IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 17, 2018, accepted April 11, 2018, date of publication April 23, 2018, date of current version June 5, 2018.

*Digital Object Identifier 10.1109/ACCESS.2018.2829142*

# An Improved Web Cache Replacement Algorithm Based on Weighting and Cost

**TINGHUAI MA[1,2], (Member, IEEE), YU HAO[1], WENHAI SHEN[3], YUAN TIAN[4], AND MZNAH AL-RODHAAN[4]**

[1]School of Computer Software, Nanjing University of Information Science and Technology, Nanjing 210-044, China
[2]CICAEET, Jiangsu Engineering Centre of Network Monitoring, Nanjing University of Information Science and Technology, Nanjing 210-044, China
[3]National Meteorological Information Center, Beijing100-080, China
[4]Computer Science Department, College of Computer and Information Sciences, KingSaud University, Riyadh 11362, Saudi Arabia

Corresponding author: Tinghuai Ma (thma@nuist.edu.cn)

This work was supported in part by the Special Public Sector Research Program of China under Grant GYHY201506080, in part by the National Science Foundation of China under Grant 61572259 Grant U1736105, in part by the PAPD, and in part by the Research Center of the Female Scientific and Medical Colleges, Deanship of Scientific Research, King Saud University.

**ABSTRACT** Cache memory plays an important role in improving the performance of web servers, especially for big data transmission, which response time is constrained. It is necessary to use an effective method, such as web cache. Because an outstanding cache replacement algorithm can not only reduce the users access time but also improve the performance of the system. The traditional used weighting replacement policy does not consider the size parameter, hence, it may perform poorly while the datasets are larger. In this paper, we propose a novel, high-performance cache replacement algorithm for the web cache, named weighting size and cost replacement policy (WSCRP) bases on the weighting replacement policy. The algorithm recalculates the objects weight with adding the cost attribute in the cache, then orders the weight. Additionally the influence of various factors on the Web object as frequency, time, and cost value are considered. When the cache space cannot satisfy the new request object, the replacement policy WSCRP replaces the largest weighting and cost object. The experiments show that proposed algorithm has higher hit rate and byte rate for different datasets, and can effectively improve the performance of web cache.

**INDEX TERMS** Web cache, replacement, weighting, cost.

## I. INTRODUCTION

In recent years, the rapid increase of the World Wide Web service has caused an exponential rise in web traffic and access latency. To solve this problem, researchers have proposed many methods and the web caching is one of the most effective techniques [1]. In order to improve cache performance effectively, two kind of methods are proposed: One is increasing the cache size or decreasing latency time by prefetching, but its cost is very large. The other one is designing an useful cache replacement algorithm. Caching is a famous performance optimization method which is widely used in Web Caching. If there is no cache, users every request for data must be sent to servers. When the number of users becomes larger, servers load will increase. It will cause the network traffic and latency problem. So the replacement algorithm becomes more important.

The web cache servers are widely deployed in many places throughout the Web [1]. For web cache, there are three different web cache patterns: client-side caching [2], [3], server-side caching [4], and proxy caching [5]. Client-side caching refers to caches that user-side stores users web browsers page addresses and other information, and it can reach the specified server with the information. However, it is only for single user. Server-side caching refers to establishing the cache on the web server side. The purpose of it is decreasing the number of request for the server, so it can reduce the server load [6]. Proxy caching usually serves as the middle connection of user and center servers [7]. When the user sends a request to the server via a proxy server, server will response the data to the user according to the original request path. During this process, proxy server will decide whether to store a copy in its cache or not because this data may be requested in the future.

There are a lot of factors affecting the cache performance. The most important factors are accessing frequency, the last access time, cache block size and cache object size [12].

For each factor, researchers have proposed the corresponding algorithms. There are many cache replacement policies and all of them have their own algorithm to decide which object will be removed when the cache is full. The general goal of cache replacement policies is to improve the cache hit ratio and byte ratio [8]. Some researches show that a lot of data is only visited once and won't be visited again because of the huge amount of data access, thus we call this type data as One-Timers (OT) data. Similarly, we call data visited more than once as N-Timer(NT) data [7]. In the cache, we should try our best to store NT data as much as possible. In this research, we propose a low-latency, high-hit rate cache replacement policy for web cache to improve the performance of cache. We consider time, frequency size and cost, then design a new weight calculation method based on cost and give a new algorithm Weighting Size and Cost Replacement Policy (WSCRP). It can perform better than existing algorithms.

The rest of this paper is organized as follows. Section 2 reviews related researches and discuss web caching algorithms. Web cache replacement algorithm is discussed in section 3. In section 4, we present the results of this research, while section 5 provides the summary and conclusions.

## II. RELATED WORK

When a new object needs to be stored and the cache is full, a replacement algorithm must be used to determine which object should be removed to make space for the new objects [5]. Cache replacement algorithms decide which data could stay in the cache. An outstanding cache replacement algorithm has few number of misses in the cache and can improve the cache hit ratio and byte ratio too. A lot of academic and industry researchers are studying toward finding the cache replacement policy [9]. At present, cache replacement algorithms mainly focus on the following categories: Least Recently Used Algorithm, Least Frequently Used Algorithm, SIZE Algorithm, Function-based Algorithm, Randomized-based Algorithm and Weighting-based Algorithm [10].

(1) Least Recently Used Algorithm. In this type of algorithm, time is the most primary factor. Least Recently Used (LRU) algorithm is one of the most popular policies in web cache [10]. Its main idea is that if the data has been requested recently, the probability of it being requested in the future will be higher. It is used to cache web browser information, like pictures, etc. When cache is full, it removes the least recently referenced objects. The advantage of this algorithm is that it is easy to implement and has a good performance in client-side. But it only considers the last referenced time while ignoring the frequency of references for a certain web object [11].

