PAPER Special Section on Special Section on Internet Technologies to Accelerate Smart Society

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

Atsushi OOKA[†a)], ***Nonmember***, **Suyong EUM**[†b)], **Shingo ATA**[††c)], ***Members***, ***and*** **Masayuki MURATA**[†d)], ***Fellow***

SUMMARY Information-centric networking (ICN) has received increasing attention from all over the world. The novel aspects of ICN (e.g., the combination of caching, multicasting, and aggregating requests) is based on names that act as addresses for content. The communication with name has the potential to cope with the growing and complicating Internet technology, for example, Internet of Things, cloud computing, and a smart society. To realize ICN, router hardware must implement an innovative cache replacement algorithm that offers performance far superior to a simple policy-based algorithm while still operating with feasible computational and memory overhead. However, most previous studies on cache replacement policies in ICN have proposed policies that are too blunt to achieve significant performance improvement, such as first-in first-out (popularly, FIFO) and random policies, or impractical policies in a resource-restricted environment, such as least recently used (LRU). Thus, we propose CLOCK-Pro Using Switching Hash-tables (CUSH) as the suitable policy for network caching. CUSH can identify and keep popular content worth caching in a network environment. CUSH also employs CLOCK and hash-tables, which are low-overhead data structure, to satisfy the cost requirement. We numerically evaluate our proposed approach, showing that our proposal can achieve cache hits against the traffic traces that simple conventional algorithms hardly cause any hits.

key words: *Information-centric networking, Content-centric networking, Caching, Cache replacement algorithm*

1. Introduction

Information-centric networking (ICN) has been proposed as a measure for overcoming the limitations of current Internet architecture. ICN provides inherent in-network caching in the framework of a novel networking architecture based on *name* address that is assigned to content rather than a device [1]. In the Internet, it is becoming more and more difficult to assign and manage the location of devices in the growing and complicating Internet technology, for example, Internet of Things, cloud computing, and a smart society, where the devices explosively increases and the connectivity is easily lost. However, network users mainly want content in network and do not want to know the location of who provides it at the risk of increasing such difficulties. In ICN, the name-based communication releases a network device from the assignment and management of the location, and enhances the availability of content.

[†]The authors are with the Graduate School of Information Science and Technology, Osaka University, Suita-shi, Osaka, 565-0871, Japan

[††]The author is with the Graduate School of Engineering, Osaka City University, Sumiyoshi-ku, Osaka-shi, Osaka 558-8585, Japan

a) E-mail: a-ooka@ist.osaka-u.ac.jp

b) E-mail: suyong@ist.osaka-u.ac.jp

c) E-mail: ata@info.eng.osaka-cu.ac.jp

d) E-mail: murata@ist.osaka-u.ac.jp

Many challenges must be resolved to realize ICN, which is a clean-slate network. In particular, most researchers focus on in-network caching mechanisms because such mechanisms can reduce network traffic volume efficiently. Broadly, the popularity of Internet traffic approximately follows a Zipf distribution, where a large amount of packets carrying redundant data are transferred. In [2], content requested more than once occupy 50% or more of the volume of network traffic, with the value depending on the observation period. The potential of caching is explored in terms of which content should be cached at the router and how to choose a "victim" chunk. The former and the latter are known as content placement and cache replacement problems, respectively.

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

We propose a cache replacement algorithm that is suitable for network traffic, achieves a high cache-hit rate, and is scalable enough to address the challenge of implementing a cache mechanism in resource-restricted hardware, inspired by CLOCK-Pro [3]. Our proposed algorithm, called CLOCK-Pro Using Switching Hash-tables (CUSH), satisfies the performance requirements by employing two strategies: two types of chunks and ghost caches. Classifying chunks enables a cache to detect and hold popular content. Ghost caches are the records of content discarded from the cache. By storing the historical information of accesses rather than the data of content, CUSH can adapt to the variety and dynamism of network traffic access patterns while avoiding an excessive memory cost.

CUSH also satisfies the cost requirements by adopting the low-overhead data structures: a CLOCK list and

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

This paper is organized as follows. We summarize various cache mechanisms ranging from computer systems to in-network caching in Section II and the following Section III uses the knowledge of them to reveal the design considerations of cache replacement algorithm for a network environment. In Section IV, we give the description of CUSH. The simulation results in Section V show that our proposed approach, which can be assumed as a low-overhead variant of CLOCK-Pro, is comparable to CLOCK-Pro and outperforms simple policies. In addition, our proposal can achieve cache hits against the traffic traces where simple conventional algorithms cannot cause any hits. The section also clearly shows that CUSH is low-overhead by comparing time and space complexity of cache replacement algorithms. Finally, we conclude this article in Section 6.

2. Related Works

Because cache resources are limited, practical cache replacement algorithms are necessary to keep popular content and remove rarely used one. There are a considerable number of cache replacement algorithms, ranging from those used in software (e.g., database and web applications) to those available in hardware (e.g., CPU and I/O buffers). In this section, we review several cache replacement algorithms that have been carried out in the different context to understand the requirements of in-network caching.

Replacement algorithms are developed originally for the purpose of paging in the computer system [4], [5]. The bottleneck of the systems is the latency of fetching pages from slow auxiliary memory to fast cache memory. On the one hand, the hardware cache such as CPU commonly used First-in, first-out (FIFO) and Not Recently Used (NRU) to reduce the memory and computational costs because of the hardly limited resources. On the other hand, the software cache such as virtual memory of OS commonly adopts LRU and LFU, which are costly to maintain a data structure or/and statistical information (i.e., the number of references to a page).

Then, researchers have uncovered access patterns that degrade the performance of the algorithms. To overcome the problematic access patterns, many variants of LRU and LFU are developed. 2Q [6], ARC [4] and LIRS [7] improve the performance by exploiting the advantages of LRU and LFU while their time and space complexities are comparable to that of LRU. They also hold *ghost caches*, which are the records of evicted chunks rather than the data of the chunk to adapt to the variety and dynamism of access patterns. In contrast to them, CLOCK [8] reduces the complexity of LRU by approximating its behavior with a fixed circular buffer while keeping the performance. The complexity of CLOCK is comparable to that of NRU which has a low computational cost. CLOCK-Pro [3] combines CLOCK with LIRS to achieve both performance improvement and cost reduction.

