Cache Design for Transcoding Proxy Caching

Keqiu Li, Hong Shen, and Keishi Tajima

Graduate School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Tatsunokuchi, Ishikawa, 923-1292, Japan

Abstract. As audio and video applications have proliferated on the Internet, transcoding proxy caching is attracting an increasing amount of attention, especially in the environment of mobile appliances. Since cache replacement and consistency algorithms are two factors that play a central role in the functionality of transcoding proxy caching, it is of particular practical necessity to involve them into transcoding cache design. In this paper, we propose an original cache maintenance algorithm, which integrates both cache replacement and consistence algorithms. Our algorithm also explicitly reveals the new emerging factors in the transcoding proxy. Specifically, we formulate a generalized cost saving function to evaluate the profit of caching a multimedia object. Our algorithm evicts the objects based on the generalized cost saving to fetch each object into the cache. Consequently, the objects with less generalized cost saving are to be removed from the cache. On the other hand, our algorithm also considers the validation and write rates of the objects, which is of considerable importance for a cache maintenance algorithm. Finally, we evaluate our algorithm on different performance metrics through extensive simulation experiments. The implementation results show that our algorithm outperforms comparison algorithms in terms of the performance metrics considered.

Key words: Transcoding proxy caching, cache maintenance, Cache replacement, cache consistency, World Wide Web.

1 Introduction

With the explosive growth of the World Wide Web, proxy caching has become an important technique to improve network performance [15, 16]. Due to the limited cache space, it is impossible to store all the web objects in the cache. As a result, cache replacement algorithms [10, 14, 16] are used to determine a suitable subset of web objects to be removed from the cache to make room for a new web object. However, the improvement of network performance, such as access latency reduction achieved by caching web objects, does not come completely for free. In particular, maintaining the content consistence with the primary servers generates extra requests. Many proxy cache implementations depend on a consistency algorithm to ensure a suitable form of consistency for the cached

documents. Cache consistency algorithms [1, 3, 11] are used to guarantee the consistency of the cache web objects.

Transcoding is used to transform a multimedia object from one form to another, frequently trading off object fidelity for size for prevailing the operating environment. Since the transcoding proxy plays an important role in the functionality of caching, transcoding proxy caching is attracting more and more attention [4, 7, 9, 12]. However, due to the new emerging factors in the environment of transcoding proxies, existing cache replacement and consistency algorithms cannot be simply applied to solve the same problems for transcoding proxy caching. In [5], the authors presented several examples to explain the influence of these factors and explored the aggregate effect for efficient cache replacement in transcoding proxies. However, they considered only the problem of cache replacement and have not involved any issues on cache consistence. We argue that cache consistence has great influence on cache design. Consequently, it is of particular practical necessity to address the problem of cache maintenance by including both the cache replacement and consistence algorithms and the new emerging factors in the transcoding proxy. In this paper, we propose an original cache maintenance algorithm for transcoding proxy caching, which integrates both the cache replacement and consistence algorithms. Specifically, we formulate a generalized cost saving function to evaluate the profit of caching a multimedia object. Our algorithm evicts the objects based on the generalized cost saving to fetch each object into the cache. Consequently, the objects with less generalized cost saving are to be removed from the cache. On the other hand, our algorithm also considers the validation and write rates of the objects, which is of considerable importance for a cache maintenance algorithm. We evaluate our algorithm on different performance metrics through extensive simulation experiments and compare our algorithm with other algorithms proposed in the literature.

The remainder of this paper is structured as follow: We present a cache maintenance algorithm in transcoding proxies in Section 2. The simulation and performance evaluation are described in Section 3 and Section 4, respectively. Section 5 summarizes our work and concludes the paper.