(2) Least Frequently Used (LFU) Algorithm. In this type of algorithm, researchers consider the object popularity (or frequency count) as the primary factor [15]. Its main idea is that if the data has been requested more frequently, the probability of it being requested in the future will be higher. It usually caches the web URLs. When cache is full, it removes web object with the lowest frequency. Its advantage is that it is easy to implement too and has a good performance in proxy-side [13]. But if one object has high frequency previously, it will be stored in the cache although it will no longer be accessed. We call it "cache pollution".

(3) SIZE Algorithm. In this type of algorithm, object size is treated as the primary factor when caching the object [17]. Its main idea is that if the data has a bigger size, it may occupy more cache space. When cache is full, it removes web object with the biggest size. This type of algorithm usually caches the entity objects those with real object size rather than urls, etc. When the size is the same, it usually considers time as the second factor. Its advantage is that it can remove the big size objects timely. But for those objects with small size in the cache, they will be stored in the cache although they would not be accessed. Then it will also cause "cache pollution" [17].

(4) Function-based Algorithm. This type of cache replacement algorithm calculates the cache value of every web object [5]. For example, Greedy Dual-Size algorithm calculates the object size, access latency, cost and etc. When the cache is full, it removes the object according to the smallest cache value. The cache values is calculated as:

$$K_i = C_i/S_i + L \tag{1}$$

Where $C_i$ is the cost of retrieved object $i$, $S_i$ is the size of object $i$, $L$ is the aging factor. However, the method ignores the frequency and other factors.

(5) Randomized-based Algorithm. In this type of algorithm, no factor is treated as the primary factor when caching the object. Its main idea is that every cache objects has the equal value. When cache is full, it selects a random object and removes [14]. It is easy to implement, but it has low performance and its hit rate is unsatisfactory [16]. Because server doesnt need analyze the objects attribute, it just remove an object randomly [28]. But the objects are impossible to request an object only once, so it may remove the data that we call "hot data".

(6) Weighting-based Algorithm [18]. In this type of cache replacement algorithm, researchers use weight to decide which object should be removed from the cache. The weight is sorted in descending order. Users could directly remove the object with the highest weight value. The algorithm in [18] is named Weighting Replacement algorithm(WRP), its weighting value is calculated by equation (2).

$$W_i = \frac{L_i}{F_i \times \Delta T_i} \tag{2}$$

Where $L_i$ is the last time the data being accessed, $F_i$ is the frequency of object $i$, $\Delta T_i$ is the average value of every two consecutive access time.

The hit ratio is a standard to measure the performance of the web cache replacement algorithms [19]. The higher the hit ratio, the better the replacement policy will be [20]. With the development of the network bandwidth and cloud computing

technology, the speed of information transmission is increasing. So transport and network consumption cost has become a criteria to evaluate the cache replacement algorithms [21].

## III. WSCRP ALGORITHM

Our algorithm is based on weighting replacement algorithm. In addition, we use a new weight and add a new cost model. Our algorithm also uses the advantage of the existing cache replacement algorithm, which makes our algorithm more reliable and accurate. There are three main parts: weighting-cost model, the cache storage structure and cache replacement policy.

### A. WEIGHTING-COST MODEL

The original weighting replacement algorithm is based on LFU, which replaces pages that are not recently used and pages that are accessed only once [30]. It has advantages in caching for the URL and other small size objects. But for those objects who have bigger size, this algorithm may cause cache pollution. So we want to improve the cache hit ratio and propose an approach that can utilize cache space as much as possible and avoid cache pollution. We recalculate the weight of cache objects and add some new attributes like object size to the formula of weight. When a request arrives, we mark a hit and cresponses to client if the requested object is in cache. However, if the requested object is not in cache and the cache capacity is enough, we mark a miss and add the object, into the cache which is obtained from server database. If the requested object is not in cache, and the cache capacity is not enough, the cache must remove any objects to store the new object.

We improve the formula of weight by adding a new attribute to equation(2), the size of object. The formula is described in equation(3). Where $L_i$ is a parameter which means the recency of block($L$) $i$ in the cache. $F_i$ is a counter which means the number of times that object $i$ in the cache has been referenced and the $S_i$ is the size of object $i$. $\Delta T_i = T_{ci} - T_{pi}$ is a time difference, where $T_{ci}$ is the last accessed time of object $i$ and $T_{pi}$ is the penultimate accessed time.

$$W_i = \frac{L_i}{F_i \times \Delta T_i} \times S_i \tag{3}$$

When a new object $k$ is put in the cache buffer, we should set the original attribute value. For example, if $k$ is the first requested object, then $F_k$ is set to 1, $\Delta T_k$ is set to 1 and $L_k$ is set to 0 because $K$ is being used now. And we will calculate the objects size $S_k$. We consider the original $\delta T_k$ equals to 1 because the time of each reference to a block would be at least one in its minimum case [7]. For every time the cache buffer is accessed, if the request object $j$ is just in buffer cache, then a hit is occurred and we update the parameters as follows:

1) $L_i = L_i + 1$, for every $i \neq j$.
2) For $i = j$, we first let $\Delta T_i = L_i$, $F_j = F_j + 1$ and then $L_j = 0$.
3) Update the $W$.

