Cache Replacement for Transcoding Proxy Caching

Keqiu Li, Keishi Tajima, and Hong Shen
Graduate School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa, 923-1292, Japan

Abstract

In this paper, we address the problem of cache replacement for transcoding proxy caching. First, an efficient cache replacement algorithm is proposed. Our algorithm considers both the aggregate effect of caching multiple versions of the same multimedia object and cache consistency. Second, a complexity analysis is presented to show the efficiency of our algorithm. Finally, some preliminary simulation experiments are conducted to compare the performance of our algorithm with some existing algorithms. The results show that our algorithm outperforms others in terms of the various performance metrics.

Key words: Transcoding, proxy caching, cache replacement, multimedia, Internet.

1. Introduction

Web caching [21] has become one of the most important technologies for improving network and system performance by reducing user access latency, network traffic and server load. To achieve such improvement, many techniques have been utilized, such as cache replacement, object placement, caching architecture, proxy placement, caching protocol. In this paper, we focus on the problem of cache replacement which is generated due to the limited cache space. If the cache size is large enough, this problem becomes trivial since all objects can be stored in the cache such that the total access cost is minimized. As a result, cache replacement algorithms [1, 17] are used to determine a suitable subset of web objects to be removed from the cache to make room for a new web object. An overview of web caching replacement algorithms can be found in [2]. However, the improvement of network performance, such as access latency reduction achieved by caching web objects, does not come completely for free. maintaining cache content consistence will also introduce additional overhead. Consistency algorithms [3, 5, 13] are widely applied by many proxy cache implementations to ensure a suitable form of consistency for the cached objects.

Transcoding is used to transform a multimedia object from one form to another, such as a different format or different resolution, to adapt the object to the constraints at the clients. It is most often used to convert video formats (i.e., Beta to VHS, VHS to QuickTime, QuickTime to MPEG), but it is also used to fit HTML files and graphics files to the hardware constraints of mobile devices and other Web-enabled products. These devices usually have smaller screen sizes, smaller memory, and slower bandwidth rates. In such cases, transcoding is performed by a transcoding proxy server, which receives the requested document or file and transcode it in accordance with the constraints at the client.

Since transcoding proxies play an important role in the functionality of web caching, transcoding proxy caching is attracting more and more attention [6,7,10,12,15,18]. However, existing cache replacement algorithms cannot be simply applied to transcoding proxy caching because of the new emerging factors in the environment of transcoding proxies, such as the additional delay caused by transcoding, different sizes and different reference rates for different versions of a multimedia object, and the aggregate effect of the profits of caching multiple versions of the same multimedia object. In [8], the authors proposed an efficient cache replacement algorithm for transcoding proxies, AE in short, which selects objects to remove from the cache based on their generalized profit function one by one. When one object is removed from the cache, the generalized profits for the relevant objects will be revised. If the free space cannot accommodate the new object, another object with the least generalized profit is removed until enough room is made for the new object. However, this method is not optimal when there is more than one object to be removed, as shown in the following example.

Example Suppose that there are two objects and each object has three versions, i.e., the object set is $\{o_{1,1}, o_{1,2}, o_{1,3}, o_{2,1}, o_{2,2}, o_{2,3}\}$, where $o_{i,j}$ denotes version j of object i. The size set of all the objects is

$\{3, 2, 1, 3, 2, 1\}$. We also assume that the generalized profits of caching one or two versions of each object are shown in Table 1. For example, the generalized profit of caching $o_{1,2}$ is 20, and the generalized profit of caching $o_{2,1}$ and $o_{2,3}$ is 30.

$o_{1,1}$	$o_{1,2}$	$o_{1,3}$	$o_{2,1}$	$o_{2,2}$	$o_{2,3}$
18	20	16	16	18	18
$o_{1,1}, o_{1,2}$	$o_{1,1}, o_{1,3}$	$o_{1,2}, o_{1,3}$	$o_{2,1}, o_{2,2}$	$o_{2,1}, o_{2,3}$	$o_{2,2}, o_{2,3}$
29	25	28	26	30	28

Table 1. Generalized Profit

If an object with size 2 is to be inserted, it is obvious that object $o_{2,1}$ should be removed because it has the least generalized profit, and its size is enough to accommodate the new object. In this case, AE is efficient. If an object with size 4 is to be inserted, AE will first remove object $o_{2,1}$ from the cache, and then remove $o_{1,3}$ from the cache because these two are the ones with the least profits, and their total size is enough to accommodate the new object. The lost profit by removing these two objects is 32. We can see AE is not efficient in this case because the lost profit is 25 when $o_{1,1}$ and $o_{1,3}$ are removed. The main reason is that the aggregate profit of removing multiple versions of the same multimedia object at the same time is not the simple summation of that of removing each version separately as explained later. □