2 A Cache Maintenance Algorithm in Transcoding Caching

The relationship among different versions of a multimedia objects can be expressed by a weighted transcoding graph [5]. Let $o_{i,j}$ denote version j of object i. $\omega(i, j)$ is the transcoding cost from version i to version j. The reference rates to different versions of objects, denoted by $f_{i,j}$, are assumed to be statistically independent, where $f_{i,j}$ is the mean reference rate to version j of object i. $\lambda_{i,j}$ is the read cost of version j of object i from the server, $\mu_{i,j}$ is the write cost of version j of object i, $\eta_{i,j}$ is the cost of validating the consistency of version j of object i, and $p_{i,j}$ is the probability of invalidating version j of object i cached.

First we calculate the cost saving from caching only one version of an object in the transcoding cache (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 write 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.*

$$CS(o_{i,j}) = \sum_{x \in D(j)} \lambda_{i,x}(d_{i,x} + \omega(1,x) - \omega(j,x) - \eta_{i,j} - p_{i,j}d_{i,x}) - \mu_{i,j}d_{i,j} \quad (1)$$

where $D(j)$ is the set of versions that can be transcoded from version j.

In Equation (1), $d_{i,x}$ is the cost of reading or writing $o_{i,x}$ from the server and $\omega(1,x)$ is the cost of transcoding from the original version to version x if $o_{i,j}$ is not cached. On the other hand, $\omega_{j,x}$ is the cost of transcoding from version j to version x, $\lambda_{i,x}$ is the read rate of $o_{i,x}$ from the client, and $\mu_{i,j}$ is the write rate of $o_{i,j}$ from the server if $o_{i,j}$ is cached.

As a matter of fact, there may be many versions of an object that can be cached at the same time if this is valuable. In the following we discuss the aggregate cost saving of caching multiple versions of an object. The aggregate cost saving of caching multiple versions of an object at the same time can be defined 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}.*

$$\begin{aligned} CS(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) = & \sum_{y \in \{j_1, j_2, \cdots, j_k\}} (\sum_{x \in D(y)} \lambda_{i,x}(d_{i,x} \\ & + \omega(1,x) - \omega(y,x) - \eta_{i,y} - p_{i,y}d_{i,x}) - \mu_{i,y}d_{i,y}) \end{aligned} \quad (2)$$

Now we define the marginal cost saving of caching a version of object i if there is at least one version cached.

Definition 3. *$CS(o_{i.j}|o_{i,j_1,o_{i,j_2},\cdots,o_{i,j_k}})$ is a function for calculating the marginal cost saving of caching $o_{i,j}$, given that o_{i,j_1}, o_{i,j_2}, $\cdots$, o_{i,j_k} are already cached, where $j \neq j_1, j_2, \cdots, j_k$.*

$$\begin{aligned} & CS(o_{i,j}|o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) = \\ & CS(o_{i,j}, o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) - CS(o_{i,j_1}, o_{i,j_2}, \cdots, o_{i,j_k}) \end{aligned} \quad (3)$$

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

$$CS^G(o_{i,j}) = \begin{cases} \frac{CS(o_{i,j})}{s_{i,j}} & \text{if no other versions are cached} \\ \frac{CS(o_{i,j}|o_{i,j_1},o_{i,j_2},\cdots,o_{i,j_k})}{s_{i,j}} & \text{if } o_{i,j_1},\ o_{i,j_2},\ \cdots,\ o_{i,j_k} \text{ are cached} \end{cases} \quad (4)$$

It is easy to see that the generalized cost saving function is further normalized by the size of $o_{i,j}$ to reflect the object size factor. The rationale behind this normalization is to order the objects by the ratio of cost saving to object size. The generalized cost saving function defined in Equation (4) explicitly takes into consideration the new emerging factors in transcoding caching and the aggregate effect of cache multiple versions of an object. Importantly, it takes into account not only the read cost but also the write cost.

