the inter arrive time for chunks is constant according to Table 1, which is determined by the statistics of our observed real traffic. The generated requests are superimposed to simulate the aggregation of request messages in the network.

Two different topologies are used for this simulation study. One is a topology in which there is only one ICN element between clients and a server. The other is a line topology that includes ten ICN elements between them. One ICN element topology is used to demonstrate the performance of the proposed algorithm compared to those of conventional approaches including an optimal performance. On the other hand, the line topology is used to present how efficiently the proposed cache replacement algorithm works without a cooperative caching mechanism¹. In ICN, a requested content can be cached while being downloaded at any node along the downloading path, which is known as on-path caching. When all nodes along the path cache downloading contents, it eliminates the effect of a cooperative caching. In other words, the simulation results with a line topology reveal how resistant our proposed algorithm in the absence of cooperative caching mechanism.

Each cache at ICN element has same capacity c of 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.

5.2. Cache Hit Rate with Synthetic Traffic

Figures 5 depicts the cache hit rates of each cache replacement policy in a single ICN element with synthetic traffic described previously: $A_{Zipf(\alpha)}^C$ changing α from 0.6 to 1.2. Our

¹It cooperatively distributes content items in the network to improve cache hits as well as to reduce the usage of network resources.

Table 1: 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 2: Statistics of Workloads in Units of Chunks

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

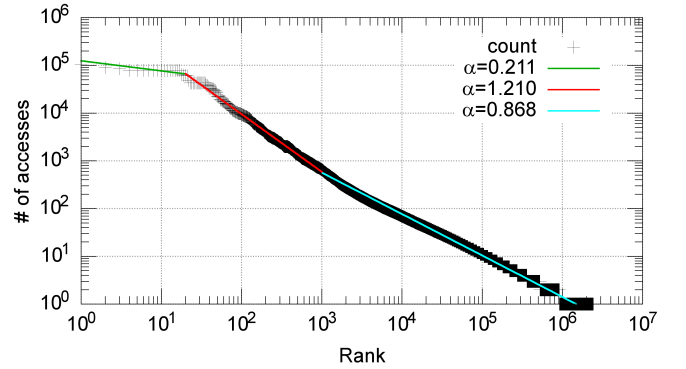


Figure 4: Popularity Distribution of Real Trace

proposal, Compact CAR, achieves a hit rate comparable to that of CAR, which is contrary to our speculation. We conjectured that the operation mixing the order in the Compact CAR would degrade its performance. 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.

In addition to the simulation using traces in units of contents, Figures 6 shows the cases when the sizes of cacheable 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, e.g., $A_{Zipf(0.6)}^{P(60KB)}$ to $A_{Zipf(0.6)}^{P(1.5KB)}$. As the value of α increases, the hit rate increases. This means that a high popularity bias results in a high hit rate. As depicted in Fig.6(d), we observe that the cache hit rate decreases substantially as the size of cacheable items becomes small, e.g., from a whole content to chunks.

5.3. Cache Hit Rate with Real Traffic Trace

Figure 7 presents the simulation results with real Video-on-demand (VoD) traffic which was collected at Osaka University.

