New Generation Computing, 31(2013)311-329
Ohmsha, Ltd. and Springer

NEW GENERATION COMPUTING

©Ohmsha, Ltd. and
Springer Japan 2013

Web Caching Replacement Algorithm Based on Web Usage Data

Sorn JARUKASEMRATANA, Tsuyoshi MURATA
Department of Computer Science
Graduate School of Information Science and Engineering
Tokyo Institute of Technology
W8-59 2-12-1 Ookayama, Meguro, Tokyo, 152-8552, JAPAN
sorn.jaru@ai.cs.titech.ac.jp, murata@cs.titech.ac.jp

Received 30 October 2012
Revised manuscript received 22 July 2013

Abstract Web caching is one of the fundamental techniques for reducing bandwidth usage and download time while browsing the World Wide Web. In this research, we provide an improvement in web caching by combining the result of web usage mining with traditional web caching techniques. Web cache replacement policy is used to select which object should be removed from the cache when the cache is full and which new object should be put into the cache. There are several attributes used for selecting the object to be removed, such as the size of the object, the number of times the object was used, and the time when the object was added into the cache. However, the flaw in these previous approaches is that each object is treated separately without considering the relation among those objects. We have developed a system that can record users' browsing behavior at the resources level. By using information gathered from this system, we can improve web cache replacement policy so that the number of cache hits will increase, resulting in a faster web browsing experience and less data bandwidth, especially at lower cache storage environments such as on smart phones.

Keywords: Web Caching Replacement Algorithm, Web Usage Mining, Web Browsing, Least Recently Used Algorithm.

§1 Introduction

In recent years, the internet has become the most important tool for communication and interaction among people, resulting in the increasing of data

bandwidth. Web caching is a well-known strategy for improving the performance of web-based systems. Web caching can improve the performance of the WWW by creating duplicates of popular objects and temporarily storing them in cache storage near the users. If those objects are requested again by the users, users will receive those objects from the cache instead of the original servers. This is called hit or cache hit. Web caching gives benefits to both web users and websites' owners because 1) caching reduces total bandwidth usage, 2) caching reduces web page load time and 3) caching reduces loads on website servers.[1)] Web caching can be applied at the original server, proxy server, or client-side machine. In this research, we focus on client-side caching (or browser caching) because client-side caching is more economical and effective than server or proxy caching.[2)]

Since cache storage has limited space, as users continuously browse the internet, cache storage will eventually become full. When a new object needs to be stored in the cache while the cache is full, cache replacement policy will determine which object will be removed to make enough space for the new object. To use the limited cache space in the most efficient way, objects that will not be used again should be removed from the cache first.

There are many cache replacement policies, each with their own algorithms for selecting the objects to be removed. The general goal of cache replacement policies is to increase cache hit rate. The most well-known algorithms are LRU (least recently used) and LFU (least frequently used). LRU chooses objects to be removed based on the last time the objects were used, while LFU chooses objects based on how many times the objects were used. Other properties are used in other algorithms such as the size of the object, the total time used for downloading the object, or the last modification time of the object. There are also many algorithms that use more than one attribute to choose which object to remove such as Hyper-G,[3)] which uses a combination of frequency, recency and object size. Each algorithm has its own advantages over others. For example, in the case of a system with sufficient processing power and memory resources, a complex algorithms which require more computation are more suitable. On the other hand, on a system with limited processing power and memory resources, randomized strategy is preferred because it requires less memory and computation.[4)]

Research involving web cache replacement policy has been active for a very long time, and many algorithms have been theorized and invented. Kin-Yeung Wong[4)] claims that there is a sufficient number of good policies and further proposals would only produce little improvement. However, the internet is changing all the time. Users' behaviors are changing rapidly, especially with the recent proliferation of social media such as Facebook and Twitter. Web browsers are also changing from standard desktop personal computers to laptops and mobile devices. Therefore, a new cache replacement policy that is suitable with current browsing behaviors is required. To understand more about users' behaviors, web usage mining is normally employed.

Web usage mining has been an interesting topic among many researchers

since the WWW has become a part of our daily life. Web usage mining is the process of extracting useful information from a user's browsing history.[11] The benefit of web usage mining is that it lets researchers understand the behavior of internet users and use that knowledge to improve their web browsing experiences.[16] This technology enables websites to be personalized for each individual user. Some successful examples of web usage mining are real-time recommendation systems, such as youtube.com VDO recommendations or amazon.com book recommendations.

In this research, we incorporate web usage mining data into the cache replacement policy to create a new algorithm that can perform better than existing algorithms. Our proposed algorithm is a hybrid algorithm based on recency, frequency and users' web usage history. The idea for this algorithm comes from the fact that nowadays, users tend to visit the same set of websites every day. Objects from those websites should be prioritized and should stay in the cache longer than objects from other websites. Our algorithm can create a higher cache hit rate than the baseline LRU algorithm in a low cache storage space environment. This makes our algorithm perfectly fit for web caching on smart phones. The downside is that our algorithm is more complex than LRU.

The remainder of this paper is organized as follows: Section 2 reviews related research and discusses web caching, web usage mining, LRU algorithm, and current browsing behavior; Section 3 explains about our algorithm; Section 4 explains about our experiment; Section 5 evaluates and discusses of our experiment; Section 6, the last section, concludes the paper.