Based on this function, we propose our cache replacement scheme as follows. Suppose the size of a new object to be cached is s, then we should find a subset of objects $O^* = \{o_{f_1,g_1}, o_{f_2,g_2}, \cdots, o_{f_l,g_l}\} \subseteq O$ that satisfies the following conditions. Here $O = \{o_{1,1}, o_{1,2}, \cdots, o_{1,l_1}, o_{2,1}, o_{2,2}, \cdots, o_{2,l_2}, \cdots, o_{m,1}, o_{m,2}, \cdots, o_{m,l_m}\}$ is the set of objects cached.

(1) $$\sum_{o_{f,g} \in O^*} o_{f,g} \leq s$$

(2) $$\sum_{o_{f,g} \in O^*} CS^G(o_{f,g}) \leq \sum_{o_{f,g} \in O'} CS^G(o_{f,g}), \forall\ O' \subseteq O \text{ that satisfies (1)}$$

Obviously, (1) is to make enough room for the new object, and (2) is to evict those objects whose total cost saving is minimized.

With the two conditions above, we can devise the pseudocode of our scheme as follows.

Algorithm GCS $(C, S_c, S_u, o_{i,j})$
1 add $o_{i,j}$ into C
2 recalulate the generalized cost saving of each version of object i
3 BuildHeap(C)
4 while $S_u - S_c < s$ do
5 Remove the first object from C
6 $S_u = S_u - s_{f,g}$
7 recalulate the generalized cost saving of each version of object i
8 BuildHeap(C)

In Algorithm GCS, C is used to hold the cached objects, S_c is the cache capacity, S_u is the cache capacity used, and $o_{i,j}$ is the object to be cached. For this algorithm, we can see that the most important thing is to find the objects with minimal cost saving.

It can be shown that the time complexity of Algorithm GCS is $O(S^2 log(S))$, where S is the number of different objects cached. 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 cost saving for them whenever we insert or evict an object into or from the cache. Such operations are, in general, very costly. Here we apply the data structure proposed in [5] to facilitate such operations.

In the actual implementation, the parameters for computing the generalized cost saving are usually not constant. To realize our algorithm, these parameters may have to be relaxed. Here, we adopt a "sliding window" technique [15] which

has been widely applied. It combines both the history data and the current value to estimate the parameters. Specially, the parameters are estimated as follow.

$$\begin{aligned} d_{i,j} &= \alpha \cdot d_{i,j}^{new} + (1-\alpha) \cdot d_{i,j}^{old} \\ \lambda_{i,j} &= \frac{K_1}{t_{i,j} - t_{i,j}^{K_1}} \\ \mu_{i,j} &= \frac{K_2}{s_{i,j} - s_{i,j}^{K_2}} \end{aligned} .$$

where $d_{i,j}^{new}$ is the newly measured cost of reading or writing $o_{i,j}$ from the client or the server and $d_{i,j}^{old}$ is the measured cost of reading or writing $o_{i,j}$ from the client or the server last time; $t_{i,j}$ is the time when the new request to $o_{i,j}$ is received from the client and $t_{i,j}^{K_1}$ is the time when the last K request is received from the client; $s_{i,j}$ is the time when the new update to $o_{i,j}$ is sent from the server and $s_{i,j}^{K_2}$ is the time when the last K update is sent from the server. $\eta_{i,j}$ is considered as a constant since it just sends an invalidation message to the server for all the documents. We estimate $p_{i,j}$ by $\frac{\lambda_{i,j}}{\lambda_{i,j}+\mu_{i,j}}$.

3 Simulation Model

In the simulation, to generate the workload of clients' requests, we model a single server that maintains a collection of m multimedia objects[1]. The object popularity followed a Zipf-like distribution [2]. 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 [13]. The clients are divided five classes. Without loss of generality, we assume that the sizes of the five versions of each video to be 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 six versions is shown in Figure 1.

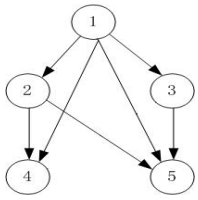

Fig. 1. Transcoding Graph for Simulation

Regarding the transcoding rate, we set it 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 [15].