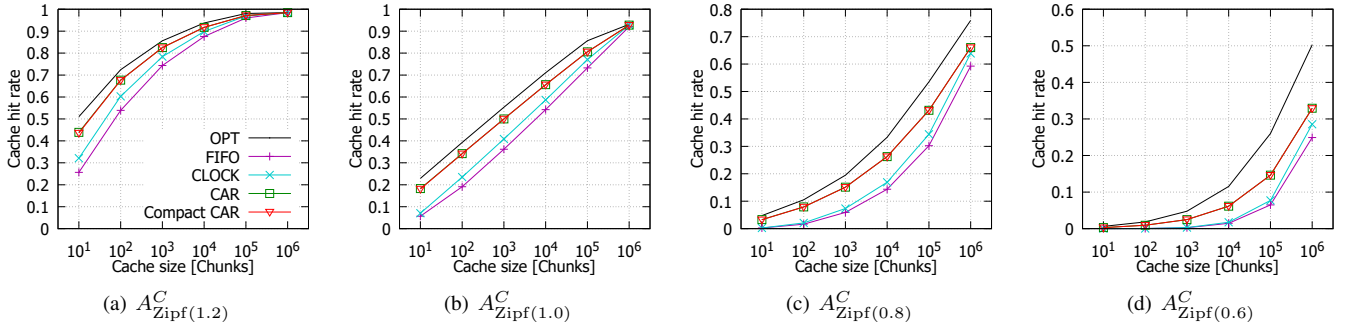


Figure 5: Results for Artificial Workloads in Units of Content

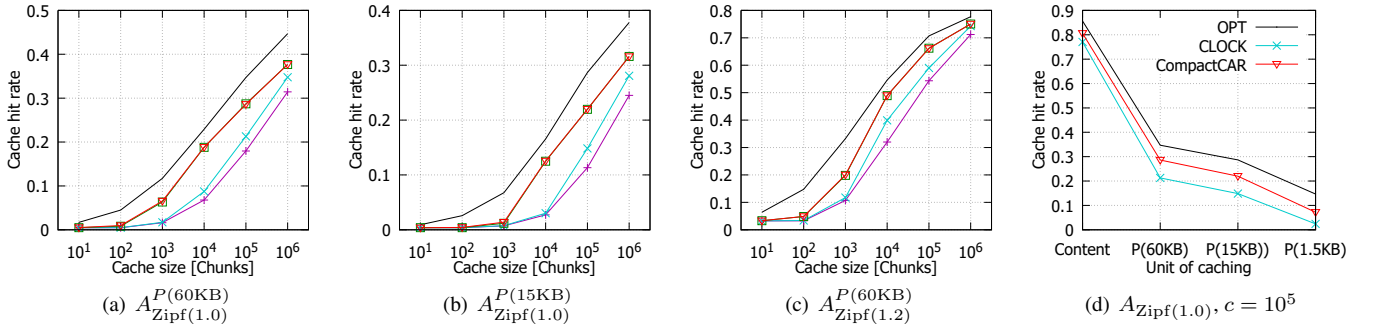


Figure 6: Results for Artificial Workloads in Units of Chunks

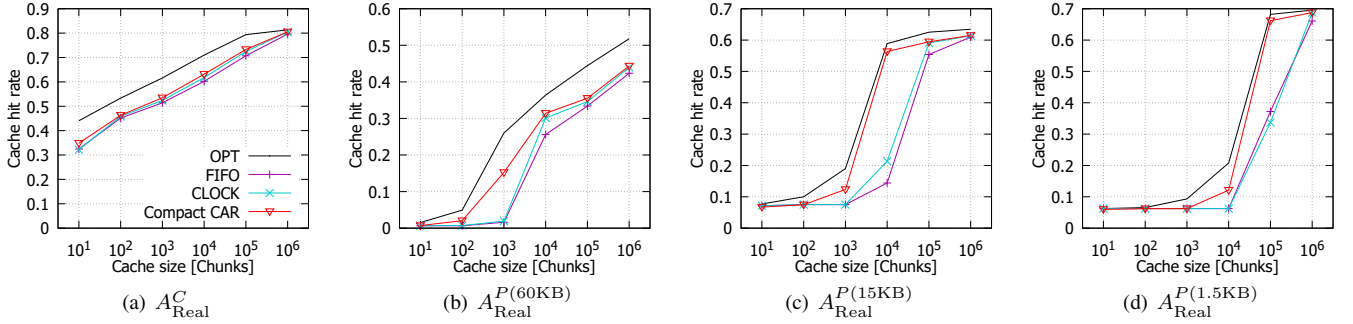


Figure 7: Results for Real Traces

Each content is segmented into small size chunks to simulate the transmission of chunks in ICN networks. The cache hit rates 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 chunks between two consecutive same chunks. For example, consider what happens

when the value of RD is larger than the size² of cache. When the first chunk in the two consecutive same chunks is cached, it has a high probability to be discarded from the cache. If this case keeps happening due to a large amount of one-time contents (e.g., SCAN and LOOP), 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 4.1, 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