After the emergence of the web services, web-cache and CDN-cache are researched intensively to improve the performance of them in terms of bottleneck, latency, overload and robustness [9]–[11]. Because the resource constraints of them are more moderate than those of computer systems, the cache replacement algorithms in a web and a CDN utilize statistical information including not only recency and frequency but also several others including size, latency, and URI [10]. However, the improvement was slight or specific to particular environments in spite of an abundance of caching algorithms [11].

In recent years, ICN has revived research on caching algorithms because ICN provides inherent in-network caching feature. Unlike web- and CDN-cache employed in the application-layer, all devices in ICN have caching capability. Because one of the most interesting problems is improvement achieved by through cooperation among ICN routers in the network-layer, many researchers focus on cache placement algorithms [12], [13]. As a cache replacement algorithm taking advantage of ICN, there are also policies that make use of content popularity [14], [15]. Previous papers on caching use only LRU [16], [17] or claim that the effect on performance of cache replacement is minimal [18]. 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 [14], [15]. Cache replcement policy based on content popularity (CCP) [14] can significantly decrease the server load and increase cache hit rate compared to that of LRU and LFU. The work in [15] analyzed the effects of chunking and proposed Highest cost item caching (HECTIC), which uses a utility-based replacement algorithm and outperforms existing polices including LRU. Their statistical approaches are too expensive to be employed in an ICN core router due to computational and memory costs. However, we propose a low-overhead cache replacement policy that outperforms LRU-based and simple replacement policies by coping with access patterns specific to ICN. Compact CAR [19] is low-overhead and copes with a part of traffic access patterns; however, the policy is weak to LOOP and assumes that the memory cost of the lookup table for ghost caches is acceptable, which are discussed in the following section.

To realize ICN, especially an ICN core router, it is required to implement a cache replacement algorithm that can be operated with severe resource constraints instead of the statistical caching algorithms for web and CDN with rich resources. The implementation cost of commonly used ap-

proaches such as LRU and LFU are also prohibitive for router hardware, as pointed out by [18], [20]. Looking back at the history of cache replacement algorithms, ICN routers need a hardware implementable approach whose complexity is comparable to that of FIFO or CLOCK. In addition to the cost, this approach should cope with access patterns specific to ICN, where the unit of caching is a fine-grained chunk rather than whole content data. To understand how to satisfy these requirements of cost and performance, we examine the knowledge of caching in computer systems and apply it to in-network caching in the following section.

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 content follows a Zipf-like distribution: a large number of content requested only once or just a few times [21]. In addition, many requests generate asynchronous requests for content, and so the temporal locality of network traffic becomes relatively low.

In particular, ICN is able to identify a chunk (its default size is 4K bytes in CCNx), which enables the chunk level caching in an ICN router. Thus, we conjecture that the distribution of the "chunk popularity" would be more biased than Zipf-like distributions. In this 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 [4], [6], [7], [22], into four categories: SCAN, LOOP, COOREALTED-REFERENCES, and FICKLE-INTEREST as follows:

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

First, although the exact access pattern of the chunk level (i.e., network level) traffic in ICN is not known due to the lack of available ICN traffic trace, such one-time used items occupy 60% or more in the network level traffic in IP networks [2]. We conjecture that the highly aggregated network level traffic in ICN would have a large number of one-time used chunks, which correspond to SCAN access pattern. SCAN makes the performance of an LRU-based approach much poor because such unpopular content occupies the whole cache.

Second, ICN is originally designed to efficiently disseminate multimedia traffic which generally occupies high network bandwidth and is requested repeatedly. Thus, we also conjecture that the chunk level traffic in ICN will have LOOP access pattern. As mentioned previously, LOOP is highly correlated to SCAN: SCAN and LOOP are generated by unpopular and popular content, respectively. Each chunk in LOOP is removed from a cache before being accessed again. LOOP has adverse effect on a LFU-based approach as well as LRU because all chunk in LOOP have the same recency and frequency.

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

For the reason above, the cache replacement algorithm for ICN should be able to deal with the access patterns described above. We here focus on SCAN and LOOP in the design of the cache replacement algorithm for ICN since it is the major traffic that occupies the network bandwidth. Among the conventional cache replacement algorithms, CLOCK-Pro is able to efficiently deal with the traffic access patterns [3] due to its strategy based on inter-reference recency (IRR) which enables to cope with SCAN by distinguishing popular and non-popular content and furthermore keep the appropriate number of popular content for the traffics with LOOP. Our proposal is based on CLOCK-Pro to inherit this feature.

3.2 Computational and Memory Cost

The cache replacement algorithm in an ICN router must satisfy the requirements for consider computational and memory costs: the former is the cost that updates the table holding the information of cached items in the ICN router, and the other is the cost that manages the table in the memory according to a cache replacement algorithm, e.g., prioritizing cached items. The computational cost includes insertion of a new caching item into the table, deletion of an existing cached item from the table, moving the location of cached items in the memory, and updating relevant information in the caching table. The memory cost increases as the number of cached items increases due to the increase of control information for the maintenance of the table [3], [5], [18], [20]. For example, LFU has much higher overhead to keep statistics of each cached item. In LRU using double-linked-list, this cost is prohibitive due to the maintenance of double pointers to other cached items.