In addition, the authors in [8] have not considered the influence of cache consistency on cache replacement. For example, when the content of the server is updated, all the cached versions should be updated as well so that the client could access the same content as it accesses from the server. This must lead to some additional cost for transferring the updated content to the cache. In real application, some multimedia data, such as images showing graphs about recent stock market, will be updated frequently. Therefore, cache consistency should be considered in a cache replacement algorithm. Otherwise, an algorithm may make inappropriate decisions as shown in the following example:

Example Assume that a multimedia object o_1 has three versions $o_{1,1}$, $o_{1,2}$, and $o_{1,3}$. The relationship among different versions of an object can be expressed by a weighted transcoding graph [8]. Figure 1 shows the weighted transcoding graph for o_1, where $o_{1,3}$ can be transcoded from $o_{1,1}$ and $o_{1,2}$, $o_{1,2}$ can be transcoded from $o_{1,1}$ only, and $o_{1,1}$ is the original version. It should be noted that not every o_{1,j_1} can be transcoded to o_{1,j_2} since o_{1,j_1} may not contain enough content information for the transcoding to o_{1,j_2}. The transcoding delay from one version to another is the number near the edge in Figure 1. For example, the transcoding delay from $o_{1,1}$ to $o_{1,2}$ is 6 ms. The transmission delays of fetching $o_{1,1}$, $o_{1,2}$, and $o_{1,3}$ from the server are assumed to be 10 ms, 8 ms, and 5 ms, respectively. The read rates for each version are assumed to be 3 and the update rates for each version are 6.

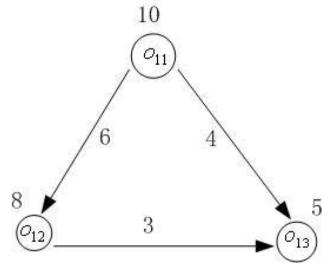

Figure 1. A Weighted Transcoding Graph

We analyze the following two cases. First, we consider the cases where we do not have updates. Suppose we do not have $o_{1,1}$ in the cache. Then, if $o_{1,1}$ is accessed, we transmit $o_{1,1}$ from the server. On the other hand, if we have $o_{1,1}$ in the cache, and $o_{1,1}$ is accessed, we need no cost. Therefore, by caching $o_{1,1}$, we can save the transmission cost of $o_{1,1}$, i.e., 10. Next, suppose $o_{1,1}$ is not cached and $o_{1,2}$ is accessed. Then, the server transcode $o_{1,1}$ to $o_{1,2}$, and transmit $o_{1,2}$ to the client, which causes the delay of $6 + 8$. On the other hand, if $o_{1,1}$ is cached, we can transcode $o_{1,1}$ to $o_{1,2}$ on the client, which only causes the delay of 6. Therefore, when $o_{1,2}$ is accessed, the cost we can save by caching $o_{1,1}$ is $6 + 8 - 6$. Similarly, when $o_{1,3}$ is accessed, the cost we can save by caching $o_{1,1}$ is $4 + 5 - 4$. Because the read rate for each version is 3, in total, the profit of caching $o_{1,1}$ is calculated by $3*10+3*(6+8-6)+3*(4+5-4) = 69$. Just similarly, the profit of caching $o_{1,2}$ is calculated by $3*(6+8)+3*(4+5-3) = 60$. Here, 3 in $4+5-3$ is the delay for transcoding the cached $o_{1,2}$ to $o_{1,3}$. Thus, we should cache $o_{1,1}$ to achieve more profit in this case.

Second, we consider the cases with data updates. When an object is updated at the server, and some versions of that object have been cached by some proxies, the updated object is transcoded into the corresponding versions at the server, and those versions are sent to the proxies. Therefore, caching an object causes the increase of the cost for that transcoding and that transmission. However, transcoding need to be done only once even when many proxies are caching a version of the object. Therefore, in the computation of the profit of each cached object, we ignore the cost for the transcoding, and we include only the cost for the transmission. Following this discussion, when we have updates, the profit of caching $o_{1,1}$ is calculated by $3 * 10 + 3 * (8 + 6 - 6) + 3 * (5 + 4 - 4) - 6 * 10 = 9$ and the profit of caching $o_{1,2}$ is calculated by $3 * (8 + 6) +$

$3 * (5 + 4 - 3) - 6 * 8 = 12$. Compared with the previous calculation, the terms $6 * 10$ and $6 * 8$, corresponding to the costs for the cache maintenance, are added. In this case, $o_{1,2}$ should be cached to achieve more profit, while it was $o_{i,1}$ when we do not consider updates. □

In this way, the existence and the ratio of updates affect the decision on what to cache; therefore, it is of particularly theoretical and practical necessity to address the problem of cache replacement for transcoding proxy caching by including not only the new emerging factors in the environment of transcoding proxies but also cache consistency. In this paper, we propose an efficient cache replacement algorithm for transcoding proxy caching, which integrates both the new emerging factors and cache consistency. Specifically, we formulate a generalized aggregate cost saving function to evaluate the profit of caching multimedia objects. Our algorithm evicts the objects in the cache with less generalized aggregate cost saving to fetch a new object into the cache. We evaluate our algorithm on different performance metrics through extensive simulation experiments and compare our algorithm with other algorithms proposed in the literature. The algorithm proposed in this paper can be viewed as an extension of our previous one proposed in [14]. To make this paper complete, we repeat some description.