²Its unit is the number of chunks

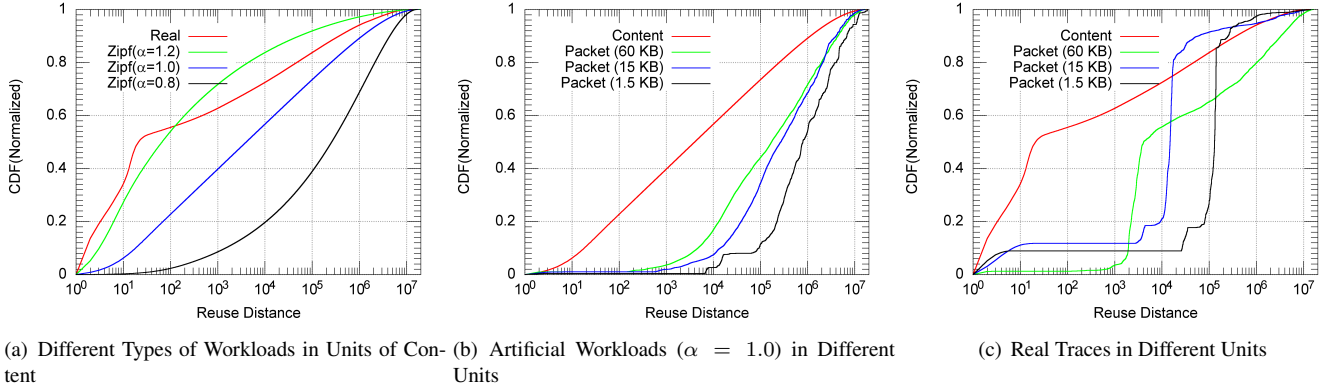


Figure 8: CDF of RD in Various Workloads

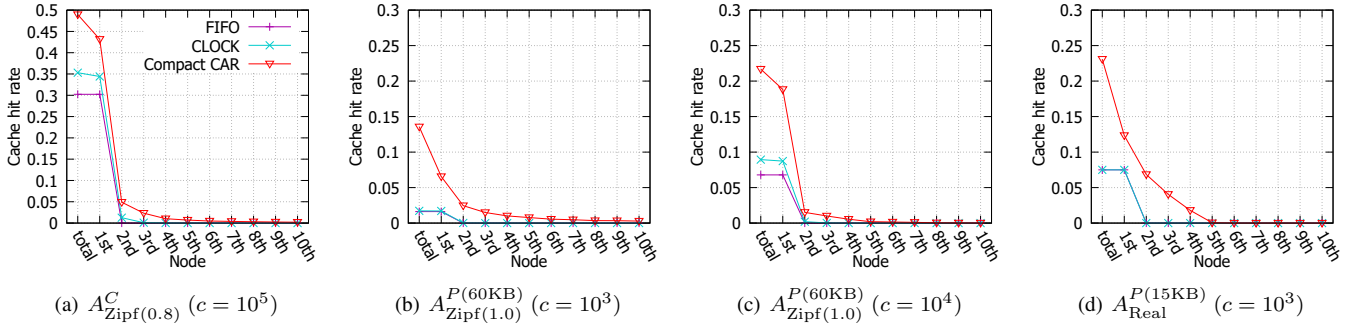


Figure 9: Results for Simulation with a Linear Topology

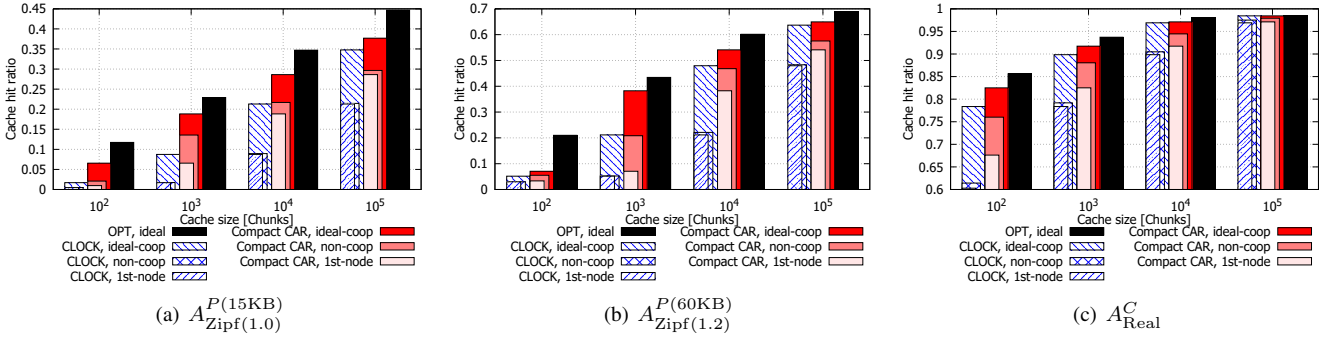


Figure 10: Comparison between non-cooperative caching and ideally-cooperative caching