CLOCK can satisfy the requirements. CLOCK has a memory link list having a shape of a clock and assigns one-bit to each entry in the list. CLOCK searches for an entry containing a cached item that needs to be replaced following a clockwise. CLOCK can keep recently accessed content because CLOCK judges whether a found entry should be removed by the assigned bit, which is set to on whenever the cache is accesses. When the bit is on, CLOCK

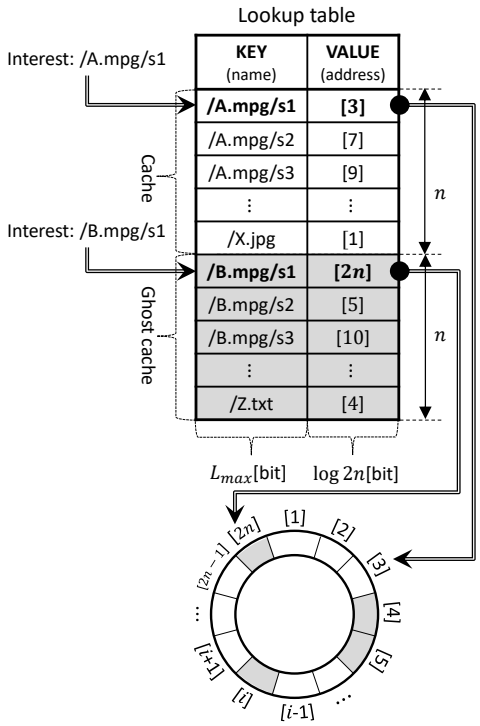


Fig. 1: Ideal Lookup Table for the Cache Replacement Policy using Ghost Caches (e.g., CLOCK-Pro)

unsets the bit and continues the search process; otherwise, CLOCK removes the cache. Thus, CLOCK requires only a single bit per chunk and few repetitions of the searching process. Our proposed mechanism also adopts this mechanism in CLOCK to reduce the computational and memory costs.

3.3 Overhead on Lookup Table caused by Ghost Caches

Some cache replacement algorithms such as LIRS and CLOCK-Pro hold ghost caches, which are historical record of cached content items rather than cached data. Although the ghost caches are essential to distinguish the chunks worth caching from chunks in SCAN and LOOP, the ghost caches impose high memory overhead on a cache lookup table. For example, consider the lookup table for a cache replacement algorithm using ghost caches as depicted in Fig. 1. In the cache in ICN, the lookup table maps the name of a chunk to the address of the entry for the chunk in the data structure of a cache policy. In the figure, we assume CLOCK-Pro as the cache replacement policy, which can cache n chunks and remember n ghost caches. Entries for the cached chunks (unshaded) and the ghost caches (shaded) are stored in the CLOCK lists shown in the bottom of the figure. In this example, the chunk "A.mpg/s1" is cached and is mapped to the entry at the address 3 in the CLOCK list. The chunk "/B.mpg/s1" is not in the cache memory but only its historical record is stored in the entry at the address $2n$ as a ghost cache.

Then, let us discuss the memory cost of a lookup table to keep ghost caches. Assuming the lookup table is implemented as a simple key-value store, it needs to store a name as a key and an address as a value. Assuming the maximum length of a name is L_{max}[bit], the lookup table totally needs $2n(L_{max}+\log 2n)$[bit], where $n(L_{max}+\log 2n)$[bit] are consumed to keep n ghost caches. Since L_{max} can be at most 512 Kbit according to the specification of CCNx, the memory cost easily becomes prohibitive. Of course, previous literature on ICN reduces the cost by proposing a low-overhead lookup table such as a tree-based implementation [23]; however, it is expensive to double the number of entries in a lookup table for ghost caches. To solve this challenge retaining benefits of ghost caches, our proposal addresses the challenge of minimizing the additional cost of ghost caches as discussed in Section 4.3.

4. CLOCK-Pro Using Switching Hash-table (CUSH)

4.1 Data Structure of CUSH

CUSH has two component parts as shown in Fig. 2. One is the circular buffer which works as a modified CLOCK list, whose entry holds information to point to an actually cached chunk. The CLOCK list has two types of entries: *hot* entries and *cold* entries. H and C in the figure denote a hot entry and a cold entry, respectively. The hot entry points to a frequently accessed chunk, and the cold entry points to the rest. Precisely, they are classified based on IRR as explained later. The other component is the two hash-tables for ghost caches, which act as "the loser bracket" providing an opportunity for discarded chunks to be re-cached.

The CLOCK list assigns two bits to each entry: a bit that indicates whether it is hot or cold (H-bit), and a reference bit (R-bit). The R-bit indicates whether the entry has been accessed. H-bit is set to "1" when the entry is hot; otherwise, "0". R-bit is set to "1" when the chunk is accessed. The entry with a check mark in Fig. 2 indicates its R-bit is set to "1". R-bit is set to "0" by using the two hands: $\mathrm{HAND}_{\mathrm{cold}}$ and $\mathrm{HAND}_{\mathrm{hot}}$. $\mathrm{HAND}_{\mathrm{cold}}$ is used to discard a cold entry when cache replacement occurs. $\mathrm{HAND}_{\mathrm{hot}}$ is used to change a hot entry to a cold entry. Both hands remember the position to start the process. When the process is needed, the hand rotate clockwise to search a entry whose R-bit is "0", and rotates ignoring it. When $\mathrm{HAND}_{\mathrm{cold}}$ ($\mathrm{HAND}_{\mathrm{hot}}$) encounters a cold (hot) entry whose R-bit is "1", the hand resets its R-bit to "0" and continues to rotate ignoring the entry. In the process of $\mathrm{HAND}_{\mathrm{cold}}$, $\mathrm{HAND}_{\mathrm{cold}}$ also sets its H-bit to "1". In addition, $\mathrm{HAND}_{\mathrm{cold}}$ ($\mathrm{HAND}_{\mathrm{hot}}$) ignores a hot (cold) chunk.

The two hash-tables store ghost caches, which are the information of chunks discarded from the CLOCK list recently. The information of a ghost cache is recorded as the flag bit in the hash-tables. In detail, when a chunk whose name is /a.txt has been removed from the CLOCK list, the flag bit of an entry at the address of $H(/a.txt)$ is set to "1", where $H(name)$ is the mapping function that maps a name in ICN to the address of a flag bit in hash-tables.