The remainder of this paper is structured as follows: Section 2 introduces object caching. We present a cache replacement algorithm for transcoding proxy caching and its analysis in Section 3. The simulation and performance evaluation are described in Section 4 and Section 5, respectively. Section 6 summarizes our work and concludes the paper.

2. Object Transcoding

Transcoding is used to transform a multimedia object from one form to another, frequently trading off object fidelity for size, i.e., the process of converting a media file or object from one format to another. Transcoding is often used to convert video formats (i.e., Beta to VHS, VHS to QuickTime, QuickTime to MPEG). But it is also used to fit HTML files and graphics files to the unique constraints of mobile devices and other Web-enabled products. These devices usually have smaller screen sizes, lower memory, and slower bandwidth rates. In this scenario, transcoding is performed by a transcoding proxy server or device, which receives the requested document or file and uses a specified annotation to adapt it to the client.

The relationship among different versions of a multimedia object can be expressed by a weighted transcoding graph [8]. An example of such a graph is shown in Figure 2, where the original version A_1 can be transcoded to each of the less detailed versions A_2, A_3, A_4, and A_5. It should be noted that not every A_i can be transcoded to A_j since it is possible that A_i does not contain enough content information for the transcoding from A_i to A_j. In our example, transcoding can not be executed between A_4 and A_5 due to insufficient content information. The transcoding cost of a multimedia object from A_i to A_j is denoted by $w_A(j_1, j_2)$. The number beside each edge in Figure 2 is the transcoding cost from one version to another. For example, $w_A(1, 2) = 6$, and $w_A(3, 4) = 4$. $\phi_A(i)$ is the set of all the versions that can be transcoded from A_i, including A_i. For example, $\phi_A(1) = \{1, 2, 3, 4, 5\}$, $\phi_A(2) = \{2, 4, 5\}$, and $\phi_A(4) = \{4\}$. In this paper, we use G_A to denote a weighted transcoding graph.

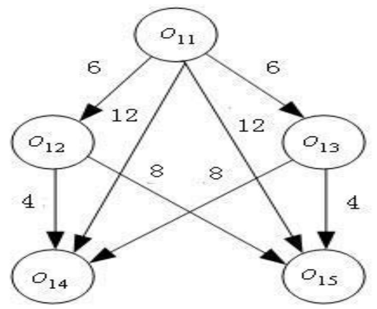

Figure 2. An Example of A Weighted Transcoding Graph

3. A Cache Replacement Algorithm

In this section we present an efficient cache replacement algorithm for transcoding proxy caching. In Section 3.1, a generalized aggregate cost saving function is defined to determine the rule for evicting the cached objects to make room for a new object if necessary. In Section 3.2, we present a cache replacement algorithm and its analysis. Finally, we introduce the method of estimating the parameter appearing in the algorithm.

3.1. Generalized Cost Saving Function

Let $o_{i,j}$ denote version j of object i and m_i denote the number of different versions of object i. $d_{i,j}$ is the cost of reading or writing $o_{i,j}$ from the server and $\omega_i(j_1, j_2)$ is the transcoding cost from version j_1 to version j_2 of object o_i. $\phi_i(j)$ is the set of all the versions of o_i that can be transcoded from $o_{i,j}$, including $o_{i,j}$ itself.

First we calculate the cost saving of caching only one version of an object (no other versions are cached). From the standpoint of clients, an optimal cache replacement algorithm should maximize the cost saving from caching multiple copies of objects by considering both the read cost and the update cost. Thus, the individual cost saving of caching only $o_{i,j}$ is defined as follows.

Definition 1 *$CS(o_{i,j})$ is a function for calculating the individual cost saving of caching $o_{i,j}$, while no other versions of object i are cached.*

$$CS(o_{i,j}) = \sum_{x \in \phi_i(j)} \lambda_{i,x}(\omega_i(1,x) + d_{i,x} - \omega_i(j,x)) - \mu_i d_{i,j} \quad (1)$$

In Equation (1), $\omega_i(1,x)$ and $d_{i,x}$ is the cost for transcoding the original version to version x and sending it to the client, which are saved by caching $o_{i,j}$, and $\omega_i(j,x)$ is the additional cost of transcoding version j to version x at the client, which is needed when caching $o_{i,j}$. $d_{i,j}$ is the additional cost of sending $o_{i,j}$ from the server to the cache upon updates of o_i so that the content of the cached version is consistent with that of the server. Now we give an example of the calculation of the individual cost saving.