In the process of network transmission and storage, saving network resource as much as possible is a goal we pursue all the time [26]. So we use a cost value as a parameter for judging the object is remove from cache or so. When all objects have been accessed, we will get a general network consumption, then we could calculate the the consumption of one object. If one object has higher replacement cost, we should cache it and save the cost. The cost-saving value ($csv$) is calculated by the following equation:

$$csv_i = c_i s_i h_i / \sum_{i=1}^{n} c_i s_i r_i \tag{4}$$

Where $c_i$ is the network traffic consumption of object $i$, $s_i$ is the size of object $i$, $h_i$ is how many valid copy of $i$ is found in the cache, and $r_i$ is the total times a data was requested. We obtain a new formula of weight as equation(5) by combining equation(3) with equation(4).

$$WC_i = \frac{L_i}{F_i \times \Delta T_i} \times \frac{s_i}{csv_i} \tag{5}$$

For an object, it should be store in the cache if its $csv$ is larger because if we replace it, we will spend more cost on caching them again. So we add $csv$ into the formula of weight to decide which object should be removed from cache when the cache is full. It could improve the cache performance.

For parameter $s_i$, big size data will occupy more cache space, if they become cache garbage, they could reduce the hit rate, byte hit date and increase average access time. So for cache, we priority to cache small data. For $csv_i$, if one datas replace cost is bigger than other, we could think it is more important. Our algorithm is replacement the data that with bigger weighting value [31], so we choose $s_i$ as numerator while $csv_i$ as denominator, combined to form the formula (5).

According to the above introductions, we propose our algorithm WSCRP algorithm. Every time we access data from cache, we should recalculate the $WC_i$. $WC_i$ is calculated by the equation(5). The algorithm 1 is the process of WSCRP.

**Algorithm 1** WSCRP

```
Calculate the size of free cache space and mark as fs
Calculate the size of requested object ro and mark as rs
while fs < rs
  if there is only one maximum WC
    remove the object with maximum WC
  else
    remove an object with maximum WC and minimum
frequency value
Update all caching objectss WC value
end
Update WC table
```

### B. CACHE STORAGE STRUCTURE

The key of our algorithm is to keep the most valuable object in the cache. Table 1 describes the information of

**TABLE 1.** Cache framework table.

| id | oid | $\Delta T_i$ | $L_i$ | $S_i$ | $F_i$ | $h_i$ | $c_i$ | $WC_i$ |
|---|---|---|---|---|---|---|---|---|
| 4 | 1 | $\Delta T_4$ | $L_4$ | $S_4$ | $F_4$ | $h_4$ | $c_4$ | $WC_4$ |
| 6 | 2 | $\Delta T_6$ | $L_6$ | $S_6$ | $F_6$ | $h_6$ | $c_6$ | $WC_6$ |
| 1 | 3 | $\Delta T_1$ | $L_1$ | $S_1$ | $F_1$ | $h_1$ | $c_1$ | $WC_1$ |
| 7 | 4 | $\Delta T_7$ | $L_7$ | $S_7$ | $F_7$ | $h_7$ | $c_7$ | $WC_7$ |
| 3 | 5 | $\Delta T_3$ | $L_3$ | $S_3$ | $F_3$ | $h_3$ | $c_3$ | $WC_3$ |
| 5 | 6 | $\Delta T_5$ | $L_5$ | $S_5$ | $F_5$ | $h_5$ | $c_5$ | $WC_5$ |
| 2 | 7 | $\Delta T_2$ | $L_2$ | $S_2$ | $F_2$ | $h_2$ | $c_2$ | $WC_2$ |

objects in cache. Every object has eight attributes, which are object *id* in the cache, the object *oid* in the database server, the time difference between the last access time and time of penultimate($\Delta T_i$), the recency of object $i(L_i)$, the size of object $i(S_i)$, Frequency($F_i$), the number of how many valid copy of object $i(h_i)$ after the object stores in the cache, the traffic cost of transfer object i($c_i$) and objects $WC_i$, $WC_i$ is the weighting-cost value of cache object $i$, the computing method has been introduced in 3.1.

In Cache Structure Table, the objects in table are ordered by $WC$. The *id* represents the order of objects enter the Cache. When a request arrives at the cache, if the requested object $i$ exists in the cache, we update the $F_i$ and other attributes, meanwhile, recalculate $WC_i$ and reset the cache data in ascending order of $WC_i$. We would reserve the objects with smaller $WC$ and remove the objects with larger $WC$.

If an objects $WC$ is larger than any others, we think this object is the most unimportant object and this object will be removed first when cache is full. If there are two objects with the maximum $WC$, we use $F_i$ as the second attribute to decide which object should be removed from the cache.

Comparing with original cache replacement algorithm based on weighting, our algorithm considers more factors which could influence the cache performance. So our algorithm is more complicated.

### C. CACHE REPLACEMENT POLICY

Our algorithm contains two parts: calculating the weighting-cost value and replacing the cache items. Weighting-cost has been introduced in 3.1, now we introduce the replacement algorithm 2 ICRP.

**Algorithm 2** ICRP

```
if request object ro in cache
  Find the data item corresponding to the object ro in table1
  Update table 1
else
  Use WSCRP to deal with the request
  Cache the object
  Update table 1
end
Update WC table
```

In order to maximize the hit ratio and improve performance, the data in the cache should be the one the user most likely to request [27]. Nowadays, network transmission is the main channel. Web cache is distributed in each node of the network. During the process of data transmission, when the system receives the request about object $i$, it will check the web cache node firstly and check whether the object $i$ exists or not. If the object is in the cache, it returns the data to user directly. If it is not exist, system returns the data to user from server and adds the object into cache then our algorithm will use the weighting-cost policy to calculate object $i's$ weighting-cost value and update the Cache Structure Table by this value in ascending order. If there is not enough space to store object, we will remove object according to Algorithm 2, ICRP(Improved Cache replacement Policy), until the space is enough to store the new data.

## IV. EXPERIMENT

In this section, we perform some experiments to evaluate our proposed algorithm. We simulate our policy and compare it with other policies like LRU, LFU and the original weighting policy WRP. Firstly, we introduce the benchmark dataset used in our experiment and the implement platform of our algorithm. Then we explain the performance metrics in cache. Finally, we show the experiment results of the comparative experiments and analyze them. Our platform is consisting of client-side and server-side.