[1] In the simulation, the multimedia objects are assumed to be videos.

The synthetic workloads are generated according to the recent results on the web workload characterization [6, 8, 13]. Table 1 lists the parameters and their values used in the simulation.

Table 1. Parameters Used in Our Simulation

Parameter	Value
Number of Nodes	10000
Delay of Fetching Objects	Exponential Distribution $p(x) = \theta^{-1}e^{-x/\theta}$ $(\theta = 0.45\ Sec)$
Number of Multimedia Objects	1000 objects
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 $\frac{1}{i^\alpha}$ $(i = 0.7)$
Average Request Rate Per Node	$U(1, 9)$ requests per second
Transcoding Rate	$20KB/Sec$

We compare our scheme with the following algorithms. (1) Least Recently Used (LRU) evicts the web object which was requested the least recently. The requested object is stored at each node through which the object passes. The cache purges one or more least recently requested objects to accommodate the new object if there is not enough room for it. (2) Least Normalized Cost Replacement ($LNC-R$) [14] is an algorithm that approximates the optimal cache replacement algorithm. (3) Aggregate Effect (AE) [5] is an algorithm that explores the aggregate effect of caching multiple versions of an object in the cache.

4 Performance Evaluation

The primary cache performance metric employed in the simulation is delay-saving ratio (DSR), which is defined as the fraction of communication and server delays which is saved by satisfying the references from the cache instead of the server. We also use average access latency (AST), object hit ratio (OHR) as secondary performance metrics. Here OHR is defined as the ratio of the number of requests satisfied by the caches as a whole to the total number of requests. We use staleness ratio (SR) as the primary consistency metric. The staleness ratio is 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

$CERWC$ shows the results for the model of coordinated en-route web caching in transcoding proxies, as proposed in Section 2.

In the experiments, we compare the performance of different models across a wide range of cache sizes, from 0.04 percent to 15.0 percent. The first experiment investigates DSR as a function of the relative cache size at each node and Figure 2(A) shows the simulation results. MA gives on average 13.3% improvement over LRU and and 5.3% over $LNC-R$. The maximal improvement over LRU and $LNC-R$ is 17.2% and 8.2% for cache size 0.5% and 2.0% respectively. On average, The DSR of MA is only 1.0% below that of AE. In the worst case, the DSR of MA is only 1.38% that of AE for cache size 10%.

Figure 2(B) shows the results of OHR as a function of the relative cache size for different models. Although MA is not designed to maximize the object hot ratio, it still provides an improvement over LRU and $LNC-R$. In particular, the average improvement is 29.6% over LRU and 22.5% over $LNC-R$. The object hit ratio provides even closer to the hit ratio of AE; on average, 1.1% and no more than 2.39% below the object hit ratio of AE.

In addition to improving performance of the cache, the MA algorithm also significantly improves irs consistence. On average, MA achieves a staleness ratio which is by factor of 3.2 better than that of AE, in the worst case it improves SR of AE by factor of 1.9 when the cache size is 0.5%. MA also improves SR over LRU and $LNC-R$. On average, MA achieves a staleness ratio which is 50.8% better than that of LRU and 50.1% better than that of $NC-R$. In the worst case, it improves SR of LRU by 10.2% when the cache size is 0.5% and improves SR of $LNC-R$ by 8% when the cache size is 2.0%. The staleness ratio comparison of the four algorithms can be found in Figure 2(C).