As a matter of fact, there may be many versions of an object that can be cached at the same time if this is beneficial. In the following we discuss the aggregate cost saving of caching multiple versions of an object. We define the aggregate cost saving of caching multiple versions of an object at the same time as below.

Definition 2 *$CS(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k})$ is a function for calculating the aggregate cost saving of caching o_{i,j_1}, o_{i,j_2}, $\cdots$, o_{i,j_k}.*

$$CS(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) = \sum_{y \in \{j_1, j_2, \cdots, j_k\}} \left[\sum_{x \in \Phi_i(y,S)} \lambda_{i,x}(\omega(1,x) + d_{i,x} - \omega(y,x)) - \mu_i d_{i,y} \right] \quad (2)$$

where $\Phi_i(y,S)$ is the set of the versions that should be transcoded from $o_{i,y}$ when a set of versions S are cached.

If we use $s_{i,j}$ to denote the size of $o_{i,j}$, then we formulate the generalized aggregate cost saving function as follows:

$$CS^G(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) = CS(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) / \sum_{\alpha=1}^{k} s_{i,j_\alpha} \quad (3)$$

It is easy to see that the generalized aggregate cost saving function is further normalized by the total size of $o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}$ to reflect the object size factor. The rationale behind this normalization is to order the objects by the ratio of aggregate cost saving to their total object size. The generalized aggregate cost saving function defined in Equation (3) explicitly takes into consideration the new emerging factors in the environment of transcoding proxies. Importantly, it also takes cache consistency into account.

3.2. A Cache Replacement Algorithm

In this section we propose an efficient cache replacement algorithm for transcoding proxy caching based on the generalized aggregate cost saving function defined in Section 3.1. Suppose that there are l different multimedia objects cached and the size of a new object to be cached is s, then we should find a subset of objects $O^* \subseteq O$ that satisfies the following conditions.

(1) $\displaystyle\sum_{o_{i,j} \in O^*} s_{i,j} \geq s.$

(2) $(\forall O' \subseteq O$ that satisfies $(1))$ $CS^G(O^*) \leq CS^G(O')$.

where $O^* = \{o_{1,\alpha_1^1}, \cdots, o_{1,\alpha_1^{r_1}}, \cdots, o_{l,\alpha_l^1}, \cdots, o_{l,\alpha_l^{r_l}}\}$ is the set of objects to be removed, $O = \{o_{1,\beta_1^1}, \cdots, o_{1,\beta_1^{c_1}}, \cdots, o_{l,\beta_l^1}, \cdots, o_{l,\beta_l^{c_l}}\}$ is the set of objects cached, and $CS^G(O^*) = \sum_{i=1}^{l} CS^G(o_{i,\alpha_i^1}, \cdots, o_{i,\alpha_i^{r_i}})$. $CS^G(O')$ can be similarly defined. Obviously, (1) is to make enough room for the new object, and (2) is to evict those objects whose generalized aggregate cost saving is minimal.

The problem of finding O_* is NP hard, same as the packing problem. In the following, we present an algorithm that computes an approximate answer of the problem efficiently by decomposing the set of the candidate objects to be removed into smaller sets and each such can be decided in polynomial time.

Before we present the algorithm, we introduce some notations. In the following, let $R^*(i,k)$ denote the minimal generalized aggregate cost saving of caching k versions of object i and $R^*(k)$ the minimal generalized aggregate cost saving of the k objects to be removed. We can see that the k objects to be removed can be k versions of a multimedia object or different versions of different multimedia objects. Thus, k can be decomposed as $k = k_1 + k_2 + \cdots + k_a$, where a is the number of different objects to be removed and $0 \leq k_i \leq k$ is the number of versions of an object that are in the set of the k objects to be removed. For example, $1 \rightarrow \{1{+}0\}$, $2 \rightarrow \{2{+}0, 1{+}1\}$, $3 \rightarrow \{3{+}0, 2{+}1, 1{+}1{+}1\}$, $4 \rightarrow \{4+0, 3+1, 2+2, 2+1+1, 1+1+1+1\}$, $5 \rightarrow \{4{+}1, 3{+}1{+}1, 3{+}2, 2{+}1{+}1{+}1, 2{+}2{+}1, 1{+}1{+}1{+}1{+}1\}$, $\cdots$. For the instance of $k = 4$, k can be the combination of $1+1+1+1$, $1+1+2$, $2+2$, $1+3$, and $0+4$, where $1+1+1+1$ means that the objects to be removed should be the first four objects with minimal generalized aggregate cost savings of caching one version, $1{+}1{+}2$ means that the objects to be removed should be the three objects, i.e., the first two objects with minimal generalized aggregate cost savings of caching one version and the last object with minimal generalized aggregate cost saving of caching two versions, etc. It can be easily proved that there are at most k^2

different such combinations in all. Therefore, we have

$$R^*(k) = \min\{R^*(1,k), R^*(2,k), \cdots, R^*(l,k), \min_{k=k_1+k_2+\cdots+k_a}\{R^*(k_1) + R^*(k_2) + \cdots + R^*(k_a)\}\}$$