§2 Related Work

2.1 Web Caching Strategies

Web caching has become a hot topic among researchers since Luotonen and Altis[5] introduced proxy server to the research field in 1994. Web caching can be deployed at three locations: the server, the proxy server, and the browser. Caching at the server mainly serves two purposes: reduce workload on a single server, or store generated dynamic pages. Proxy caching involves shared cache between sets of clients, such as a proxy server for a university or company. Their cache policies are designed to exploit the overlap of HTTP requests among users in the group. Browser caching occurs on users' machine and can provide the biggest boost in browsing speed for users.[2]

There are at least three survey papers on web cache replacement algorithms. One survey of web cache replacement strategies was presented by Podlipngi and Bszrmenyi[1] in 2003. They classified replacement strategies into five classes: recency-based strategies, frequency-based strategies, recency/frequency-based strategies, function-based strategies, and randomized strategies. Recency-based strategies use a temporal factor to manage the cache. Basically, the least recently referenced object will be removed from the cache first. Recency-based strategies are adaptive to popularity change and mostly require low overhead. However, these strategies usually place too much emphasis on the recency factor

alone, which is a major disadvantage. Frequency-based strategies use frequency as their main factor. Frequently-called objects or popular objects tend to be kept in the cache longer than unpopular objects. These strategies perform very well at caching objects in an environment where popular objects do not change frequently. However, in a quickly changing environment, frequency-based strategies will perform poorly. Recency/frequency-based strategies use both recency and frequency properties to identify objects to be removed. Many strategies in this class also use other factors in their algorithms. These combinations usually give better results but at the cost of overhead and complexity. Function-based strategies use a general function to calculate scores for all objects in the cache, and then the object with the lowest score will be removed. Many parameters such as size, recency, and frequency are used in the function. The strongest advantage of these strategies is that no particular attribute is dominant. However, these strategies create the largest overhead and complexity among all the classes. The last class constitutes randomized strategies. The goal of these strategies is to reduce complexity and overhead. Therefore, these strategies are very simple to implement. The problem with these strategies is in the evaluation. Different simulations on the same test data set can give different results.

Wong[4)] stated that there were more than 50 cache policies by the year 2006, when he published his cache replacement policies review. Instead of arguing about which cache replacement algorithms were better in general, he stated that each algorithm will perform better than others in its favored environment. For example, frequency-based strategies perform well when popular objects are not changing rapidly. Romano and ElAarag[6)] did a further quantitative study of web cache replacement strategies by comparing 19 different strategies running the same data sets. There are also several tools created for measurement the goodness of caching techniques. One of them is created by Cao[26)] which included LRU, SIZE, and HYBRID algorithm. This tool is used as the base code of our simulator.

Cache replacement policies have been heavily researched in the past. Therefore, newer researches on this topic are mostly conducted by using knowledge from other research fields. Tirdad et al.[7)] used a genetic algorithm and genetic computation to create a model for cache replacement policy. Ali and Shamsuddin[8)] used neuro-fuzzy system to create an intelligent web caching scheme. Torkzaban and Rahmani[9)] used a multi-expert technique to create a cache system that can select the best cache policy depending on the environment. Geetha et al.[10)] created SEMALRU, a LRU-based algorithm that uses semantic data from the web pages.

2.2 Web Usage Mining

Web usage mining is the process of extracting useful information from a user's browsing history. Srivastava et al.[11)] offered a taxonomy for web usage mining in 2000; since then, web usage mining has been given more interest in the computing field. There are three stages in web usage mining: data collection and pre-processing, pattern discovery, and pattern analysis.[16)] In the first stage,

raw data is cleaned and arranged so that it can be easily processed. In the second stage, many operations such as machine learning are performed in order to find meaningful patterns in the data set. In the last stage, the discovered patterns are analyzed and processed into useful models or applications such as visualization models or recommendation engine.

Similar to web caching, web usage mining can be done at three levels: server, proxy server, or client-side machine. In the early stages, web usage mining was usually done by analyzing server logs obtained from servers or proxy servers, such as the work of Myra Spiliopoulou[12] that utilized web usage mining data for evaluating websites. Client-side log mining is getting more popular because of technological advances such as web browser plug-ins that allow researchers to collect data directly from users. Studies of client-side logs can give many insights about users' behaviors. Zhou et al.[13] proposed a temporal-based web access behavior which focuses on the time of day that users access the web. Khoury et al.[14] created a graph based on users activities over some period of time. The result was that they could analyzed users' behaviors such as finding the frequent traverse paths of users as they utilize hubs like Wikipedia or e-mail to traverse to other websites.

According to Cooley et al.,[17] data can be categorized into four main groups: usage data, content data, structure data, and user data. Usage data is normally the primary target for web usage mining. It represents which web pages the user visits while browsing the internet. This type of data typically comes in the form of server logs. Content data refers to web objects that the user receives when visiting websites. This data can be in various file formats like, image files, text files, or HTTP files. Content data also includes other embedded information within the sites such as page descriptions or tags. Structure data reflects to the content organization of the web pages, such as how web pages store data on their servers, or how web pages are divided into separate frames. User data comprises data about the users themselves. This data may include a user's region (from IP address), purchase records, web browser version, or more. In this research, structure data and usage data are used.

2.3 LRU Algorithm

LRU (Least Recently Used) is probably the simplest and most widely used cache algorithm. The idea behind the LRU cache eviction process is very simple: remove the oldest object in the cache. LRU can be implemented by using a linked list. Each object in the cache refers to one object in the list. A new object is always added at the head of the list. If the list is full, the last object in the list will be evicted. The most important rule of LRU is that when a cache hit occurs, whereby the object requested by a user is already in the cache, that object is moved to the front of the list, similar to a new object. Despite its simplicity, LRU is very efficient.[4]

The disadvantage of this algorithm is that it cannot keep popular objects (objects that have a high possibility of being used again) if the cache storage is small. Take for example a user who visits a social media website everyday on

a mobile device (where cache storage size is small due to memory limitation). The social media website's web objects, such as logos or menus, are stored in the cache. However, if the user visits many other websites afterward, this will cause social media website's web objects to be evicted from the cache even though they would be very likely used again. To combat this flaw, we have designed our algorithm to incorporate users' browsing history so that popular objects are stored in the cache even when the cache is full and eviction process occurs.