### A. DATASET AND EXPERIMENT CONFIGURATION

In our experiment, we used 50000000 meteorological data records (about 4T) from China Meteorological Data Sharing Service System(CMDSSS) [22]. The data records contain six months meteorological data. According to the charactors of data, we divided these meteorological data into three data types. In these data records some are request UR, some are entity objects and some are mixture. We call them UTD(Url test data), ETD(Entity test Data) and MTD(Mix test Data), respectively. Each type has many datasets, and all of them are used in our algorithm. Our experimental environment is based on a Dell server, with the 64G RAM, running Linux system. In order to simulate the network environment, the platform of our program contains server side and client side. We run our WSCRP algorithm and then compare the algorithm with LRU, LFU, GDSF and WRP in our platform. The algorithm is implemented by Java.

### B. EXPERIMENT METRICS

In experiment, we mainly use three metrics -cache hit rate, byte hit rate and average access latency, to evaluate our algorithm performance.

#### 1) CACHE HIT RATE

Hit Rate (HR) is one of the most important metrics used in performance evaluation criteria. Cache hit rate is computed according to the following equation:

$$HR = \sum_{i=1}^{N} r_i / N \tag{6}$$

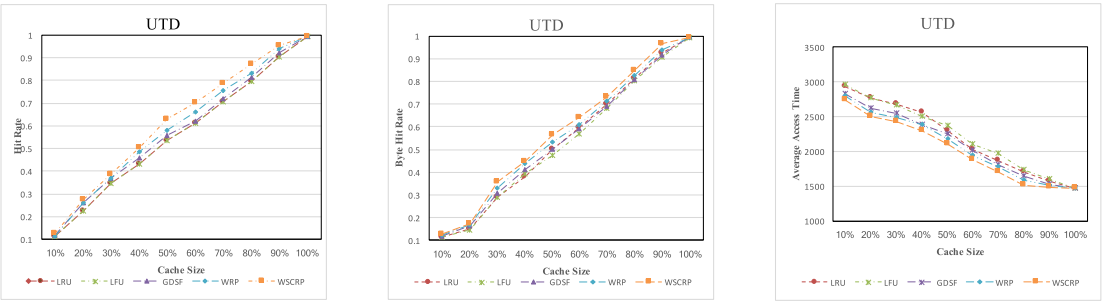


**FIGURE 1.** Result of DataSet UTD.

**TABLE 2.** Result of DataSet UTD.

| cache size | Hit Rate | | | | | Byte Hit Rate | | | | | Average Access Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LRU | LFU | GDSF | WRP | WSCRP | LRU | LFU | GDSF | WRP | WSCRP | LRU | LFU | GDSF | WRP | WSCRP |
| 10% | 0.1156 | 0.115 | 0.117 | 0.115 | 0.127 | 0.115 | 0.115 | 0.118 | 0.123 | 0.128 | 2931 | 2961 | 2832 | 2791 | 2736 |
| 20% | 0.254 | 0.226 | 0.263 | 0.26 | 0.276 | 0.15 | 0.147 | 0.164 | 0.168 | 0.173 | 2772 | 2768 | 2613 | 2562 | 2502 |
| 30% | 0.361 | 0.3481 | 0.369 | 0.372 | 0.387 | 0.292 | 0.29 | 0.31 | 0.33 | 0.36 | 2684 | 2669 | 2548 | 2481 | 2427 |
| 40% | 0.447 | 0.432 | 0.461 | 0.486 | 0.505 | 0.379 | 0.394 | 0.414 | 0.44 | 0.45 | 2559 | 2496 | 2387 | 2377 | 2294 |
| 50% | 0.526 | 0.537 | 0.56 | 0.582 | 0.63 | 0.504 | 0.474 | 0.501 | 0.534 | 0.567 | 2289 | 2364 | 2254 | 2174 | 2106 |
| 60% | 0.606 | 0.613 | 0.622 | 0.664 | 0.704 | 0.597 | 0.572 | 0.594 | 0.612 | 0.644 | 2024 | 2105 | 2008 | 1936 | 1884 |
| 70% | 0.694 | 0.705 | 0.721 | 0.755 | 0.786 | 0.701 | 0.684 | 0.691 | 0.715 | 0.734 | 1867 | 1964 | 1805 | 1784 | 1706 |
| 80% | 0.788 | 0.797 | 0.813 | 0.831 | 0.872 | 0.813 | 0.802 | 0.808 | 0.828 | 0.847 | 1705 | 1738 | 1648 | 1587 | 1513 |
| 90% | 0.891 | 0.903 | 0.924 | 0.942 | 0.956 | 0.927 | 0.906 | 0.918 | 0.941 | 0.968 | 1572 | 1604 | 1528 | 1506 | 1428 |
| 100% | 0.994 | 0.994 | 0.994 | 0.994 | 0.994 | 0.992 | 0.994 | 0.996 | 0.992 | 0.993 | 1464 | 1467 | 1464 | 1465 | 1465 |

Where $r_i$ is a state of the object. It is a boolean value, where '0' represents miss and '1' represents hit.$N$ is the sum of requests. When all requests are finished, we could count the hit times by summing up all $r_i$ value.

### 2) CACHE BYTE HIT RATE

Byte Hit Rate (BHR) is another most important metric used in evaluating performance. Cache hit rate can be described by the following equation:

$$BHR = \sum_{i=1}^{N} r_i s_i / \sum_{i=1}^{N} S_i \tag{7}$$

Where $r_i$ has the same meaning as that described in Hit Rate. The parameter $s_i$ is the size of object $i$. We record the size of each cache object. When all requests are finished, we can calculate the total object size, and then use formula (6) to calculate the value of BHR.