We denote the set of all the objects that achieves $R^*(k)$ by $O^*(k)$ and their total size is $S^*(k)$. Now we give an example to show how to calculate $R^*(k)$. For the case of $k = 3$, we have the combination of $3+0$, $1+2$, and $1+1+1$, each of which can be computed using the previous calculation results. For $3+0$, we just choose three versions from one object with minimal generalized aggregate cost savings of caching three versions. For $1+2$, we choose the version from an object with minimal generalized aggregate cost savings of caching one version and two versions from another object with minimal generalized aggregate cost savings of caching two versions. When we calculated $R^*(1)$ and $R^*(2)$, they may be using a same version. In this case, we select another version with minimal generalized cost saving that is not included. We denote the set of versions calculated by $R^*(1)$ and $R^*(2)$ as $O^*(1)$ and $O^*(2)$, respectively. In this case we will recalculate the set of versions with minimal number of elements by another set of versions of the same object with the same number of elements with more generalized aggregate cost saving. For example, if $o_{1,1} \in O^*(1)$ and $o_{1,1} \in O^*(2)$, then we will recalculate $O^*(1)$, i.e., finding $o_{1,j}$ with the minimal generalized aggregate cost saving except $o_{1,1}$ to represent $o_{1,1}$. Although this will be very costly in theory, the fact that the number of objects we hope to remove in practice is very small makes it feasible. We shall further study this issue in our future work. Based on the above calculation, we finally find how the k objects should be selected such that the generalized aggregate cost saving is minimized. In fact, there may exist a replacement decision by removing more than k objects and the generalized aggregate cost saving is less. Thus, the minimization here is conditional, i.e., under the condition that the minimal number of different objects is to be removed.

With the above analysis, we can devise the pseudocode of our algorithm as follows. In the algorithm, C is used to hold the cached objects, S_c is the cache capacity, S_u is the cache capacity used, o is the object to be cached, and its size is s.

```
Algorithm $MOR$ $(C, S_c, S_u, o)$
Input: $C, S_c, S_u, o$
Output: $O^*(n)$
1. INSERT $o$ INTO $C$
2. $n = 0$
3. $S^*(n) = 0$
4. WHILE $S_c - S_u - S^*(n) < s$ DO
5.     $n = n + 1$
6.     FOR $i = 1$ TO $l$ DO
7.         CALCULATE $R^*(i, n)$
8.     CALCULATE $R^*(n)$
9.         CHECK $O^*(n)$ (make the $n$ objects different)
```

Regarding to the time complexity of this algorithm, we have the following theorem.

Theorem 1 *The time complexity of Algorithm MOR is $O(K(l+K)\log(l+K))$, where l is the total number of different objects cached, $K = k^2$ and k is the number of objects to be removed.*

Proof Suppose k objects are removed to make room for the new object. The running time of Algorithm MOR mainly depends on Steps 4, 6, 8, and 9. The running time of Step 6 is determined by computing $R^*(i,n)$ for $1 \le i \le l$. For object i, calculating $R^*(i,n)$ is to find the minimal generalized aggregate cost saving of caching n versions of object i. Note that we should compute the aggregate profit of caching n versions of object i, and then order them according to the calculated profit. Thus, the running time for calculating $R^*(i,n)$ is $O(C(m_i;n)\log C(m_i;n))$. Therefore, The running time of Step 6 is $O(\sum_{i=1}^{l} C(m_i;n)\log C(m_i;n))$ since there are l objects cached and $C(m_i;n) = m_i!/(n!(m_i - n)!)$. The running time for Step 8 is $O((l+n)\log(l+n))$ because we should order all $l+n$ items to find the minimal one among them. Thus, the total running time for Algorithm MOR (Step 4) is $O(\sum_{n=1}^{K}[(l+n^2)\log(l+n^2) + \sum_{i=1}^{l} C(m_i;n)\log C(m_i;n)]) = O(K(l+K)\log(l+K))$ since in general $m_i \approx 10$ and l is very very large, where $K = k^2$, k is the number of objects to be removed, l is the number of different objects, and m_i is the number of versions of object i. Since the running time for Step 9 is $O(\log l)$, the total running time for Algorithm MOR is $O(K(l+K)\log(l+K))$. Hence, the theorem is proven. $\square$