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 a Line Topology

The results with a line topology are shown for the purpose of showing the lower bound of the performance where a cooperative caching mechanisms fail. 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). We can assume that one node topology represents a case where a cooperative caching

algorithm works ideally. If one node whose caching capacity 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 one node topology shows the upper bound of the performance where cooperative caching works ideally. On the other hand, the line topology represents non-cooperative caching algorithms, especially when contents being downloaded from one end to the other are cached every nodes in the line topology. In this case, the cache capacity of a whole network is considerably wasted by redundant caches. Thus, our simulation results clarify the upper and lower bound of the performance caused by coopera-

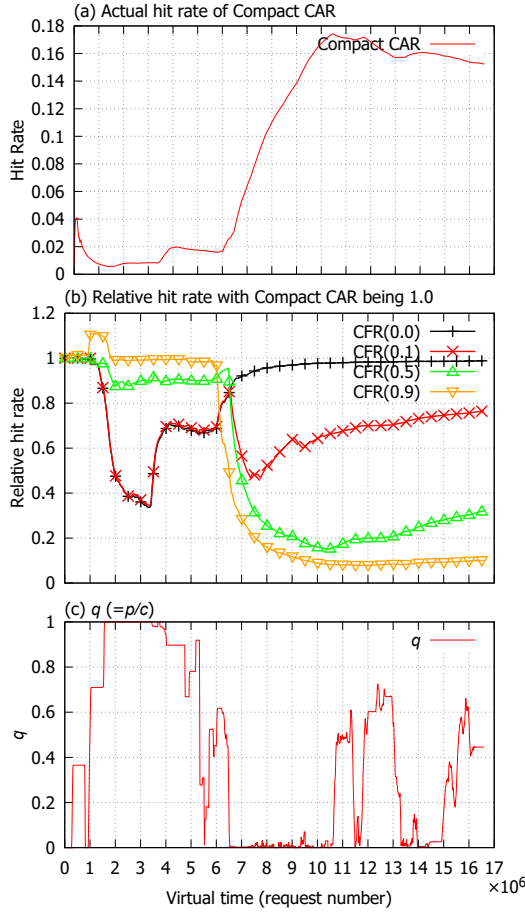


Figure 11: Dynamics of Hit Rate of CFR(q) and Adaptive Parameter q

tive caching mechanisms.

Figure 9 presents the cache hit rates of individual nodes on a line topology. 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. We denote the performance of ideally cooperative caching by “ideal-coop”, which specifies the upper bound. The result denoting “non-coop” means the total cache hit rates of nodes in a 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 a 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 algo-

rithm that is feasible and appropriate for an ICN element. 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 4.2, 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 a 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 of T_1 . The parameter p ranges from zero to the cache size c . As the value of p increases, the operational behavior of Compact CAR becomes similar to the case where 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 rates of CFR(q). The x -axis shows the virtual time t , which is equivalent to the total number of requests. The cache hit rates of CFR(q) are shown as relative value with that of Compact CAR being 1.0 in Fig. 11 (b). When $0 < t < 6 \times 10^6$, the hit rate of CFR(q) with high q tends to increase as q increases, and vice versa. The results show that Compact CAR can adaptively change the parameter. 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. 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 rates among CFR(q) are at most nearly 1.0. By contrast, the relative hit rates 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 complexities 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

Table 3: Time Complexity of Cache Replacement Algorithm's Overhead

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

Table 4: Space Complexity of Cache Replacement Algorithm's Overhead

| policies | Space Complexity | | number of history |
|----------------------------|---|-------------------|-------------------|
| | memory [bit] | order | |
| 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 |

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

Although we analyze only two cache replacement algorithms: CAR and Compact CAR, Table. 4 and 4 summarize the analytical results of space and time complexities of not only the two of them 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 other than CAR and Compact CAR are presented in the Appendix A.