2.4 Current Users' Browsing Behavior

The main idea behind this algorithm is based on the results of Khoury et al.,[18] Goel et al.[19] and our own usage mining. These studies show that users tend to visit the same set of websites every day. Moreover, users mostly utilize those websites as hubs, which denote websites that link to other websites. Khoury et al. reported that Wikipedia websites, e-mail websites, and news websites are the main hubs of users' browsing sessions. Two examples from Khoury et al. are Reddit.com - a social news website where users can post interesting things found on other websites and Google Reader - an online RSS feed website. As seen from Fig. 1(a), this user went through Reddit.com and Google Reader to visit many other websites. The node sizes of these two hubs are very big, which indicates that they are visited very often. Therefore, web objects from these two hubs should be kept in the cache storage. Our own data and Goel et al. also found out that news websites and social media websites such as Facebook are the main hubs of users' browsing activity. Figure 1(b) presents the research results from Goel et al. It can be seen that web usage nowadays is very skewed, especially

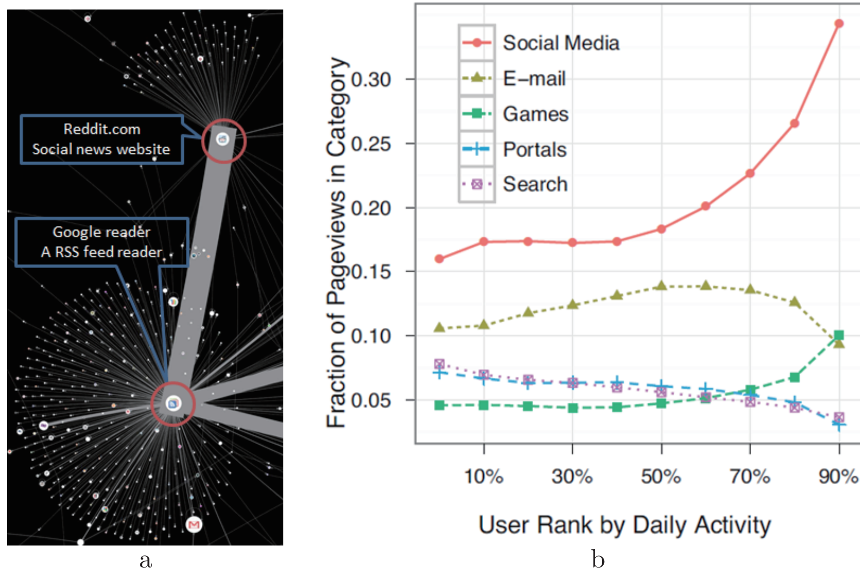


Fig. 1 (a) A frequent browsing websites of one participant, Khoury et al.[18] (b) Social Media is visited 90% daily, Goel et al.[19]

toward social media. Social media websites received 38% of pageviews in 90% of daily activity. Web objects from frequently visited websites like these should be cached and should be preserved in cache longer than other objects.

§3 Our Algorithm

In our previous experiment, "Visualizing Web Structure based on Browsing Sessions,"[21] we transformed users' browsing sessions into graphs. Nodes of the graph consist of web page files, such as HTML or PHP files, web objects, such as image files or text files, and hosts of those objects. Edges are generated based on the relations between web pages, web objects, and hosts. The example of this system is in Fig. 2.

While a user continued browsing amazon.com, we found some web objects that appeared in every pages or appeared very often (Fig. 3). These objects are mostly logos, texture for menus, loading icons, and so on. We believed that these type of objects should be prioritized and kept in cache. Since what these

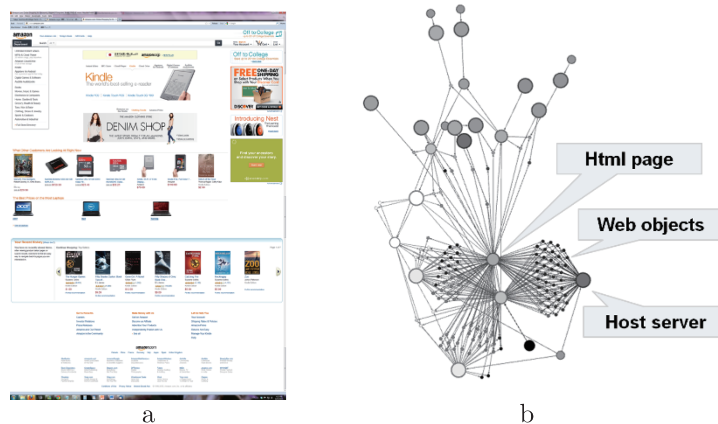


Fig. 2 (a) Main page of amazon.com in web browser (b) A graph created from (a). There are two main HTML pages, A lots of web objects, and three main hosts.[21]

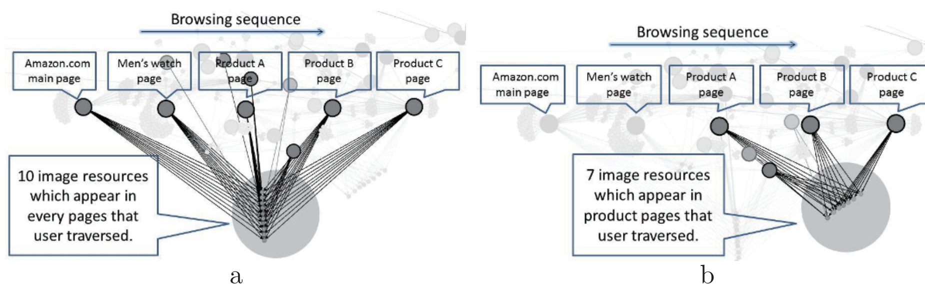


Fig. 3 Some objects are used very frequently. (host nodes are omitted in these pictures to make graph easier to read.)

objects had in common are host URL, we decided to use host URLs as the tool in our algorithm.

Our algorithm is based on LRU strategy but with extra properties, which are a host URL and a hit-point value (algorithm generated value). However, unlike LRU, our algorithm falls into Recency/frequency-based strategies class based on Podlipngi and Bsz Bszrmenyi classification[1] because we also incorporate frequency-based properties into our algorithm. Our algorithm is divided into three parts: cache storage structure, cache insertion policy, and cache eviction policy.