From Theorem 1, we know that the time complexity of Algorithm MOR depends on k, i.e., the number of objects to be removed. In practical execution, we always stop the execution of searching the objects to be removed to make room for the new object when k reaches a certain number. This is based on the fact that it is not beneficial to remove many objects to accommodate only one object. So the practical time complexity of Algorithm MOR is $O(l\log l)$, which is the same as that of the algorithm proposed in [8]. However, from the algorithm we know that we have to search the entire cache for the other versions of the object and then recalculate the generalized aggregate cost savings for them whenever we insert or evict an object into or from the cache. Such operations are, in general, very costly. Here, we save calculated results for later computation, which will save a lot of computations. For example, after we finish computing $R^*(n)$, we save it using an array. When we hope to compute $R^*(n+1)$, we do not need to recalculate $R^*(k)$ for $1 \leq k \leq n$ again by reading it from the array directly.

4. Simulation Model

We have performed extensive simulation experiments to compare the relative performance of our algorithm with existing cache replacement algorithms. The system configuration is outlined in section 4.1, and existing cache replacement algorithms used for the purpose of comparison are introduced in Section 4.2.

4.1. System Configuration

We use the same system configuration and parameter estimation methods as we used in [14] in our simulation. To generate the workload of clients' requests, we model a single server that maintains a collection of m multimedia objects. The multimedia objects are assumed to be videos. The object popularity followed a Zipf-like distribution [4]. Specifically, the popularity of the ith video was proportional to $1/i^{\alpha}$. The default values of m and α were set to be 1000 and 0.75 respectively. The sizes of the videos followed a heavy tailed distribution with the mean value of $12K$ Bytes [16]. Although the average video size is assumed to be $12K$ Bytes, we think that the size of each data will not affect the relative performance of our algorithm as long as the distribution of the ratio of the sizes to each other is the same. The clients are divided into five classes and we assume that the sizes of the five versions of each video are 100 percent, 80 percent, 60 percent, 40 percent, and 20 percent of the original video size. The access probabilities of the clients are described as a vector of $< 0.2, 0.15, 0.3, 0.2, 0.15 >$. The transcoding relationship of the five versions is shown in Figure 3.

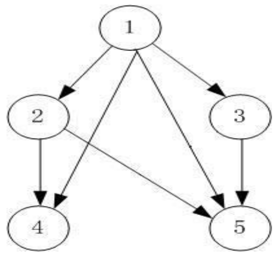

Figure 3. Transcoding Graph for Simulation

Regarding the transcoding rate, we set it, as in [6], to be $20K$ bytes per second. The delays of fetching the videos from the server are given by an exponential distribution. We assume that there is no correlation between the video size and the delay of fetching it from the server. This is justified by Shim et al. in [19].

The synthetic workloads are generated according to the results on the web workload characterization [9, 16], which are widely used for evaluating the performance of different caching systems [8, 20]. Table 2 lists the parameters and their values used in the simulation.

Parameter	Value
# of Objects	1000 objects
Delay of Fetching Objects	Exponential Distribution $p(x) = \theta^{-1}e^{-x/\theta}$ $(\theta = 0.45\ Sec)$
Web Object Size Distribution	Pareto Distribution $p(x) = \frac{ab^a}{a-1}$ $(a = 1.1, b = 8596)$
Web Object Access Frequency	Zipf-Like Distribution $\frac{1}{i^{\alpha}}$ $(i = 0.7)$
Transcoding Rate	$20KB/Sec$

Table 2. Parameters Used in Our Simulation

4.2. Evaluated Algorithms

We include the following algorithms for evaluating our proposed algorithm.

- LRU: Least Recently Used (LRU) evicts the web object which was requested the least recently. The cache purges one or more least recently requested objects to accommodate the new object if there is not enough room for it.
- $LNC-R$ *[17]*: Least Normalized Cost Replacement ($LNC-R$) is an algorithm that approximates the optimal cache replacement algorithm. It selects for replacement the least profitable objects. The profit function is defined as $profit(O_i) = (c_i \cdot f_i)/s_i$, where c_i is the average delay to fetch document O_i to the cache, f_i is the total number of references to O_i, and s_i is the size of document O_i.
- AE *[8]*: Aggregate Effect (AE) is an algorithm that formulates a generalized profit function to evaluate the aggregate profit from caching multiple versions of an object. The difference between AE and the solution proposed in this paper lies in that AE removes the objects from the cache one by one, and our solution removes the objects at the same time by considering the aggregate effect of caching multiple versions of the same object.

5. Performance Evaluation

In this section, we compare the performance of our algorithm with those algorithms introduced in Section 4.2 in terms of several performance metrics. The main performance metrics employed in the simulation include: delay-saving ratio (DSR), defined as the fraction of communication and server delays which is saved by fetching objects from the cache instead of the server; request response ratio (RRR), defined as the ratio of the access latency of the target object to its size; object hit ratio (OHR), defined as the ratio of the number of requests satisfied by the caches as a whole to the total number of requests; and staleness ratio (SR), defined as a fraction of cache hits which return stale objects. Here "stale" means that the time that an object was brought to the cache is less than the last-modified timestamp corresponding to the request. In the following figures, LRU, $LNC-R$, and AE denote the results for the three algorithms, and OA denotes the results for the algorithm proposed in Section 3.2. In the simulation, we only include the implementation without consideration of cache consistency.