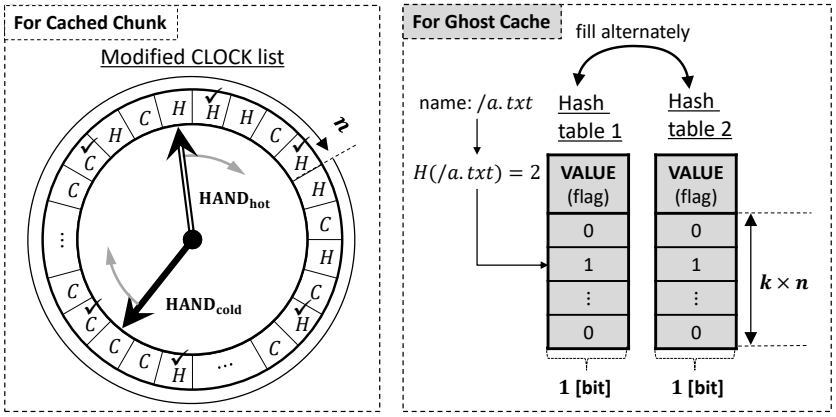


Fig. 2: Data Structure of CUSH

When the chunk is accessed, the chunk is cached as hot if its corresponding flag bit is "1"; otherwise, it is cached as cold. CUSH has two hash-tables to fill (and clear) them alternately. It is undesirable to have only one hash-table and remove all ghost caches when needed because the chance of the access to a ghost cache (i.e., the chance to detect popular chunks) is lost shortly after the clear operation.

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

4.2 Operation of CUSH

Here, we explain the operation of CUSH. There are two cases when a new request arrives: (1) the requested chunk is in the cache memory or (2) the chunk is not in the cache memory. In the case (1), the entry of the requested chunk is in the CLOCK list. If the R-bit of the corresponding entry is "0", it changes to "1"; otherwise, it remains "1". In the case (2), CUSH firstly retrieves the requested chunk from the source of the chunk. Then, CUSH verifies whether the requested chunk has been cached previously or not by checking the hash tables. If the flag at the address of $H(name)$ is "1", it means a *ghost hit*, that is, the chunk with the name has been cached previously but the chunk does not exist in the cache memory. Thus, when the chunk is cached, its entry is added to the CLOCK lists as a hot entry. If the flag bits in the hash tables are "0", the requested chunk has not been cached recently. Thus, a new cold entry for the chunk is added to the CLOCK list. We elaborate these processes in detail in the following.

Figures 3 and 4 illustrates the operation classifying the entry into hot or cold. The figures show the recently accessed requests to a router until now, which are arranged in

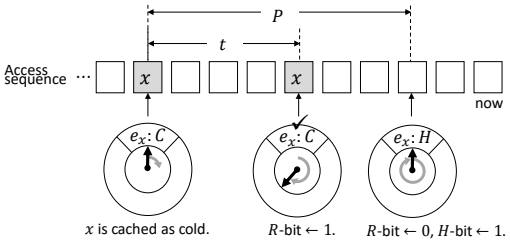


Fig. 3: The Operation of CUSH in the Case where e_x Becomes Hot

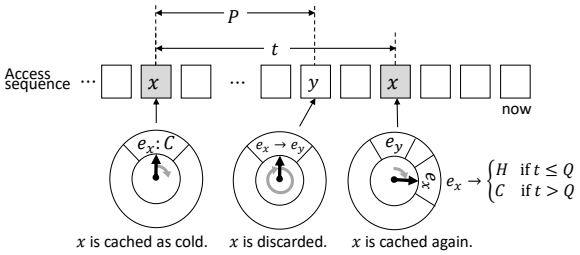


Fig. 4: The Operation of CUSH in the Case where e_x Becomes Cold

the order of the accessed time. In this example, we focus on a certain chunk "x" and illustrate how to classify its entry denoted by "e_x" in the CLOCK list. The figures show the case when two requests to the chunk x arrives with the time interval t. The time interval t is the number of chunks accessed from the penultimate access to the last access, which is the definition of IRR [7]. In addition, we introduce two more parameters: P and Q. P is the rotation period of $\mathrm{HAND}_{\mathrm{cold}}$. Q is the period clearing the hash-tables (we discuss the period in Section 4.4). For brevity, we illustrate only $\mathrm{HAND}_{\mathrm{cold}}$, which rotates whenever a new chunk arrives and a cache replacement process is performed.

Here, we can consider two cases: (a) the first chunk x has a hot entry and (b) the first chunk x has a cold entry. In the case (a), the hot entry e_x remains in the CLOCK list regardless of t. The case (b) is further divided into two cases: (i) $t \leq P$ and (ii) $t > P$. Figure 3 shows the case (i). The cold entry e_x becomes hot because e_x is accessed

before $\mathrm{HAND}_{\mathrm{cold}}$ encounters e_x. Figure 4 shows the case (ii). The cold entry e_x is discarded from the CLOCK list and is registered in the hash-table to provide an opportunity for the discarded chunk x to be re-cached with a hot entry. In short, t affects the behavior of CUSH. Since t represents IRR, a chunk with low IRR is classified as hot and remains in the CLOCK list. On the other hand, a chunk with high IRR remains cold and is finally discarded when $\mathrm{HAND}_{\mathrm{cold}}$ encounters it.

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

CUSH can deal with SCAN and LOOP by using two types of chunks and ghost caches. The both access patterns pollute a cache, that is, all chunks in a cache is removed from a cache by the access patterns as explained in Section 3.1. CUSH can avoid the adverse effect of SCAN because only cold chunks are replaced by one-time requested chunks and popular chunks remain as hot chunks. CUSH can also cope with LOOP as explained in Section 4.4.

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

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

Figure 5 compares the overhead of a lookup table for a cache replacement policy using ghost caches. As shown in the left of the figure, which is the same as Fig. 1, ghost caches impose a high memory cost on a lookup table although ghost caches are essential to deal with SCAN and LOOP. The memory cost becomes prohibitive due to a name used in ICN in contrast to the computer system as explained in Section 3.3.

CUSH can avoid the memory cost on a lookup table for ghost caches. As shown in the right of Figure 5, CUSH requires the lookup table only for the CLOCK list. The hash-tables of CUSH do not require the support of the lookup table because an entry in the hash-tables can be accessed by computing $H(name)$. Thus, we can reduce the memory cost of the lookup table for ghost caches.

To reduce the cost, CUSH must accept a hash collision on searching a ghost cache. Suppose, for example, hashes of /b.txt and /c.txt take the same value as shown in Figure 5. When an initial request for /c.txt arrives at a router after /b.txt was removed from the cache and has been stored as a ghost cache, CUSH stores /c.txt as a hot chunk because $H(/b.txt) = H(/c.txt)$ and a hash collision occurs. It is concerned that, if /c.txt is one-time requested content, the unpopular content wastes the cache capacity for a long time and degrades the cache hit rate of CUSH.