3.1 Cache Storage Structure

Similar to other LRU based strategies, our cache objects (such as pictures, Java script files, HTML files) are kept in a list. There are six properties for each object: object id, next object id, previous object id, size of the object, object's host, and object's HP (an integer value generated by our algorithm). Our algorithm structure is illustrated in Fig. 4. Object's host is the URL of the server that provides users this object. This is the extra information gained from web usage mining. In normal web browsing situation, this information is omitted from the users. However, with a packet analyzer (a computer program that can intercept and log traffic passing over a digital network or part of a network), it is possible to record this information while users are browsing the internet.

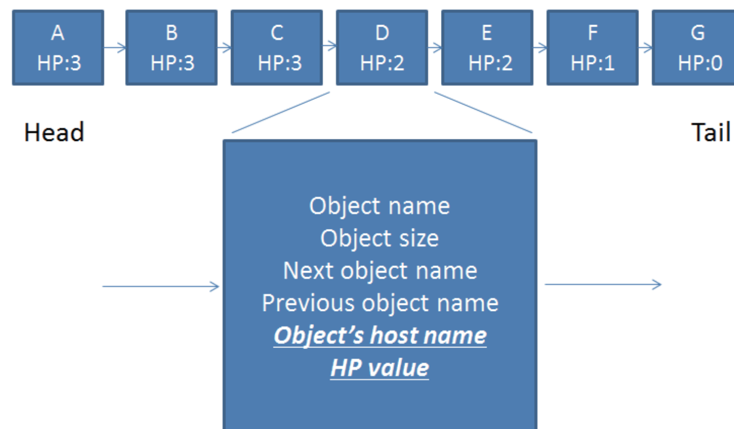


Fig. 4 Our cache structure, object's host name and HP value are added properties from LRU structure.

Object's HP stands for object's hit-point. This integer value is generated and managed by our algorithm. Objects with less HP are considered as an unimportant object and will be removed first when eviction occurs. The maximum HP value needs to be defined to prevent objects from being too high HP and will never be evicted from the cache. In this experiment, maximum HP is set at 5. The minimum HP value is 0. If the HP of the object reaches 0, it is guaranteed

that in the next eviction phase, this object is going to be evicted.

Apart from the objects link list, this algorithm needs to keep at least two lists of unique host URL: today's host list and yesterday's host list. The size of the list is very small compared to objects list because many objects share the same host URL. These lists take part in determining the HP of the objects. The lists are maintained on daily basis.

Compared with LRU algorithm, our algorithm requires more space since ours has to store lists of unique hosts. Moreover, more space is required for each object, because our algorithm has more properties per object.

3.2 Cache Insertion Policy

Our insertion policy is quite different from LRU since we have more properties (HP and host URL) and host lists to maintain. LRU based algorithm inserts new object at the front of the list while remove the last object in the list (oldest object), while our algorithm puts new object at the back of the list instead. The reason is because our algorithm eviction process is different from the eviction process of LRU. When an object wants to be entered in the cache, the following algorithm (Algorithm 1) is applied.

```
Data: Cache insertion algorithm
object enters the cache;
if object is the new object then
    if cache is full then
        performs cache eviction;
    end
    adds new object to the end of the cache list;
    object's hp = 1;
    adds object's host url to today's list;
else
    if object's host url is new then
        object's hp += 1;
        add object's host url to today's list;
    else
        object's hp += 2;
    end
    moves object to the end of the cache list;
end
```

Algorithm 1: Cache Insertion Algorithm

The most important point of this algorithm is that the hit objects are given more hit-point if they are related to previous browsing history. This results in objects that come from previously visited host to be kept in cache longer than objects that do not. Our algorithm also includes the part that remembers today's host url to be used in the later day. If an object entering the cache is a new object, that object will be placed at the end of the list and its HP is 1. At 24 hours interval, today's host list will be copied to yesterday's host list. Then today's host list will be emptied.

To explain it more clearly, suppose a user browsed website A yesterday, then he/she browses website A and B today. When objects from website B is hit, their HP will go up by 1. On the other hand, when objects from website A is hit, their HP will go up by 2. This is because website A was browsed yesterday and was stored in history list. Since higher HP objects got removed slower than low HP objects, objects from A are kept longer than objects from B. However, both websites A and B are now stored in the system. Tomorrow, if the user creates a cache hit from website A or B, their objects' HP will go up by 2 (because website A and B are already in the system).