### 3) AVERAGE ACCESS LATENCY

The AAL's(Average Access Latency) value represents the average access time during the whole dataset. AAL can be described by the following equation:

$$AAL = \sum_{i=1}^{N} T_i / SUM \tag{8}$$

Where the parameter $T_i$ is the time when the program finished accessing dataset $i$ and the parameter $SUM$ is the number of datasets. The AAL is smaller, the performance of the cache replacement algorithm is better. It means users can get objects they requested without waiting for a long time.

## C. EXPERIMENT RESULTS

Firstly, we test the performance of our algorithm and some other algorithms as LRU [11], LFU [24], WRP [18] and GDSF [25] in UTD data. Both data are URLs. LRU, LFU and WRP we have been introduced in section 2, GDSF is a function based algorithm based on size and frequency. We use percentage to show the size of the cache, where 10% means cache size is 10 percentage of the total cache size. For every dataset, we sent 10000 requests. Each dataset will generate a result of the three metrics in different cache size, and we use the average results of them.

Figure 1 Result of DataSet UTD shows the information of hit ratio, byte ratio and average access time. It's a graphic display of Table 2. When the cache is 10% all algorithms performance are not obvious. But with the increase of cache size, we can see our algorithm has better performance than others in hit rate, byte hit rate and average access time. When cache size arrive 100% while the hit rate and bite hit rate are not 100%, we believe that this is due to the fact that the initial cache is empty and it leads a number of miss. We can also see that the performance of hit rate of LRU is better than LFU when cache size is less than 50%. When cache size is larger than 50%, LFU has better performance in hite rate. We think LRU has certain advantages in the page cache or some small data size cache, and the dataset UTD is URL format. And In practical application, cache size rarely more than 50%, so if

**TABLE 3.** Result of DataSet ETD.

| cache size | Hit Rate | | | | | Byte Hit Rate | | | | | Average Access Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LRU | LFU | GDSF | WRP | WSCRP | LRU | LFU | GDSF | WRP | WSCRP | LRU | LFU | GDSF | WRP | WSCRP |
| 10% | 0.123 | 0.136 | 0.12 | 0.13 | 0.13 | 0.152 | 0.16 | 0.147 | 0.169 | 0.174 | 3943 | 3872 | 3987 | 3926 | 3903 |
| 20% | 0.218 | 0.224 | 0.252 | 0.286 | 0.297 | 0.25 | 0.62 | 0.28 | 0.306 | 0.327 | 3642 | 3595 | 3510 | 3453 | 3404 |
| 30% | 0.337 | 0.325 | 0.375 | 0.398 | 0.429 | 0.333 | 0.34 | 0.362 | 0.384 | 0.407 | 3269 | 3306 | 3212 | 3128 | 3018 |
| 40% | 0.421 | 0.43 | 0.46 | 0.494 | 0.522 | 0.426 | 0.43 | 0.46 | 0.492 | 0.53 | 2938 | 2977 | 2820 | 2714 | 2614 |
| 50% | 0.506 | 0.51 | 0.53 | 0.58 | 0.62 | 0.515 | 0.528 | 0.555 | 0.581 | 0.63 | 2694 | 2731 | 2570 | 2486 | 2376 |
| 60% | 0.592 | 0.601 | 0.624 | 0.645 | 0.691 | 0.631 | 0.645 | 0.667 | 0.694 | 0.716 | 2336 | 2385 | 2246 | 2132 | 2034 |
| 70% | 0.704 | 0.728 | 0.738 | 0.761 | 0.785 | 0.706 | 0.718 | 0.746 | 0.768 | 0.794 | 2182 | 2228 | 2095 | 2007 | 1994 |
| 80% | 0.821 | 0.83 | 0.845 | 0.857 | 0.871 | 0.795 | 0.804 | 0.824 | 0.846 | 0.877 | 2028 | 2084 | 1975 | 1886 | 1804 |
| 90% | 0.924 | 0.935 | 0.937 | 0.942 | 0.95 | 0.914 | 0.922 | 0.936 | 0.944 | 0.954 | 1805 | 1829 | 1785 | 1762 | 1704 |
| 100% | 0.993 | 0.993 | 0.993 | 0.993 | 0.993 | 0.993 | 0.994 | 0.993 | 0.994 | 0.994 | 1665 | 1658 | 1664 | 1667 | 1660 |

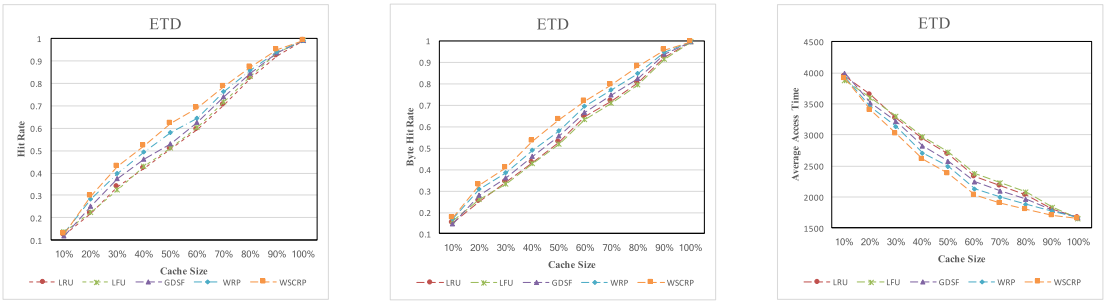


**FIGURE 2.** Result of DataSet ETD.

the cache object is page or url we should choose LRU first if there are only LRU and LFU we can use. LRU and LFU are only use one parameter as the judgment standard, while GDSF and WRP are function based algorithms, so they have better performance than LRU and LFU. GDSF algorithm only consider parameters like frequency, time and size, while WRP add weighting value in the algorithm, so WRP has the better performance than GDSF. Our algorithm is based on the weight and join the cost analysis, so it has more advantages than other algorithms. Average access time is very smaller with hit rate, because if hit rate is high, it means clients would not wait for a long time.