However, the collision problem is less serious compared to the benefits of dealing with the access patterns. In addition, the scalable data structure of the hash tables enables to reduce the probability of hash collisions. We can expand the hash space by increasing k. Another solution is to use multiple bits instead of a one-bit flag. We explore the influence of the difference in the implementations of ghost caches in Section 5.2.

4.4 Management of Hash-tables to deal with LOOP

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

As mentioned in Section 3.1, each chunk in LOOP is removed from a cache before being accessed again; therefore, the all accesses to chunks in LOOP cause cache misses in spite of the repetition of accesses. Although LOOP is the repetition of the same SCAN accesses, the solution to SCAN, which classifies chunks into two types and keeps hot chunks, cannot be directly applied to LOOP. This is because all chunks in LOOP have the same recency and frequency; therefore, the number of chunks becoming hot is too large and the chunks overflow the capacity of hot chunks. Specifically, even if the ghost cache of a chunk in LOOP is accessed and the chunk becomes a hot chunk, too many chunks also becomes hot after that and the chunk is removed from the cache before being accessed again. Because this is applied to all chunks in LOOP, a cache hit never occurs.

Our solution to LOOP is not only to detect the LOOP access pattern but also to hold a cacheable portion of the chunks in the LOOP as hot chunks. The detection needs ghost caches, which is already introduced in CUSH, and the hold process needs to manage the number of ghost caches. To realize the latter process, CUSH clears the hash-tables at an appropriate time interval. If the interval is too long, CUSH tries to hold more hot chunks than the capacity and

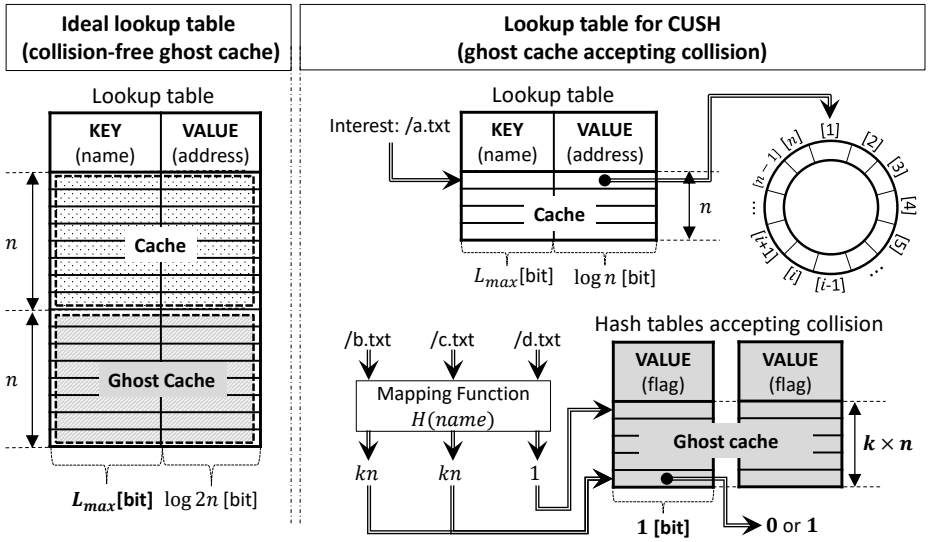


Fig. 5: Variations of Lookup Table for Ghost Caches

hot chunks overflow as described above. On the other hand, the too short interval reduces the opportunity to detect the repetition of LOOP because ghost caches are cleared before cacheable chunks are detected. We propose the criteria to determine the appropriate interval according to the number of hot chunks n_h. n_h is the upper bound that hot chunks do not overflow. Thus, CUSH clears hash-tables whenever the count of both a cache hit and a ghost hit becomes n_h.

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

5. Performance Evaluation

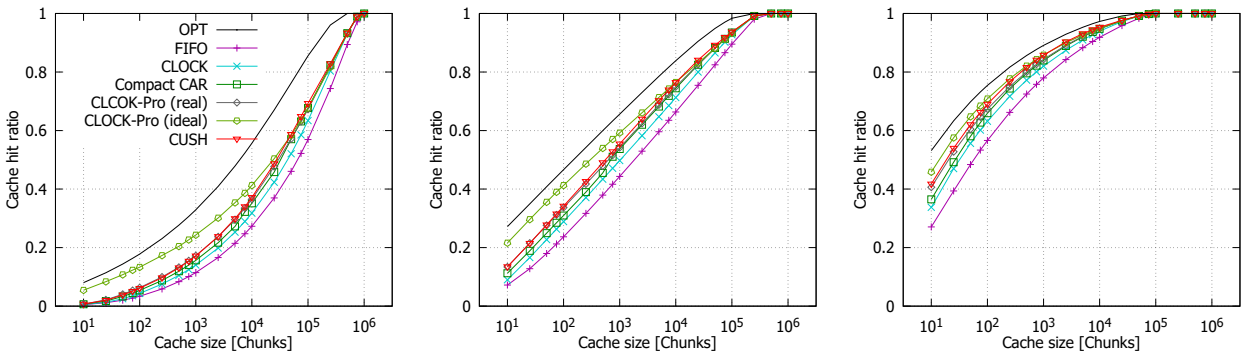
This section explores the following three facts with simulation studies: (1) Our proposal improves the performance against the access patterns specific to network traffic compared to existing cache replacement policies policies. (2) Our proposed management scheme of ghost caches improves the LOOP-resistant property in terms of its capacity and its collision probability. (3) The memory and computational costs of our proposal are low enough to install it into an ICN router.