3.3 Cache Eviction Policy

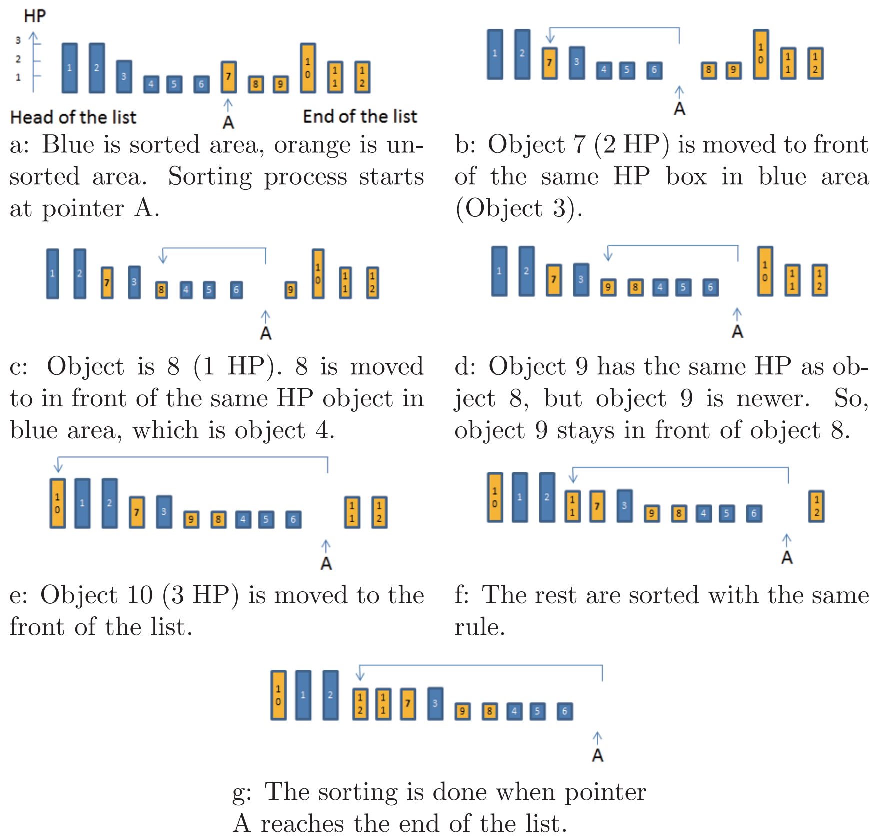
Once the cache is full and new objects are needed to be put in, eviction process will be called. The eviction process is explained in Algorithm 2.

```
Data: Cache eviction algorithm
Sort_cache_list;
int total_removed_size = 0;
node_pointer current_node = last_node;
while total_removed_size < X*cache_size do
    total_removed_size += current_node.size;
    current_node = current_node.previous;
    delete(current_node.next);
end
```

Algorithm 2: Cache Eviction Algorithm. X is the Percentage of Eviction.

The sorting process is explained with an example shown in Fig. 5.
In LRU, once the cache is full, insertion and eviction occurs almost always together because LRU only evicts as less objects as possible to maintain the cache to be near full all the time. This can increase the hit chance because there are many objects in the cache. However, in our algorithm, we already determined that unpopular objects is at the end of the list. Therefore, it is safe to remove many of them at once. Moreover, with this HP system, new objects need to stay in the cache for sometimes in order to gain more HP value. If eviction process occurs too often, all new objects are likely to be removed.

The final step of reducing all objects HP by 1 is crucial in this algorithm.



a: Blue is sorted area, orange is unsorted area. Sorting process starts at pointer A.

b: Object 7 (2 HP) is moved to front of the same HP box in blue area (Object 3).

c: Object is 8 (1 HP). 8 is moved to in front of the same HP object in blue area, which is object 4.

d: Object 9 has the same HP as object 8, but object 9 is newer. So, object 9 stays in front of object 8.

e: Object 10 (3 HP) is moved to the front of the list.

f: The rest are sorted with the same rule.

g: The sorting is done when pointer A reaches the end of the list.

Fig. 5 Sorting in cash eviction process. Blue objects are sorted from previous eviction process. Yellow objects are new objects or objects that got cache hit and moved to the end of the list.

In case of a popular object which already reach maximum HP (assume that maximum HP is 5) that suddenly never got any cache hit again. This object will be reduced to 0 HP in 5 eviction processes. Objects that have 0 HP are guaranteed to be removed from the cache on next eviction process because new objects are starting with 1 HP.

3.4 Configurable Parameters

Our algorithm has some configurable parameters which can be adjusted.
- Maximum HP: the higher maximum HP results in a longer stay of the object in cache (if it can manage to reach that HP). In a data set where popularity of the objects is changing regularly, maximum HP should be set to a low value.
- X% of eviction: the higher X value means more objects are deleted from the cache and the longer time before cache need to perform eviction process again. However, the higher X creates a higher risk of losing popular objects. We have perform another experiment on our data set, using X value range from 10% to

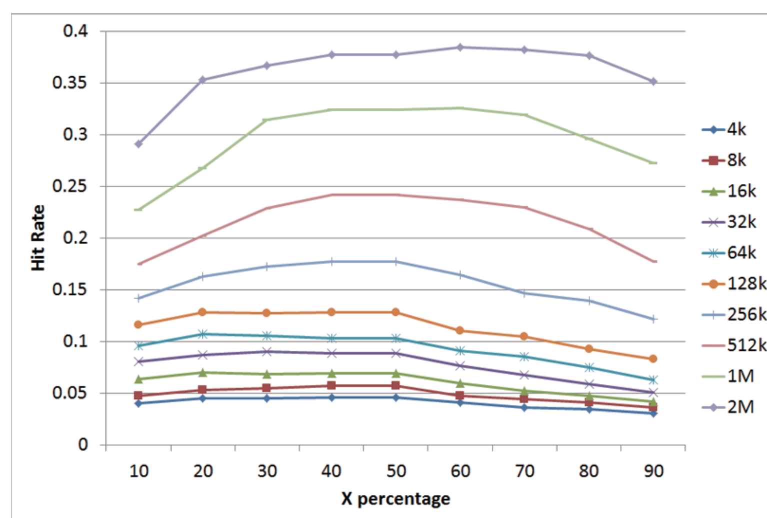


Fig. 6 Cache hit-rate on various cache storage sizes with different X percentage. The higher the hit-rate, the better.

90% at various cache storage size. The result is as shown in Fig. 6.

As seen from the graph, the highest performance point is in the middle. This is because the value around 50% is the most balanced point between the risk of losing old popular objects and the risk of keeping too many unpopular objects.

- Number of previous host list: in this experiment, we used only one list which is yesterday's list. However, it is also possible to keep record of several lists such as last week list or last month list and award objects that share host URLs accordingly.

3.5 Algorithm Complexity Comparison

Algorithm complexity is compared between our algorithm and LRU. LRU algorithm complexity is O(n) on cache insertion because the algorithm needs to check if the newly inserted object is already in the list. Linked list finding function complexity is O(n). Our algorithm performs the same function as LRU in this stage, therefore the complexity is the same. LRU algorithm cache eviction complexity is O(1), since its only pop the last item in the list. On the other hand, our algorithm cache eviction requires a sort process. However, with sorting algorithm used in Fig. 5, our sorting can be done in one pass through, results in O(n) complexity. In conclusion, both algorithms have the same complexity as O(n).