5.6.1. Space Complexity

First, we analyze the space complexity of CAR and Compact CAR. Compact CAR maintains four CLOCKS shown in Fig. 2. Our simple swapping renders Compact CAR free from the additional costs of memory or process for maintaining the order of sweeping the list. Furthermore, B_1 and B_2 do not need R -bit and the total length of the other two CLOCKS, T_1 and T_2 , is n . Thus, Compact CAR costs $(n + 9P)$ bits for two normal CLOCKS whose total length is n , two CLOCKS without a R -bit, four information of the size of the lists, and a parameter of a target size.

On the other hand, CAR has two variable-size CLOCK lists and two LRU lists. The variable-size CLOCK 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 list. Because the approach illustrated in Fig. 1(b) imposes no additional memory cost, we focus on CAR implemented with a doubly-linked list. Space complexity of two CLOCK lists and two LRU lists is comparable to that of four doubly-linked lists whose total maximum 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

Second, we elaborate the time complexity of them. 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 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.

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 L_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 $0 \leq \gamma \leq \text{zetaeta} \leq 1$. Time complexity of Compact CAR can be calculated in the same way as CAR. Time complexity at a cache hit is $t_w + \delta$, and worst-case and average-case complexity at a cache miss are $O(n)$ and $O(\frac{1}{1-\beta})$, respectively.

6. Discussion on the Implementation of Compact CAR for High Performance ICN Core Element

6.1. Computational Overhead of Variants of CLOCK

In the previous section, 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 one. 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 space complexity of CAR can be reduced by using a stack instead of a doubly-linked list, the implementation with a stack makes time complexity prohibitive as illustrated in Fig. 1(b). In conclusion, the computational and memory costs of Compact CAR are acceptable in the design of high performance ICN core element.

6.2. Feasibility of Hardware Implementation

The throughput and capacity of a cache are the most serious obstacles to realize in-network caching. Assuming 10 Gbps of traffic and with 64-byte data packets, a single-line card would not have throughput of approximately 20 million accesses

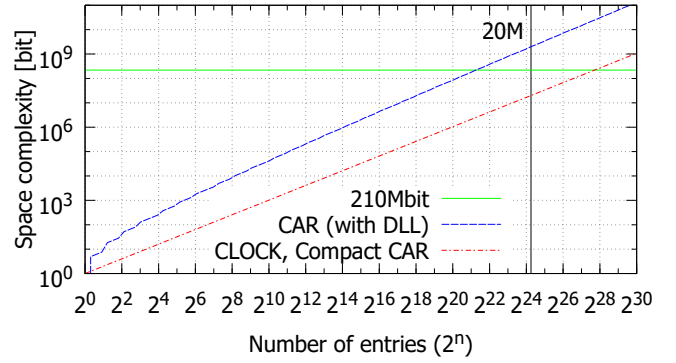


Figure 12: Space Complexities of CAR and Our Proposal (Compact CAR)

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; however, a cache decision policy is necessary to enable ten-fold improvement.

Figure 12 shows a memory overhead of CAR and Compact CAR. As explained in 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 210 Mbit [24]. On the other hand, Compact CAR requires a memory overhead of one bit per chunk. Compact CAR consumes 20 Mbit; therefore the cost of Compact CAR is feasible.

If SRAM access time is 0.45 ns [24], the router can handle the traffic of the four line cards quickly enough to keep up, even in the case of a cache miss causing several rotations of the hand and several swapping processes. 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

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 content delivery network previously. This paper argued that the conventional cache replacement algorithms cannot be directly applied to the design of a high performance ICN core element.

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 element. Compact CAR outperforms compared to conventional cache replacement algorithms in terms of cache hit rates and reduction of memory usage in the design of ICN element. In detail, the proposed algorithm can achieve the same cache hit rates with only one-tenth of memory usages that simple conventional algorithms use. 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 ICN core element becomes critical. We believe that the proposed cache replacement algorithm plays a key role in the design of such a high performance ICN core element in near future.