5.1 Evaluation of Property Resistant to Access patterns using Synthetic Traffic

First, we demonstrate the adaptability of CUSH to several traffic access patterns compared to some cache replacement policies. The access patterns on which we focused in this simulation are SCAN and LOOP, which are caused by a large number of one-time used content items and the fine granularity of items as stated in Section 3.1. Unfortunately, ICN traffic traces are not available yet. Thus, we generate two types of synthetic traffic workloads: requests for content items and requests for chunks, which simulate the access patterns of ICN. The synthetic workloads follow a Zipf-like distribution because the popularity of Internet content (e.g., VoD, web pages, file sharing, and user generated content) has been reported to follow the Zipf-like distribution [2], [24]. According to the previous literature, we vary α, which is a constant parameter of the Zipf-like distribution, from 0.8 to 1.4. We also change the chunk size L from 1.5 KB to 60 KB.

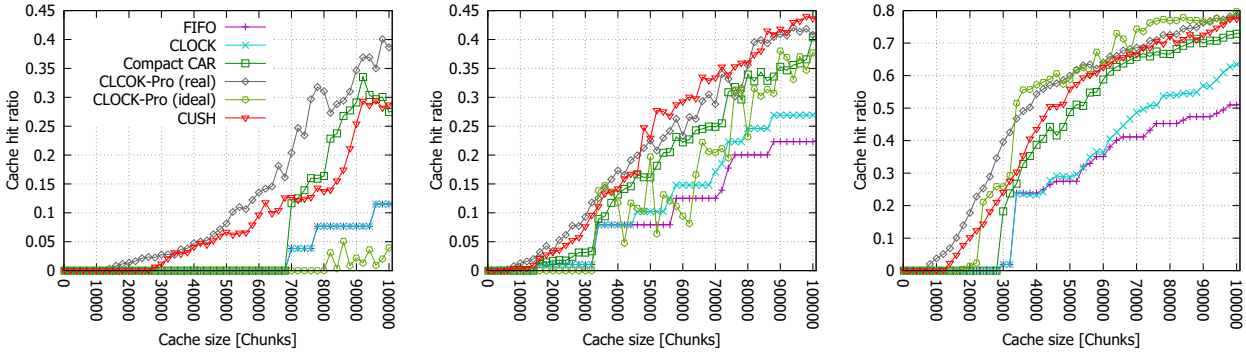
The cache replacement policies used in our evaluation are OPT (here, an offline optimal algorithm with a priori knowledge of the stream of requests), FIFO, CLOCK, Compact CAR, CLOCK-Pro, and our proposal. OPT provides the absolute upper bound on the achievable hit rate. FIFO, CLOCK and Compact CAR are the policies consuming the low memory and computational costs which can be implemented in an ICN router. Compact CAR is SCAN-resistant but is weak to LOOP. Although Compact CAR is originally designed on the assumption that it has the collision-free ghost cache as depicted in the Fig. 1, it uses the ghost cache that allows hash collision in our evaluation. CLOCK-Pro also uses ghost caches but we evaluate the performance of both implementations to show the effect of hash collision. `CLOCK-Pro(ideal)` and `CLOCK-Pro(real)` denote CLOCK-Pro using the collision-free ghost cache and CLOCK-Pro using the ghost cache without collision resolution, respectively.

An ICN router manages n entries, where n ranges from 10^1 to 10^6 chunks which are adjusted according to the traffic trace we adopt. In this evaluation, we assume that CUSH consumes $4n$ [bit] for ghost caches $4n$. We did not use complicated topology because our purpose is demonstrating the adaptability of our proposal to network traffic. A topology we used in our simulation can be assumed to have only one



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

Fig. 6: Evaluation for SCAN-resistant Property with Synthetic Workloads in Units of Content



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

Fig. 7: Evaluation for LOOP-resistant Property with Synthetic Workloads in Units of Chunks

ICN router between clients and a server. The transmission delay of each chunk on links and the unnecessary computation in the protocol stacks are also ignored to simplify the simulation.

Figure 6 depicts the cache hit rates for synthetic workloads in units of content. There is SCAN in these scenarios but LOOP does not appear. CUSH improves utilization efficiency by up to several-fold compared with simple policies that are weak to SCAN. We can find CUSH is superior to Compact CAR although we conjectured Compact CAR was better in a scenario without LOOP. This is because Compact CAR fails to detect and hold popular chunks caused by the hash collisions. Our low-overhead approximation shows the performance as good as original `CLOCK-Pro(real)`, while the ideal policy with collision resolution achieves the best performance.

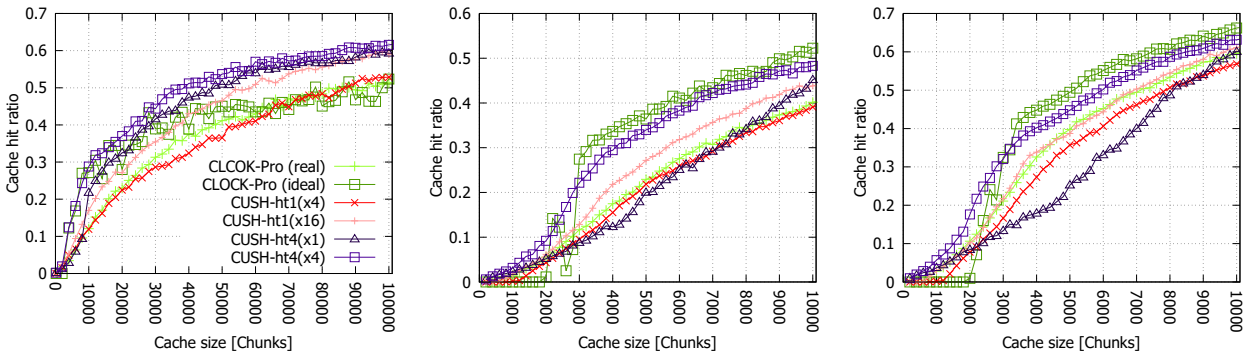
Figure 7 shows the cases when the size of cacheable chunks L is 1.5 KB. We can find the adverse effect of LOOP as some policies have no cache hit happen until the cache size exceeds a certain value. The policies that are weak to LOOP are disturbed by LOOP when the cache size is less than a certain value (e.g., 30,000 when $\alpha = 1.4$); however, CUSH achieves cache hits in the same condition. Thus, these results show CUSH can adapt to access patterns in network traffic. CUSH is also a good approximation of CLOCK-Pro. In fact, the performance of CUSH is comparable to that of CLOCK-Pro.