§4 Experiment

To evaluate our algorithm, we collected raw web usage data from 10 volunteers from different occupations which are entrepreneur engineer programmer and student. All of the volunteers use internet regularly both for work and per-

sonal life. Data collection lasted from 3-9 days, depending on the usage of each volunteer. There are two main reasons we decided to collect data by our own, instead of using existing trace data (raw web usage data) that are available in the internet. The first reason is that our algorithm requires host data. Most trace data that are available for download does not provide this property. The second reason is that most trace data are very old and possibly outdated. For example, the latest client-sided trace data from The Internet Traffic Archive,[14] is collected in 1996 or the latest client-sided trace data from web-caching.com[15] is dated back in 1998. The WWW is changing all the time, especially the trend of social networking websites, such as Facebook. In our trace data, Facebook appears very often as both page hit and web hubs.

Data collection was performed by installing Mozilla Firefox and Mozilla Firefox plug-in on client machines called Tamper Data.[23] Special configurations of Mozilla Firefox are needed to be adjusted. Most cache functions of web browser were disabled in order for plug-in to collect accurate browsing data. Volunteers can use Firefox as normal web browser while add-on collects the browsing data in the background (Fig. 7). There are some disadvantages for volunteers in this method. 1) Browsing data are highly confidential data. To ensure maximum privacy, we had transformed all raw data into numerical number with irreversible method. Testers were also made anonymous. 2) Volunteers were asked to use a Mozilla Firefox on personal computer as their main internet browser. Since some volunteers were using a Google Chrome or an Internet Explorer as their main web browser, the Mozilla Firefox was new to them. The reason we choose this browser in our research is from Firefox's rich add-on functionality. 3) Since all cache functions were disabled during the data collecting period, browsing speed was reduced. 4) Unlike other data gathering methods, such as performing a task or answering questionnaires, data collection lasted for several days. The task was a burden for volunteers.

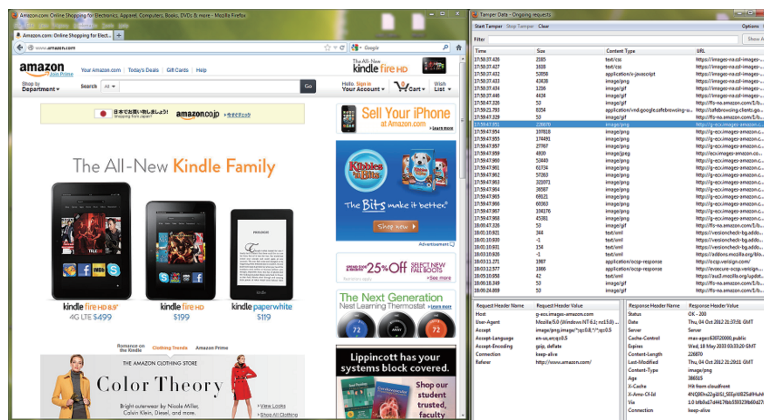


Fig. 7 Mozilla Firefox with add-on for collecting HTTP requests. Each line in add-on top window represents one object. Two bottom windows give details about highlighted object.

HTTP data was collected in XML format. One HTTP request contains many information about data and transaction. One request is for one object, which means if web page contains 20 pictures and 1 html file, there will be 21 requests for that page.

There were total of 1180 megabytes of raw log data with more than 400,000 HTTP requests. We compressed the data into numerical number so that it was irreversible and easier to compute in the simulation. The example of compressed data for one transaction is 1 3510 47551 0 0 0 221 3502 375 316894181.

We tested our algorithm and other five algorithms which are LRU, SIZE, HYBRID, GSDF[24] and LFUDA. LRU is the baseline of recency-based algorithm. SIZE algorithm emphasizes on size of the object. HYBRID is a mixed strategy between recency, frequency, and size. GDSF or greedy dual size frequency is a function based algorithm based on size and frequency. LFUDA or Least frequency used with dynamic aging is the improved version of LFU where dynamic aging is used to remove the old objects that get stuck in cache. The reason that we choose LRU as our baseline algorithm is because LRU is still probably the most popular cache replacement algorithm. Squid,[25] the popular caching proxy software based on GNU license, uses LRU as their default web cache replacement algorithm (version 3.3.1, 9 February 2013). Other algorithms that are available in Squid are GSDF and LFUDA, which is the reason why we included these two algorithms in our experiment as well. The parameters of our algorithm were X=50, maximum HP was 5 and we used one host list which was yesterday's list.

We run all algorithms on different cache storage sizes, range from 4KB to 4MB. The results are shown in Figs. 8 and 9.

In Fig. 8, Y axis refers to hit-rate. X axis refers to cache storage size.

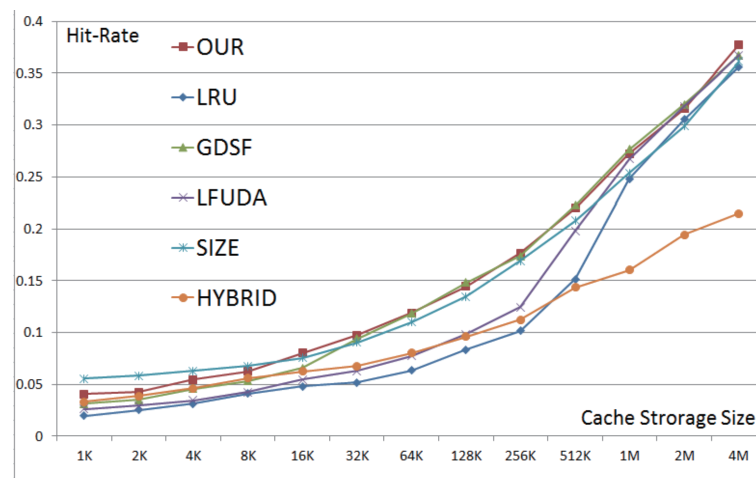


Fig. 8 Our algorithm results compared with LRU, SIZE, HYBRID, GDSF, and LFUDA algorithm. The higher the hit-rate, the better.