Secondly, we use ETD to test our algorithm and keep other conditions unchanged. This type data is mainly entity objects, so our cache must store the real object. Because objects have bigger size than URL, cache performance is also different from the first type. Figure 2 Result of DataSet ETD shows the hit ratio, byte ratio and average access time of different cache size in this type data.

Figure 2 Result of DataSet ETD shows the results. It's a graphic display of Table 3. It is clearly in Figure 2 that our WSCRP policy has better performance than LRU, LFU and WRP. Our hit rate is 11.4% higher than LRU when cache size is the 50%. Because in 50 percentage, algorithms have the biggest difference value. Compared to UTD, this test dataset has many differences in byte hit rate and average access latency because this dataset is entity data and accessing massive data need much more time, so average access time in Figure 2 is larger than average access time in Figure 1. We can also see LFU has better performance than LRU int hit rate and byte hit rate, it can confirm that LRU are more useful in caching pages. GDSFs main parameter is size, so it has better performance than LRU and LFU. While WRP and our WSCRP both contain parameter size and they have more better improve so they have better performance than others.

Finally, we used the MTD for the final test and keep other conditions unchanged. This dataset is mixed of Urls and entity data. Although our algorithm has similar performance to other algorithms when the cache size is small, our algorithm is better than them in overall performance. Fig 3 shows the hit ratio, byte ratio and average access time of different cache size in this type data.

As Figure 3 Result of DataSet MTD shows, when the cache size is 50%, the hit rate of our algorithm is almost 7.1% higher than LRU. When the cache size is very small, it is close to no cache, therefore all algorithms have almost the same performance. Because this dataset is a mix of UTD and ETD, so the performance of all algorithm is between UTD and ETD. Figure 3 is a graphic display of Table 4.

From the experiments results, we can see when the cache size is 10% and 100%, the performance of all algorithms is almost the same. We believe when the cache size is very small like 10%, cache block can only cache several files and the cache sequence is short, but we have a large number of requests, so the cache performance of the algorithms are not high. But when the cache size is 100%, it can cache all files, so when different files are cached, the cache hit rate is almost 100%. When cache size arrive 100% while the hit rate

**TABLE 4.** Result of DataSet MTD.

| cache size | Hit Rate | | | | | Byte Hit Rate | | | | | Average Access Time | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | LRU | LFU | GDSF | WRP | WSCRP | LRU | LFU | GDSF | WRP | WSCRP | LRU | LFU | GDSF | WRP | WSCRP |
| 10% | 0.178 | 0.169 | 0.168 | 0.185 | 0.185 | 0.203 | 0.196 | 0.191 | 0.215 | 0.224 | 3628 | 3684 | 3710 | 3629 | 3613 |
| 20% | 0.28 | 0.299 | 0.302 | 0.318 | 0.336 | 0.293 | 0.305 | 0.317 | 0.325 | 0.333 | 3226 | 3285 | 3182 | 3161 | 3102 |
| 30% | 0.395 | 0.391 | 0.407 | 0.415 | 0.436 | 0.382 | 0.394 | 0.418 | 0.431 | 0.442 | 3115 | 3169 | 3061 | 3005 | 2918 |
| 40% | 0.482 | 0.48 | 0.492 | 0.505 | 0.537 | 0.477 | 0.482 | 0.495 | 0.512 | 0.528 | 2852 | 2901 | 2802 | 2728 | 2672 |
| 50% | 0.558 | 0.582 | 0.594 | 0.605 | 0.629 | 0.528 | 0.544 | 0.562 | 0.575 | 0.592 | 2606 | 2650 | 2568 | 2486 | 2314 |
| 60% | 0.654 | 0.662 | 0.671 | 0.689 | 0.703 | 0.615 | 0.627 | 0.641 | 0.655 | 0.669 | 2258 | 2316 | 2191 | 2105 | 2001 |
| 70% | 0.752 | 0.764 | 0.771 | 0.785 | 0.795 | 0.708 | 0.714 | 0.722 | 0.735 | 0.744 | 2001 | 2068 | 2028 | 1955 | 1910 |
| 80% | 0.861 | 0.872 | 0.884 | 0.897 | 0.901 | 0.811 | 0.821 | 0.835 | 0.844 | 0.858 | 1825 | 1860 | 1740 | 1658 | 1568 |
| 90% | 0.93 | 0.938 | 0.94 | 0.948 | 0.952 | 0.915 | 0.921 | 0.93 | 0.941 | 0.95 | 1731 | 1784 | 1639 | 1608 | 1601 |
| 100% | 0.993 | 0.993 | 0.993 | 0.993 | 0.993 | 0.993 | 0.994 | 0.993 | 0.995 | 0.993 | 1582 | 1601 | 1595 | 1588 | 1583 |

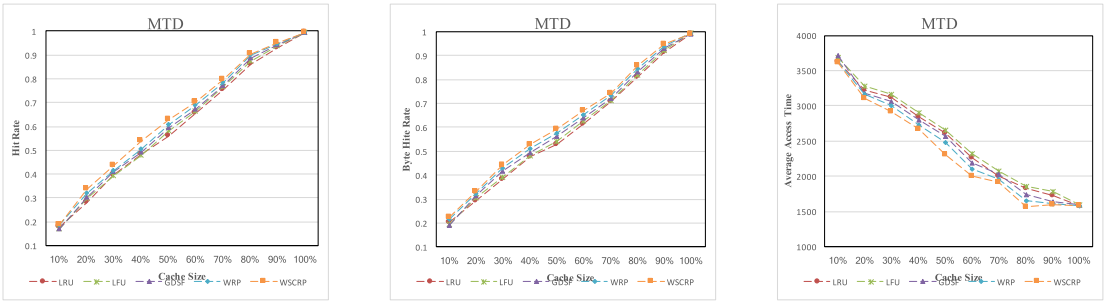


**FIGURE 3.** Result of DataSet MTD.

and byte hit rate are not 100%, we believe that this is due to the fact that the initial cache is empty and it leads a number of miss.