Table 1: Cache Policies Used to Analyze Effect of Ghost Cache

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

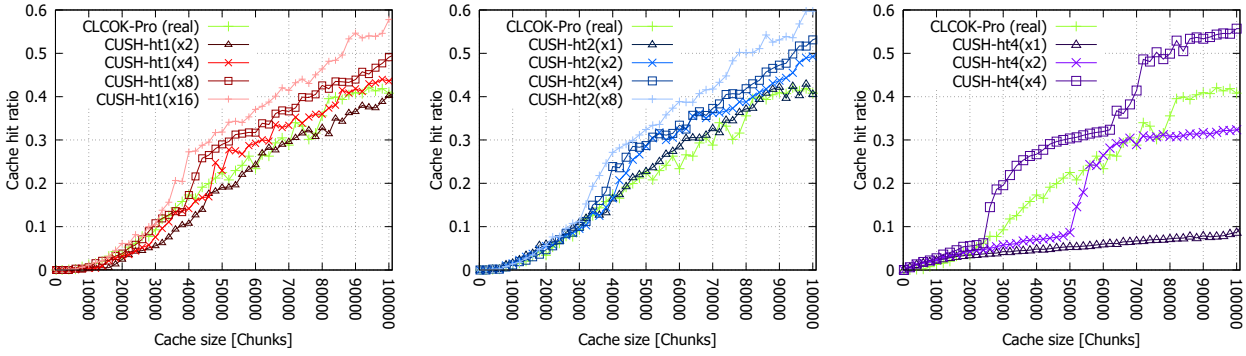
5.2 Evaluation of the Influence of Ghost Caches to LOOP-resistant Property

Then, we analyze the influence of the difference in the implementation of a ghost cache. The LOOP-resistant property is improved by increasing ghost caches and decreasing the hash-collision probability as mentioned in Section 4.3. Thus, we compare the policies shown in Table 1. CUSH-ht$b$(xk) denotes the implementation of CUSH with a hash-table that contains $k \times n$ entries and assigns b-bits per entry. For example, CUSH used in the previous section corresponds to CUSH-ht1(x4). Ghost caches increase as k becomes larger. Increasing b reduces the collision probability.



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

Fig. 8: Effect of Implementation of Ghost Cache on LOOP-Resistant Property



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

Fig. 9: Effect of the Amount of Ghost Caches on LOOP-Resistant Property (Using Synthetic Trace in Units of Chunks with$\alpha = 1.2$ and $L = 1.5\text{KB}$)

`CLOCK-Pro(real)` and `CLOCK-Pro(ideal)` are the same as used in the previous section.

In the evaluation, we adopt the synthetic workloads in units of chunks described previously according to the purpose of analysis the LOOP-resistant properties. Figure 8 shows the hit rates for the synthetic workloads where $L = 15K$ and α changes from 1.0 to 1.4. Figure 9 compares the hit rates for the synthetic workloads with $L = 1.5K$ and $\alpha = 1.2$ among different implementations of a ghost cache. The larger k is, the better performance becomes typically. On the other hand, when $k = 1$ or 2, the hit rates with $b = 4$ is inferior to the hit rates with $b = 1$ or 2. This indicates a part of LOOP becomes accidentally a hot chunk when the number of ghost caches are too small to detect the repetition part of LOOP. This is because a high collision probability (i.e., small b) causes a cold chunk to turn into a hot chunk randomly. In fact, the hit rates of CUSH-ht4(x4) are best because this case fulfills both a low collision probability and many ghost caches enough to enable collision-free detection of a part of LOOP. Thus, we can enhance the LOOP-resistant properties by only adding bits to each entry without a significant change to the data structure of CUSH.

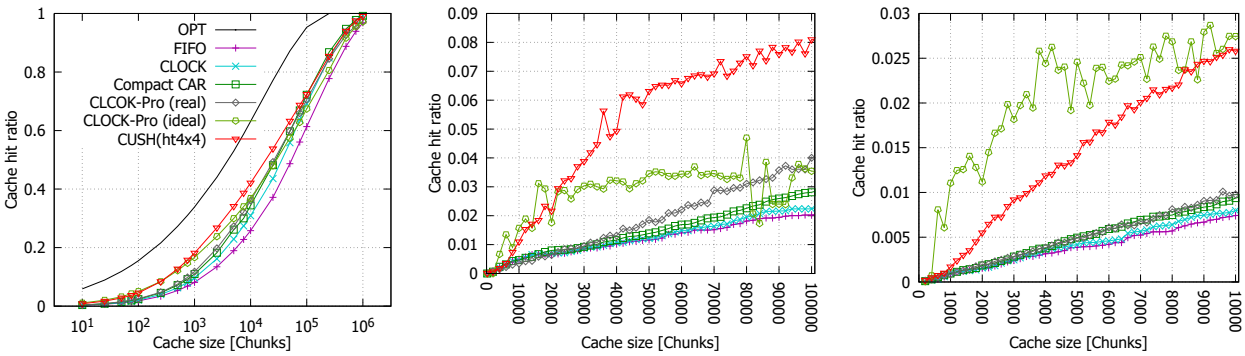
Table 2: Number of Chunks Per Second[pck/sec]

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

5.3 Cache Hit Rate with Real Traffic Trace

To justify the results using synthetic traffic, we also use the real traffic traces. We collected traces of VoD (e.g, YouTube, DailyMotion, and NicoVideo) from a network gateway at Osaka University campus. The traces are gathered from July 26th 2013 to February 26th 2015. The number of unique content is 1,451,558; the number of content requested at least twice is 381,527; and the number of total accesses is 3,378,925. The popularity distribution of the real traffic trace follows the Zipf-like distribution, as depicted in Fig. 10. We also show the statistics of the real traffic traces in units of chunks in Table 3. The inter arrive time for chunks is assumed to be constant according to Table 2, which is determined by the statistics of our observed real traffic.