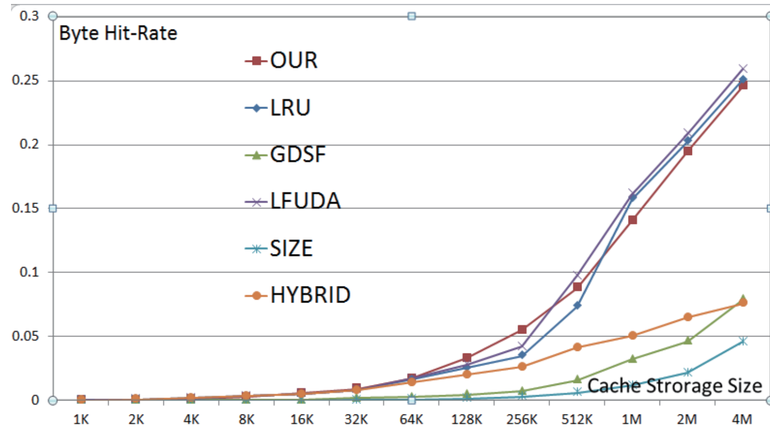


Fig. 9 Byte hit ratio comparison between our algorithm, LRU, Size, Hybrid, GDSF, and LFUDA. The higher byte hit ratio the better.

Hit-rate translates directly to the performance of the cache algorithms, which means the higher, the better.

Another common measurement of web caching is byte hit ratio.[22] Byte hit ratio is calculated by the number of bytes that got cached hit divided by total byte. This number is the actual representation of how many bytes that saved from re-downloading. The byte hit ratio of all algorithms are shown in Fig. 9. The higher the byte hit ratio, the better.

§5 Evaluation and Discussion

On hit-rate, our cache replacement algorithm performs better than baseline algorithm LRU at all cache sizes, the advantage can be clearly seen at low cache storage size. From 512 kilobytes, our algorithm and LFUDA's performances start to converge and at 1 megabyte, performances are converged into about the same value. GDSF and SIZE perform similar to our algorithm at all cache size. Our algorithm also outperformed Hybrid at all cache storage size. The reason that our algorithm performs better at lower cache size is because popular objects did not get called fast enough before it got evicted from others' cache. On the other hand, our strategy gives more priority to popular objects by learning from the past (via yesterday's host list). However, when the cache size is big enough, the performances of both algorithms are the same. This is because the cache size is big enough for all cache algorithms to store all popular objects.

On byte hit ratio, our algorithm performs roughly similar to LRU and LFUDA, while outperforms SIZE, HYBRID, and GDSF. Unlike hit-rate, byte hit rate does not solely depend on popularity of the objects. Unpopular large object, when got cache hit, will create more impact on byte hit ratio than impact on hit-rate. At higher cache storage size, LRU/LFUDA stores a lot of unpopular objects, while our algorithm will evict them. This create more chances for

LRU/LFUDA to create a hit on large unpopular objects, which result in a little better byte hit ratio than our algorithm at the end. In conclusion, our algorithm has better hit-rate than LRU, LFUDA, and HYBRID especially at lower level storage, while maintain same hit-rate as GDSF and SIZE. However, Our algorithm significantly outperforms HYBRID, GDSF and SIZE in byte hit-rate while maintaining similar byte hit-rate as LRU and LFUDA. Comparison of our algorithm against other algorithms is as follows.

Advantages

- Our algorithm hit-rate is better at low storage space especially sub 1 megabytes region.
- Once cache is full, other strategies will perform cache eviction process all the time while our strategy performs once in a while. If counting overhead for files I/O, even total file transfer is the same, our algorithm will create less overhead.

Disadvantages

- Our algorithm is more complex in both structure and calculation than LRU, SIZE, and HYBRID. Space overhead of our algorithm is for storing host property and HP property for every object in cache list, plus space for storing unique host list. CPU overhead is for calculation of HP value and sorting list when eviction occurs. The complexity and overhead of our algorithm is on a par with GDSF and LFUDA.

Our algorithm can fit perfectly for web browsers on smart phones, because our algorithm is working well in low memory situation. Since EDGE and 3G were introduced, browsing internet via mobile devices, where memory is limited, is very common. Moreover, memory on smart phones is shared by its operating system, web browser, and other always-on application such as 3G or GPS, making memory a very precious resource. Normally, memory cache size of mobile web browsers is hidden. However, Ryan Grove from Yahoo Interface Blog[20] had performed a test on several popular smart phones to measure their internal memory cache size. The result is shown in Fig. 10.

As seen from example, popular smart phones such as Nexus ONE (Android 2.1), iPhone 3GS, and iPhone 4, all have internal memory cache size equal or less than 2 megabytes; the same number as our converge point in the experiment. Noted that those phones have 512, 256, 512 megabytes of total memory.

Model	Memory Cache size	Total memory
Nexus One (android2.1)	2MB	512MB
iPhone 3GS	1MB	256MB
iPhone 4	1.9MB	512MB

Fig. 10 Total memory and browsers cache of Nexus One (Android 2.1), iPhone 3GS, and iPhone 4, Ryan Grove.[20]

In many low-end Android based smart phones such as Samsung Galaxy Ace or HTC Wildfire, they contains only 158 and 384 megabytes of memory, respectively. In this case, by giving away only small portion of memory for web caching, up to 24% of objects can receive cache hit, thus save times for users. Moreover, smart phones' market now is very competitive about CPU speed while memory is mostly neglected. With previous mentioned reasons, our algorithm is very suitable for low-end smart phones.