From the experiments results, we can also see that dataset type and cache size can also influence the cache performance. For different data sets, the same algorithm also has different results. For cache size, as it increases, all algorithms performance will be better than before. But we find when cache size is 50%, algorithms have the biggest different value, so we think cache size is not greater than 50%, because the cost of cache is larger. Our algorithm can be used in different applications, then we can continue to compare the performance with LRU, LFU, WRP,GDSF and other replacement policy.

Through the experiment we can see that our algorithm is not only suitable for caching simple URLs, but also can cache entity data. This is because our algorithm contains both parameters of other algorithm and add the cost analyze. LRU has better performance in caching pages and small data, if we only have LRU and LFU, we should use them reasonable in different application. Experiment can also prove our algorithm can not only be used in the network nodes, but also can be used in the server to cache data entity.

## V. CONCLUSIONS

In this paper, we propose a new web cache replacement policy WSCRP in order to improve the hit rate ratio when cache memory is limited. Our proposed replacement policy takes the impacts of weight and cost into consideration. We compare our proposed algorithm with other algorithms such as LRU, LFU and WRP replacement policy in hit rate, byte hit rate and average access time. Our experimental results indicate that the proposed replacement policy can increase the hit rate and is beneficial for the limited cache memory.

We believe that our algorithm must have a good performance in web caching, especially in big data web site. Now there are still some other factors can affect the cache efficiency, so we will continue our research to make it better in web caching.

## REFERENCES

[1] D. Singh, S. Kumar, and S. Kapoor, "An explore view of Web caching techniques," *Int. J. Adv. Eng. Sci.*, vol. 1, no. 3, pp. 38–43, 2011.
[2] S. M. Shamsuddin and W. A. Ahmed, "Integration of least recently used algorithm and neuro-fuzzy system into client-side Web caching," *Int. J. Comput. Sci. Secur.*, vol. 3, no. 1, pp. 1–15, 2009.
[3] W.-G. Teng, C.-Y. Chang, and M.-S. Chen, "Integrating Web caching and Web prefetching in client-side proxies," *IEEE Trans. Parallel Distrib. Syst.*, vol. 16, no. 5, pp. 444–455, May 2005.
[4] K. Kim and D. Park, "Reducing outgoing traffic of proxy cache by using client-cluster," *J. Commun. Netw.*, vol. 8, no. 8, pp. 330–338, 2006.
[5] X. Wu, H. Xu, X. Zhu, and W. Li, "Web cache replacement strategy based on reference degree," in *Proc. IEEE Int. Conf. Smart City/SocialCom/SostainCom*, Chengdu, China, Dec. 2015, pp. 209–212.
[6] S. Hiranpongsin and P. Bhattarakosol, "Integration of recommender system for Web cache management," *Maejo Int. J. Sci. Technol.*, vol. 7, no. 2, pp. 232–247, 2013.
[7] T. Ma *et al.*, "*KDVEM*: A *k*-degree anonymity with vertex and edge modification algorithm," *Computing*, vol. 70, no. 6, pp. 1336–1344, 2015.
[8] S. Jin and A. Bestavros, "Popularity-aware greedy dual-size Web proxy caching algorithms," in *Proc. Int. Conf. Distrib. Comput. Syst.*, 2000. pp. 254–261, doi: 10.1109/ICDCS.2000.840936.
[9] A. Jain and C. Lin, "Back to the future: Leveraging Belady's algorithm for improved cache replacement," in *Proc. ACM/IEEE 43rd Annu. Int. Symp. Comput. Archit. (ISCA)*, Jun. 2016, pp. 78–89.