Figure 11 presents the simulation results. Here, we use CUSH-ht4(x4) to exhibit the benefits of an enhanced ghost cache. In the real environment, it is important to deal with access patterns caused by the volatility of popu-



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

Fig. 11: Results for Real Traces

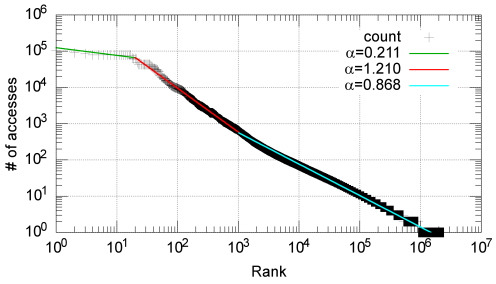


Fig. 10: Popularity Distribution of a Real Trace

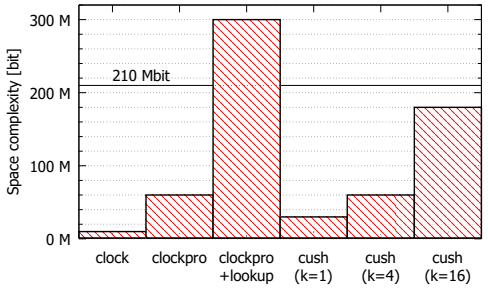


Fig. 12: Space Complexities of CLOCK, CLOCK-Pro, and Our Proposal When $n = 10^7$

Table 3: Statistics of Workloads in Units of Chunks

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

larity such as CORRELATED-REFERENCE and FICKLE-INTEREST. We can find that CUSH achieves higher hit rates than the others in the results for workloads in units of content shown in Fig. 11(a), which reveals the SCAN-resistant properties of CUSH. Figures 11(b) and 11(c) show the results for workloads in units of chunks, which contain LOOP. As `CLOCK-Pro(real)` is as bad as simple policies, CLOCK-Pro is difficult to cope with the changing popularity. CUSH outperforms the other policies except `CLOCK-Pro(Ideal)`. Thus, CUSH has the properties to cope with four access patterns discussed in Section 3.1, and therefore can adapt to real network traffic.

5.4 Analysis on Space and Time Complexities

5.4.1 Space Complexity

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

Table 4 summarizes the results of calculation with the big O notation. Our analysis compares seven algorithms: FIFO, LRU, CLOCK, CAR, Compact CAR, CLOCK-Pro, and CUSH. Because there are several ways to implement LRU, we consider the method using a doubly-linked list (denoted by LRU$_{DLL}$) in this analysis. Although an additional cost of a lookup table depends on how to implement the lookup table, we assume the lookup table for ghost caches is implemented as a hash-table without collision resolution and its additional memory cost is $O(x \log x)$ when the number of ghost caches is x. We also define the following notations and variables. n is the number of cache entries. The number of ghost cache entries is k times of n in CUSH.

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

The cache replacement policies using a cache history information (CAR, CLOCK-Pro, Compact CAR, and CUSH) add a memory cost of ghost caches in addition to

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

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

Table 5: Average hand movement count of CLOCK-based policies

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

a cost to manage actually cached entries. The implementation of CAR is based on doubly-linked lists and the number of ghost caches is n; therefore, the costs for cache management and ghost caches are $O(n \log n)$.

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

We also calculate the actual memory cost added by CLOCK, CLOCK-Pro, and our proposal when the number of cache entries n is 10 million according to the previous literature [23], [25]. Figure 12 shows the results of the calculation. A policy based on a CLOCK list typically requires a memory overhead of several bits per chunk. CLOCK consumes 10 Mbits because it assigns only R-bit to each chunk. CLOCK-Pro has a modified CLOCK list, which contains $2n$ entries and assigns three bits to each entry (R-bit, H-bit, and a bit called a test flag), and so consumes 60 Mbits. CUSH assigns two bits to n entries and has n-bits hash-tables; therefore it consumes 30 Mbits when $k = 1$.

When taking account of the costs for ghost caches, CLOCK-Pro additionally consumes 240 Mbit for looking-up ghost caches. Thus, the total cost becomes 300 Mbit (denoted by "`clockpro+lookup`" in Figure 12). This cost is prohibitive because of the severe constraints of fast memory enough to be employed in an ICN router such as SRAM, whose available size is 210 Mbit [26]. On the other hand, CUSH is free from additional cost of a lookup table. In addition, CUSH can conserve the cost when enhancing a hash-table for ghost caches. Even if $k = 16$, the total memory overhead cost of CUSH is 180 Mbit. Thus, CUSH is a low-overhead and scalable cache replacement policy that satisfies the memory requirements of an ICN router.

5.4.2 Time Complexity

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

Table 5 shows the number of hand movement. We evaluated the four policies: CLOCK, CLOCK-Pro, CUSH and Compact CAR. The number of hand movement intuitively depends on a cache hit rate because the process continues when there are many entries with $R = 1$ caused by many cache hits. For clarity, we use the synthetic trace with $\alpha = 1.0$, where a hit rate increases in proportion to $\log n$. We show both the average counts for the all accesses and the average counts for the cache misses because a cache miss begins the hand movement process. We omit an evaluation of the worst case because the worst-case complexity is obviously $O(n)$.

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

also has two types of chunks, the average hand movement count of Compact CAR is constantly less than four because the two types of chunks are maintained separately.

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

6. Conclusion

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

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] L. Zhang, D. Estrin, J. Burke, V. Jacobson, J.D. Thornton, D.K. Smetters, B. Zhang, G. Tsudik, K. Claffy, D. Krioukov, D. Massey, C. Papadopoulos, T. Abdelzaher, L. Wang, P. Crowley, and E. Yeh, "Named data networking (NDN) project," October 2010.

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

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

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

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

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

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

[8] F.J. Corbato, "A paging experiment with the Multics system," tech. rep., DTIC Document, May 1968.

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

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

[11] A.M.K. Pathan and R. Buyya, "A taxonomy and survey of content delivery networks," tech. rep., University of Melbourne Grid Computing and Distributed Systems Laboratory, February 2007.

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

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

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

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

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

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

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

[19] A. Ooka, S. Eum, S. Ata, and M. Murata, "Compact CAR: Low-overhead cache replacement policy for an ICN router," December 2016. arXiv:1612.02603.

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

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

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

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

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

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

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