Acknowledgment

This work was supported by the Strategic Information and Communications R&D Promotion Programme (SCOPE) of the Ministry of Internal Affairs and Communications, Japan.

References

- [1] V. Jacobson, D. K. Smetters, J. D. Thornton, M. F. Plass, N. H. Briggs, R. L. Braynard, Networking named content, in: Proceedings of the ACM CoNEXT 2009, 2009, pp. 1–12.
- [2] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J. D. Thornton, D. K. Smetters, B. Zhang, G. Tsudik, K. Claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, E. Yeh, Named data networking (NDN) project (October 2010).
URL <http://named-data.net/techreport/TR001ndn-proj.pdf>
- [3] T. Levä, J. Gonçalves, R. J. Ferreira, 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] N. Fotiou, P. Nikander, D. Trossen, G. C. Polyzos, Developing information networking further: From PSIRP to PURSUIT, in: Proceedings of the 7th International ICST Conference on Broadband Communications, Networks, and Systems, 2010, pp. 1–13.
- [5] N. Megiddo, D. S. Modha, ARC: a self-tuning, low overhead replacement cache, in: Proceedings of the 2Nd USENIX Conference on File and Storage Technologies, 2003, pp. 115–130.
- [6] S. Bansal, D. S. Modha, CAR: Clock with adaptive replacement, in: Proceedings of the 3rd USENIX Conference on File and Storage Technologies, 2004, pp. 187–200.
- [7] T. Johnson, D. Shasha, 2Q: a low overhead high performance buffer management replacement algorithm, in: Proceedings of the 20th International Conference on Very Large Data Bases, 1994, pp. 439–450.
- [8] S. Jiang, X. Zhang, Making LRU friendly to weak locality workloads: a novel replacement algorithm to improve buffer cache performance, IEEE Transactions on Computers 54 (8) (2005) 939–952.
- [9] F. J. Corbato, A paging experiment with the Multics system, Tech. rep. (May 1968).
- [10] J. Wang, A survey of web caching schemes for the Internet, ACM SIGCOMM Computer Communication Review 29 (5) (1999) 36–46.
- [11] K.-Y. Wong, Web cache replacement policies: a pragmatic approach, IEEE Network 20 (1) (2006) 28–34.
- [12] A.-M. K. Pathan, R. Buyya, A taxonomy and survey of content delivery networks, Tech. rep., University of Melbourne Grid Computing and Distributed Systems Laboratory (February 2007).
- [13] G. Zhang, Y. Li, T. Lin, Caching in information centric networking: A survey, Computer Networks 57 (16) (2013) 3128–3141.
- [14] M. Zhang, H. Luo, H. Zhang, A survey of caching mechanisms in Information-Centric Networking, IEEE Communications Surveys Tutorials 17 (3) (2015) 1473–1499.
- [15] J. Ran, N. Lv, D. Zhang, Y. Ma, Z. Xie, On performance of cache policies in named data networking, in: Proceedings of the International Conference on Advanced Computer Science and Electronics Information 2013, 2013, pp. 668–671.
- [16] L. Wang, S. Bayhan, J. Kangasharju, Optimal chunking and partial caching in information-centric networks, Computer Communications 61 (1) (2015) 48–57.
- [17] S. Arianfar, P. Nikander, J. Ott, Packet-level caching for information-centric networking, Tech. rep., Finnish ICT SHOK (June 2010).
- [18] D. Rossi, G. Rossini, Caching performance of content centric networks under multi-path routing (and more), Tech. rep., Telecom ParisTech (July 2011).
- [19] L. Breslau, P. Cao, L. Fan, G. Phillips, S. Shenker, Web caching and Zipf-like distributions: evidence and implications, in: Proceedings of IEEE INFOCOM'99, Vol. 1, 1999, pp. 126–134.
- [20] A. Jaleel, K. B. Theobald, S. C. Steely, Jr., J. Emer, High performance cache replacement using re-reference interval prediction (RRIP), ACM SIGARCH Computer Architecture News 38 (3) (2010) 60–71.
- [21] F. Guillemin, B. Kauffmann, S. Moteau, A. Simonian, Experimental analysis of caching efficiency for YouTube traffic in an ISP network, in: Proceedings of the 25th International Teletraffic Congress, 2013, pp. 1–9.
- [22] S. Jiang, F. Chen, X. Zhang, CLOCK-Pro: An effective improvement of the CLOCK replacement, in: Proceedings of the USENIX 2005, 2005, pp. 323–336.
- [23] C. Fricker, P. Robert, J. Roberts, N. Sbihi, Impact of traffic mix on caching performance in a content-centric network, in: Proceedings of the IEEE Conference on Computer Communications 2012, 2012, pp. 310–315.
- [24] D. Perino, M. Varvello, A reality check for Content Centric Networking, in: Proceedings of the ACM SIGCOMM workshop on Information-centric networking, 2011, pp. 44–49.
- [25] A. Ooka, S. Ata, K. Inoue, M. Murata, High-speed design of conflict-less name lookup and efficient selective cache on CCN router, IEICE Transactions on Communications E98-B (04) (2015) 607–620.