5.1. Impact of Cache Size

In the first experiment set, we compare the performance of different algorithms across a wide range of cache sizes, from 0.04 percent to 15.0 percent of the total size of all the objects. The relative cache size of 15 percent is very large in the context of web caching due to the large network under consideration [20].

The first experiment investigates DSR as a function of the relative cache size and Figure 4 shows the simulation results. As presented in Figure 4, we can see that our algorithm outperforms the others. Specifically, the mean improvements of DSR over LRU, $LNC-R$, and AE are 9.5 percent, 21.6 percent, and 23.5 percent, respectively.

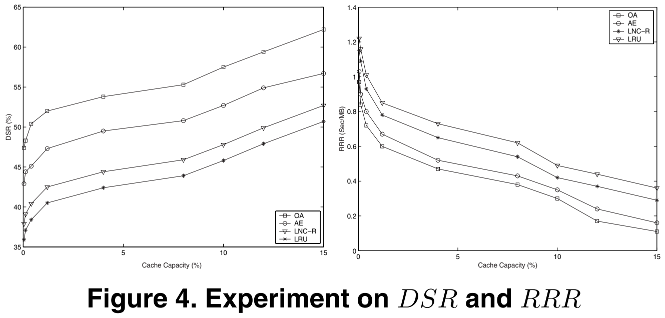

Figure 4. Experiment on DSR and RRR

We also describe the results of RRR as a function of the relative cache size in Figure 4. Clearly, the lower the RRR, the better the performance. As we can see, all algorithms provide steady performance improvement as the cache size increases. We can also see that OA constantly improves RRR compared to AE, others do not satisfy such criterion. For RRR to achieve the same performance as OA, the other algorithms need 1.4 to 5 times as much cache size.

Figure 5 also shows the results of OHR as a function of the relative cache size for different algorithms. By computing the objects with minimal generalized aggregate cost saving to be removed, we can see that the results for our algorithm constantly outperforms those of the others, especially for smaller cache sizes. We can also see that OHR steadily improves as the relative cache size increases, which conforms to the fact that more requests will be satisfied by the caches as the cache size becomes larger.

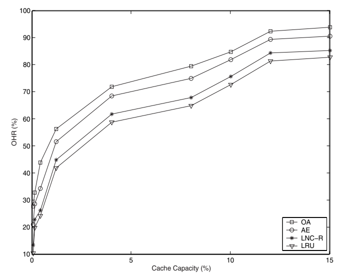

Figure 5. Experiment for OHR

5.2. Impact of Object Access Frequency

This experiment set examines the impact of object access frequency distribution on the performance of different algorithms. Figures 6, and 7 shows the performance results of DSR, and OHR for the values of Zipf parameter α from 0.2 to 1.0. We can see that OA consistently provides the best performance over a wide range of object access frequency distributions. Specially, OA reduces or improves DSR by 25 percent, 21 percent, and 11 percent compared to LRU, by 18 percent, 15 percent, and 7 percent compared to $LNC-R$, and by 15 percent, 10 percent, and 5 percent compared to AE for Zipf parameters of 0.2, 0.6, and 1.0, respectively; the default cache size used here (4 percent) is fairly large in the context of caching due to the large network under consideration (e.g. that of a regional ISP).

6. Conclusions

In this paper, we proposed an efficient cache replacement algorithm for transcoding proxy caching, which combined both the new emerging factors in transcoding proxies and cache consistency. The simulation results indicate that our algorithm constantly achieve good performance than the algorithms considered.

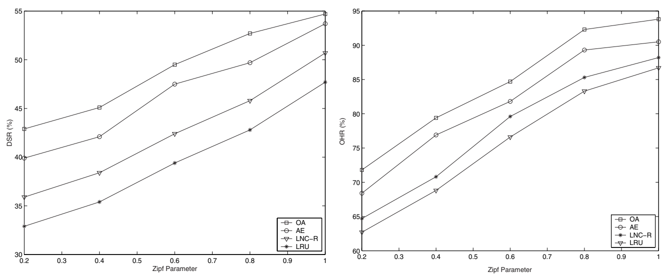

Figure 6. Experiment for DSR and OHR

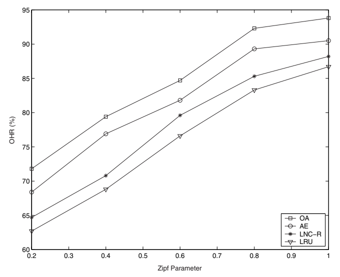

Figure 7. Experiment for OHR

References

[1] C. Aggarwal, J. L. Wolf, and P. S. Yu. *Caching on the World Wide Web*. IEEE Trans. on Knowledge and Data Engineering, Vol. 11, No. 1, pp. 94-107, 1999.