§6 Conclusion

In this research, we have proposed a web cache replacement policy based on recency-based algorithm and users web usage data. Our algorithm can perform significantly better than baseline LRU algorithm in low cache storage environment and perform the same as LRU at higher cache storage environment. Our algorithm also generates less file I/O overhead at the cost of higher complexity and structure overhead.

We believe that this technique is suitable for web caching on mobile devices especially on low-end smart phones where memory is severely limited but computing power is sufficient.

References

1) Podlipnig, S., Bszrmenyi, L., "A Survey of Web Cache Replacement Strategies," *ACM Computing Surveys, 35, 4*, pp. 374-398, December 2003.
2) Mookerjee, V. S., Tan, Y., "Analysis of a Least Recently Used Cache Management Policy for Web Browsers," *Operations Research, Linthicum, 50, 2*, 2002, pp. 345-357.
3) Williams, S., Abrams, M., Standridge, C. R., Abdulla, G. and Fox, E. A., "Removal policies in network caches for World-Wide Web documents," in *Proc. of ACM SIGCOMM*, ACM Press, pp. 293-305, 1996.
4) Wong, K.Y., "Web Cache Replacement Policies: A Pragmatic Approach," *IEEE Network*, pp. 28-34, 2006.
5) Luotonen, A., Altis, K., "World-Wide Web proxies," *Computer Networks And ISDN System, 27, 2*, pp. 147-154, 1994.
6) Romano, S., ElAarag, H., "A quantitative study of recency and frequency based web cache replacement strategies," in *Proc. of the 11th communications and networking simulation symposium*, ACM, New York, 2008.
7) Tirdad, K., Pakzad, F., Abhari, A., "Cache replacement solutions by evolutionary computing technique," in *Proc. of the 2009 Spring Simulation Multiconference*, Society for Computer Simulation International, 2009.
8) Ali, W., Shamsuddin, S. M., "Intelligent Client-Side Web Caching Scheme Based on Least Recently Used Algorithm and Neuro-Fuzzy System," in *Proc. of the 6th International Symposium on Neural Networks: Advances in Neural Networks - Part II*, Springer-Verlag Berlin Heidelberg, 2009.
9) Torkzabah, V., Rahmani, S., "SCRAME: Selection of Cache Replacement Algorithm based on Multi Expert," in *Proc. of the 11th International Conference on Information Integration and Web-based Applications & Services*, ACM, New York, 2009.

10) Geetha, K., Gounden, N. A., Monikandan, S., “SEMALRU: An Implementation of modified web cache replacement algorithm,” *World Congress on Nature & Biologically Inspired Computing*, 2009.

11) Srivastava, J., Cooley, R., Deshpande, M., Tan, P., “Web Usage Mining: Discovery and Applications of Usage Patterns from Web Data,” *ACM SIGKDD Explorations Newsletter, 1, 2*, January, 2000.

12) Spiliopoulou, M., “Web usage mining for Web site evaluation,” *Communications of the ACM, 43, 8*, 2000.

13) Zhou, B., Hui, S. C., Fong, A. C. M., “Discovering and Visualizing Temporal-Based Web Access Behavior,” in *Proc. of the Web Inteligence 2005*, IEEE Computer Society, Washington, 2005.

14) http://ita.ee.lbl.gov/

15) http://www.web-caching.com

16) Lui, B., *Web Data Mining Exploring Hyperlinks, Contents, and Usage Data Second Edition*, Springer, July, 2011.

17) Cooley, R., Mobasher, B., Srivastava, J., “Data preparation for mining world wide web browsing patterns,” *Knowledge and Information systems*, 1999.

18) Khoury, R., Dawborn, T., Huang, W., “Visualising Web Browsing Data for User Behaviour Analysis,” in *Proc. of the 23rd Australian Computer-Human Interaction Conference*, ACM New York, NY, USA.

19) Goel, S., Hofman, J. M., Sirer, M. I., “Who Does What on the Web A Large-Scale Study of Browsing Behavior,” in *Proc. of ICWSM-12, International Conference on Weblogs and Social Media 2012*, pp. 4-6.

20) Grove R., “Mobile Browser Cache Limits: Android, iOS, and webOS,” http://www.yuiblog.com/blog/2010/06/28/mobile-browser-cache-limits, June 28th, 2010.

21) Jarukasemratana, S., Tsuyoshi, M., “Visualizing Web Structure based on Browsing Sessions,” in *Proc. of APCHI 2012*, The 10th Asia Pacific Conference on Computer Human Interaction, August, 2012.

22) Shi, L., Wei, L., Ye, H., Shi, Y., “Measurements of web caching and applications,” in *Proc. of the Fifth International Conference on Machine Learning and Cybernetics*, Dalian, 13-16 August 2006.

23) Judson, A., https://addons.mozilla.org/en-us/firefox/addon/tamper-data/

24) Cherkasova, L., “Improving Web Servers and Proxies Performance with GDSF Caching Policies,” Computer System Laboratory, Hewlett Packard, HPL-98-69 (R.1), November 1998.

25) http://www.squid-cache.org/

26) Cao, P., Wisconsin Web Cache Simulator, http://www.cs.wisc.edu/cao/, 1997.

Sorn Jarukasemratana: He is currently a Ph.D. student at Tokyo Institute of Technology under the supervision of Dr.Tsuyoshi Murata. He graduated his master degree from University of Tampere, Finland with M.Sc. in User Interface Software Development. His bachelor degree is obtained from faculty of engineering, Chulalongkorn University, Thailand with computer engineering major.

Tsuyoshi Murata, Dr.: He is an associate professor in the Department of Computer Science, Graduate School of Information Science and Engineering, Tokyo Institute of Technology. He obtained his doctor's degree in Computer Science at Tokyo Institute of Technology in 1997, on the topic of Machine Discovery of Geometrical Theorems. At Tokyo Institute of Technology, he conducts research on Web mining and social network analysis.