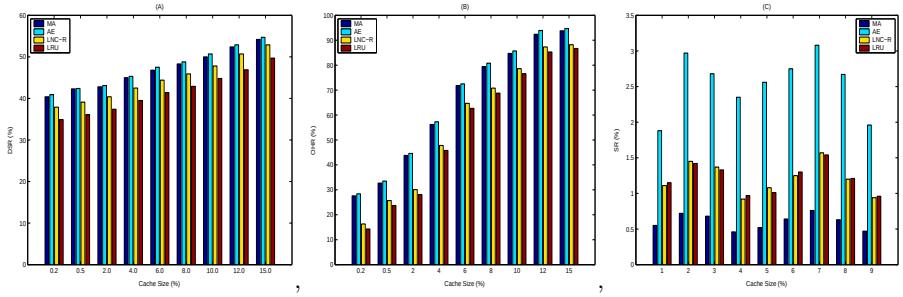

Fig. 2. Experiments for $DSR, OSR, and SR$

5 Conclusions

In this paper, we proposed a maintenance algorithm for transcoding proxy caching, which combined both cache replacement and cache consistence. The

simulation indicated that our algorithm could significantly improve the staleness ratio, while keeping the cache performance within acceptable loss. This greatly benefit the cache design for transcoding proxy caching.

References

1. M. Bhide, P. Deolasee, A. Katkar, and A. Panchbudhe. *Adaptive Push-Pull: Disseminating Dynamic Web data.* IEEE Transactions on Computers, Vol. 51, No. 6, pp. 652-667, June, 2002.
2. L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. *Web Caching and Zip-like Distributions: Evidence and Implications.* Proc. IEEE INFOCOM'99, pp. 126-134, 1999.
3. P. Cao and C. Liu. *Maintaining String Cache Consistency in the World Wide Web.* IEEE Transactions on Computers, Vol. 47, No. 4, pp. 445-457, April, 1998.
4. C. Chandra and C. S. Ellis. *JPEG Compression Metric as a Quality-Aware Image Transcoding.* Proc. USENIX Second Symposium Intenet Technology and Systems, pp. 81-92, 1999.
5. C. Chang and M. Chen. *On Exploring Aggregate Effect for Efficient Cache Replacement in Transcoding Proxies.* IEEE Transactions on Parallel and Distributed Systems, Vol. 14, No. 6, pp. 611-624, June, 2003.
6. C. Cunha, A. Bestavros, and M. Crovella. *Characteriatics of WWWW Client-Based Traces.* Technical Report TR-95-010, Boston University, April, 1995.
7. R. Floyd and B. Housel. *Mobile Web Access Using Network Web Express.* IEEE Personal Comm., Vol. 5, No. 5, pp. 47-52, Dec., 1998.
8. S. Glassman. *A Caching Relay for the World Wide Web.* Computer Network and ISDN Systems, Vol 27, No. 2, pp, 165-173, 1994.
9. 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, Dec., 1998.
10. S. Jin and A. Bestavros. *Greeddual* Web Caching Algorithm Exploiting the Two Sources of Temporal Locality in Web Request Streams.* Computer Comm., Vol. 4, No. 2, pp. 174-183, 2001.
11. R. Ladin, B. Liskov, L. Shrira, and S. Ghemawat. *Providing Availability Using Lazy Replication.* ACM Transactions on Conputer Systems, Vol. 10, No. 4, pp. 360-391, 1992.
12. R. Mohan, J. R. Smith and C. Li. *Adapting Multimedia Internet Content for Univeral Access.* IEEE Transactions on Multimedia, Vol. 1, No. 1, pp. 104-114, March 1999.
13. J. Pitkow. *Summary of WWW Characteristics.* World Wide Web, Vol. 2, No. 1-2, pp. 3-13, 1999.
14. P. Scheuermann, J. Shim, and R. Vingralek. *A Case for Delay-Conscious Caching of Web Documents.* Computer Network and ISDN Systems, Vol 29, No. 8-13, pp. 997-1005, 1997.
15. J. Shim, P. Scheuermann, and R. Vingralek. *Proxy Cache Algorithms: Design, Implementation, and Performance.* IEEE Transactions on Knowledge and Data Engineering, Vol 11, No. 4, pp. 549-562, 1999.
16. R. P. Wooster and M. Abrams. *Proxy Caching that Estimates Page Load Delays.* Computer Networks and ISDN Systems, Vol 29, Nos. 8-13, pp. 977-986, 1997.