[2] A. Balamash and M. Krunz. *An Overview of Web Caching Replacement Algorithms*. IEEE Communications Surveys & Tutorials, Vol. 6, No. 2, pp. 44-56, 2004.

[3] M. Bhide, P. Deolasee, A. Katkar, and A. Panchbudhe. *Adaptive Push-Pull: Disseminating Dynamic Web Data*. IEEE Trans. on Computers, Vol. 51, No. 6, pp. 652-667, 2002.

[4] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. *Web Caching and Zip-like Distributions: Evidence and Implications*. Proc. of IEEE INFOCOM'99, pp. 126-134, 1999.

[5] P. Cao and C. Liu. *Maintaining String Cache Consistency in the World Wide Web*. IEEE Trans. on Computers, Vol. 47, No. 4, pp. 445-457, 1998.

[6] V. Cardellini, M. Colajanni, R. Lancellotti, and P. S. Yu. *A Distributed Architecture of Edge Proxy Servers for Cooperative Transcoding*. Proc. of the Third IEEE Workshop on Internet Applications (WIAPP'03), pp. 66-70, 2003.

[7] C. Chandra and C. S. Ellis. *JPEG Compression Metric as A Quality-Aware Image Transcoding*. Proc. of USENIX Second Symposium Internet Technology and Systems, pp. 81-92, 1999.

[8] C. Chang and M. Chen. *On Exploring Aggregate Effect for Efficient Cache Replacement in Trancoding Proxies*. IEEE Trans. on Parallel and Distributed Systems, Vol. 14, No. 6, pp. 611-624, 2003.

[9] C. Cunha, A. Bestavros, and M. Crovella. *Characteristics of WWWW Client-Based Traces*. Technical Report TR-95-010, Boston University, 1995.

[10] R. Floyd and B. Housel. *Mobile Web Access Using Network Web Express*. IEEE Personal Comm., Vol. 5, No. 5, pp. 47-52, 1998.

[11] S. Glassman. *A Caching Relay for the World Wide Web*. Computer Networks and ISDN Systems, Vol. 27, No. 2, pp, 165-173, 1994.

[12] R. Han, P. Bhagwat, R. LaMaire, T. Mummert, V. Perret, and J. Rubas. *Dynamic Adaption in an Image Transcoding Proxy for Mobile Web Browsing*. IEEE Personal Comm., Vol. 5, No. 6, pp. 8-17, 1998.

[13] R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. *Providing Availability Using Lazy Replication*. ACM Trans. on Computer Systems, Vol. 10, No. 4, pp. 360-391, 1992.

[14] K. Li, H. Shen, and K. Tajima. *Cache Design for Transcoding Proxy Caching*. Proc. of IFIP International Conference on Network and Parallel Computing 2004 (NPC04), pp. 187-194, 2004.

[15] R. Mohan, J. R. Smith, and C. Li. *Adapting Multimedia Internet Content for Univeral Access*. IEEE Trans. on Multimedia, Vol. 1, No. 1, pp. 104-114, 1999.

[16] J. Pitkow. *Summary of WWW Characterizations*. World Wide Web, Vol. 2, No. 1-2, pp. 3-13, 1999.

[17] P. Scheuermann, J. Shim, and R. Vingralek. *A Case for Delay-Conscious Caching of Web Documents*. Computer Networks and ISDN Systems, Vol. 29, No. 8-13, pp. 997-1005, 1997.

[18] B. Shen, S. J. Lee, and S. Basu. *Caching Strategies in Transcoding-Enabled Proxy Systems for Streaming Media Distribution Networks*. IEEE Trans. on Multimedia, Vol. 6, No. 2, pp. 375-386, 2004.

[19] J. Shim, P. Scheuermann, and R. Vingralek. *Proxy Cache Algorithms: Design, Implementation, and Performance*. IEEE Trans. on Knowledge and Data Engineering, Vol. 11, No. 4, pp. 549-562, 1999.

[20] X. Tang and S. T. Chanson. *Coordinated En-Route Web Caching*. IEEE Trans. on Computers, Vol. 51, No. 6, pp. 595-607, 2002.

[21] J. Wang. *A Survey of Web Caching Schemes for the Internet*. ACM Computer Communication Review, Vol. 29, No. 5, pp. 36-46, 1999.

[22] L. Yin, G. Cao, and Y. Cai. *A Generalized Cache Replacement Policy for Mobile Environments*. Proc. of the 2003 Symposium on Applications and the Internet (SAINT'03), pp. 14-21, 2003.