[10] T. S. Warrier, B. Anupama, and M. Mutyam, "An application-aware cache replacement policy for last-level caches," in *Architecture of Computing Systems—ARCS*. Berlin, Germany: Springer, 2013, pp. 207–219.
[11] S. Pashmforoush, S. Pashmforoush, A. Akbari, M. Bagheri, and N. Beikmahdavi, "Presenting a novel page replacement algorithm based on LRU," *J. Basic Appl. Sci. Res.*, vol. 2, no. 10, pp. 377–383, 2012.
[12] J. P. Sheu and Y. C. Chuo, "Wildcard rules caching and cache replacement algorithms in software-defined networking," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 1, pp. 19–29, Mar. 2016.
[13] T. Ma *et al.*, "LED: A fast overlapping communities detection algorithm based on structural clustering," *Neurocomputing*, vol. 207, pp. 488–500, Sep. 2016.
[14] W. Meizhen, S. Yanlei, and T. Yue, "The design and implementation of LRU-based Web cache," in *Proc. 8th Int. ICST Conf. Commun. Netw. China (CHINACOM)*, Guilin, China, Aug. 2013, pp. 400–404.
[15] J. S. Kushwah, J. K. Gupta, and B. Patel, "Modified LRU algorithm to implement proxy server with caching policies," *Int. J. Comput. Trends Technol.*, vol. 3, no. 10, pp. 337–343, 2011.
[16] W. Kin-Yeung, "Web cache replacement policies: A pragmatic approach," *IEEE Netw.*, vol. 20, no. 1, pp. 28–34, Jan./Feb. 2006.
[17] W. Ali, S. M. Shamsuddin, and A. S. Ismail, "A survey of Web caching and prefetching," *Int. J. Adv. Soft Comput. Appl.*, vol. 3, no. 1, pp. 18–44, 2011.
[18] K. Samiee, "A replacement algorithm based on weighting and ranking cache objects," *Int. J. Hybrid Inf. Technol.*, vol. 2, no. 2, pp. 93–104, 2009.
[19] N. Kamiyama, Y. Nakano, and K. Shiomoto, "Cache replacement based on distance to origin servers," *IEEE Trans. Netw. Service Manage.*, vol. 13, no. 4, pp. 848–859, Dec. 2016.
[20] Q. Li, X. Liao, H. Jin, L. Lin, X. Xie, and Q. Yao, "Cost-effective hybrid replacement strategy for SSD in Web cache," in *Proc. IEEE Int. Conf. Comput. Inf. Technol., Ubiquitous Comput. Commun. Depend., Auto. Secure Comput., Pervasive Intell. Comput. (CIT/IUCC/DASC/PICOM)*, Liverpool, U.K., Oct. 2015, pp. 1286–1294.
[21] Y. Lv *et al.*, "An efficient and scalable density-based clustering algorithm for datasets with complex structures," *Neurocomputing*, vol. 171, pp. 9–22, Jan. 2016.
[22] CMDSSS. *China Meteorological Data Service Share System (CMDSSS)*. Accessed: Apr. 4, 2018. [Online]. Available: http://cdc.nmic.cn/home.do
[23] C. F. Kao and C. N. Lee, "Aggregate profit-based caching replacement algorithms for streaming media transcoding proxy systems," *IEEE Trans. Multimedia*, vol. 9, no. 2, pp. 221–230, Feb. 2007.
[24] Tanwir, G. Hendrantoro, and A. Affandi, "Early result from adaptive combination of LRU, LFU and FIFO to improve cache server performance in telecommunication network," in *Proc. Int. Seminar Intell. Technol. Appl.*, 2015, pp. 429–432, doi: 10.1109/ISITIA.2015.7220019.
[25] S. Jarukasemratana and T. Murata, "Web caching replacement algorithm based on Web usage data," *New Generat. Comput.*, vol. 31, no. 4, pp. 311–329, 2013.
[26] A. Sarhan, A. M. Elmogy, and S. M. Ali, "New Web cache replacement approaches based on internal requests factor," in *Proc. 9th Int. Conf. Comput. Eng. Syst. (ICCES)*, Dec. 2014, pp. 383–389.
[27] H. Rong, T. Ma, M. Tang, and J. Cao, "A novel subgraph $K^+$ -isomorphism method in social network based on graph similarity detection," *Soft Comput.*, vol. 22, no. 8, pp. 2583–2601, 2018.
[28] J.-Q. Niu, H.-R. Zheng, H. Li, and X.-F. Wang, "Limited history based multi-LRU Web cache replacement algorithm," *J. Chin. Comput. Syst.*, vol. 29, no. 6, pp. 1010–1014, 2008.
[29] T. Ma, H. Rong, C. Ying, Y. Tian, A. Al-Dhelaan, and M. Al-Rodhaan, "Detect structural-connected communities based on BSCHEF in C-DBLP," *Concurrency Comput., Pract. Exper.*, vol. 28, no. 2, pp. 311–330, 2016.
[30] T. Banditwattanawong, "From Web cache to cloud cache," in *Advances in Grid and Pervasive Computing*. Berlin, Germany: Springer, 2012, pp. 1–15.

**TINGHUAI MA** received the bachelor's and master's degrees from the Huazhong University of Science and Technology, China, in 1997 and 2000, respectively, and the Ph.D. degree from the Chinese Academy of Sciences in 2003. He was a Post-Doctoral Associate with Ajou University in 2004. From 2007 to 2008, he visited Chinese Meteorology Administration. In 2009, he was a Visiting Professor with the Ubiquitous Computing Laboratory, Kyung Hee University. He is currently a Professor of computer sciences with the Nanjing University of Information Science and Technology, China. He has authored or co-authored over 100 journal/conference papers. His research interests include data mining, cloud computing, ubiquitous computing, and privacy preserving.

**YU HAO** received the bachelor's degree in computer science and technology from the Nanjing University of Information Science and Technology, China, in 2014, where he is currently pursuing the degree in software engineering. His research interests include Web cache replacement algorithm.

**WENHAI SHEN** received the bachelor's degree in mathematics from Fudan University, China, in 1982. He was with the Chinese Academy of Meteorological Sciences, National Climate Center, National Weather Center, for over 30 years. He is currently a Professor with the National Meteorological Information Center. His research interests include meteorological information design and application.

**YUAN TIAN** received the master's and Ph.D. degrees from Kyung Hee University. She is currently an Assistant Professor with the College of Computer and Information Sciences, King Saud University, Saudi Arabia. Her research interests are broadly divided into privacy and security, which are related to cloud computing, bioinformatics, multimedia, cryptograph, smart environment, and big data. She is a member of technical committees for several international conferences. She is an active reviewer of many international journals.

**MZNAH AL-RODHAAN** received the B.S. degree (Hons.) in computer applications and the M.S. degree in computer science from King Saud University, Riyadh, Saudi Arabia, in 1999 and 2003, respectively, and the Ph.D. degree in computer science from the University of Glasgow, Scotland, U.K., in 2009. She is currently the Vice Chair with the Computer Science Department, College of Computer and Information Sciences, King Saud University. She has participated in several international conferences. Her current research interests include mobile ad hoc networks, wireless sensor networks, multimedia sensor networks, cognitive networks, and network security. She has served on the editorial boards for some journals, such as the *Ad Hoc* journal (Elsevier).