Appendix A. Time and Space Complexity of the Remaining Policies

We analyze time and space complexity of policies which are skipped in Section 5.6. The complexity is calculated based on 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, therefore, 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 ghost caches since it should be compared with the amount of memory required for cache data rather than control information.

In addition to the notations in Section 5.6, we define the following notations and variables. m is the number of ghost cache entries in LIRS. Statistical policies assign C -bit information (as a counter used in LFU) to every entry.

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

Appendix A.2. Complexity of LRU

LRU_{DLL}. 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 us denote an entry by $e_i (i = 1, 2, \dots, n)$, where smaller i means that the entry is more recent, and its pointers that point previous and next entries by p_i^{prev} and p_i^{next} , 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 evicted 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. LRU_S introduces no additional memory cost because its data structure maintains all control information needed to perform the algorithm. LRU entry, which is evicted when a

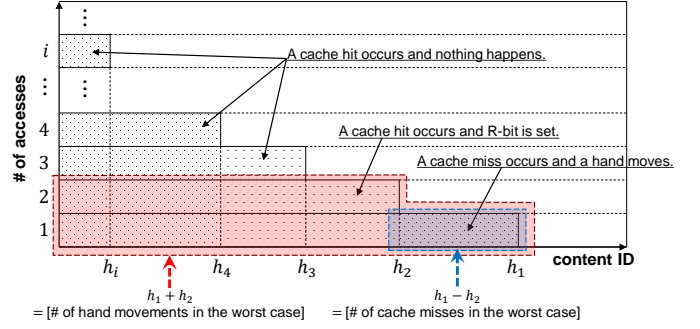


Figure A.13: Description of calculating ω of CLOCK

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 4.2. 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 element because of an excessive amount of entries.

LRU_C. LRU_C assigns a C -bit counter to each entry. In addition, a C -bit counter is necessary to remember the total number of accesses. Thus, LRU_C imposes $(Cn + C)$ -bit space complexity.

Time complexity at a cache hit is $O(1)$ in accordance with processes updating a counter and writing the value at a new entry. 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.

Appendix A.3. Complexity of CLOCK

To store n R-bits and a position located by a clock hand, space complexity of CLOCK is $(n + P)$ bits. 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. A.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 max-

imized 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. A.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. A.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 $0 \leq \beta \leq 1$.

Appendix 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.

Appendix A.5. Complexity of ARC

ARC has two LRUs and each LRU contains n entries, therefore, the space complexity of ARC implemented with LRU_{DLL} is more than 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. Time complexity is $O(1)$ as well as LRU_{DLL} because there is no repetition in ARC 's algorithm.

Appendix A.6. Complexity of $LIRS$

$LIRS$ uses two LRUs 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 ghost cache. 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 a scan and a loop, is theoretically unlimited.

Time complexity can grow significantly since there is an operation called stack pruning in $LIRS$. Stack pruning removes cold chunks that have not been accesses for a very long time including ghost caches. In the worst case, m ghost caches are removed by only a single stack pruning operation, therefore, worst-case complexity is $O(m)$. Especially, if there is a long loop or scan, this overhead becomes extraordinarily large according to the length of the access pattern.

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.