On Exploring Aggregate Effect for Efficient Cache Replacement in Transcoding Proxies

Cheng-Yue Chang and Ming-Syan Chen, *Senior Member*, *IEEE*

Abstract—Recent technology advances in mobile networking have ushered in a new era of personal communication. Users can ubiquitously access the Internet via many emerging mobile appliances, such as portable notebooks, personal digital assistants (PDAs), and WAP-enabled cellular phones. While the transcoding proxy is attracting an increasing amount of attention in this environment, it is noted that new caching strategies are required for these transcoding proxies. We propose, in this paper, an efficient cache replacement algorithm for transcoding proxies. Specifically, we formulate a generalized profit function to evaluate the profit from caching each version of an object. This generalized profit function explicitly considers several new emerging factors in the transcoding proxy and the aggregate effect of caching multiple versions of the same object. It is noted that the aggregate effect is not simply the sum of the costs of caching individual versions of an object, but rather, depends on the transcoding relationship among these versions. The notion of a weighted transcoding graph is devised to evaluate the corresponding aggregate effect efficiently. Utilizing the generalized profit function and the weighted transcoding graph, we propose, in this paper, an innovative cache replacement algorithm for transcoding proxies. In addition, an effective data structure is designed to facilitate the management of the multiple versions of different objects cached in the transcoding proxy. Using an event-driven simulation, it is shown that the algorithm proposed consistently outperforms companion schemes in terms of the delay saving ratios and cache hit ratios.

Index Terms—Mobile computing systems, transcoding proxies, weighted transcoding graphs, cache repleacement.

1 Introduction

Recent technology advances in mobile communication have ushered in a new era of personal communication. Users can ubiquitously access the Internet via many emerging mobile appliances, such as portable notebooks, personal digital assistants (PDAs), and WAP-enabled cellular phones. As these devices are divergent in size, weight, input/output capabilities, network connectivity, and computing power, differentiated services should be tailored and delivered in a certain way to meet their diverse needs. In addition, users may have different content presentation preferences. Both lead to the demand of transcoding technologies to adapt the same Web object to various mobile appliances [5], [8], [10], [16], [21].

Transcoding is defined as a transformation that is used to convert a multimedia object from one form to another, frequently trading off object fidelity for size. It can be applied to media transformation within the same format of media type (e.g., from high-resolution JPEG to low-resolution JPEG format), between different formats of the same media type (e.g., from JPEG to GIF format), or between different media types (e.g., speech to text or video clip to image set). For the mobile appliances featured with lower-bandwidth network connectivity, transcoding can be used to reduce the Web object size by lowering the image resolution or downscaling the image size. For the mobile appliances which only accept a text, transcoding can be used to covert the image or speech into a text. As pointed out in [7], from the aspect of the place where transcoding is performed, the transcoding technologies can be classified into three categories, i.e., server-based, client-based, and proxy-based approaches. In the server-based approaches [17], Web objects are offline transcoded to multiple versions and stored in the server disks. The advantage of this approach is that no additional delay will be incurred by transcoding during the time between the client issues a request and the server responses to it. The drawback is, however, keeping several versions of the same object in the server may cost too much storage space. Further, this approach is not flexible in dealing with the future change of clients' needs. Conversely, in the client-based approaches, transcoding is left for mobile clients for considerations. The advantage of this approach is that it can preserve the original semantic of system architecture and transport protocols. However, transcoding at the client side is extremely costly due to the limited connection bandwidth and computing power of a mobile device. For these reasons, it will be better to transcode the Web objects at the intermediate proxies. Many studies have recently been conducted to explore the advantages of proxy-based approaches [8], [9], [12], [13], [15], where an intermediate proxy is able to on-the-fly transcode the requested object to a proper version according to the client's specification before it sends this object to the client. Such an intermediate proxy which possesses the transcoding capability is referred to as a transcoding proxy in this paper.

While the transcoding proxy is attracting more and more attention, it is noted that a transcoding proxy, just as a traditional Web proxy, plays an important role in the functionality of caching. To enable the cache replacement algorithms devised for traditional Web proxies [1], [2], [3],

- The authors are with the Department of Electrical Engineering, National Taiwan University, Taipei, Taiwan, Republic of China.
E-mail: {mschen, cychang}@cc.ee.ntu.edu.tw.

Manuscript received 30 Nov. 2001; revised 30 Dec. 2002; accepted 3 Jan. 2003.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number 115473.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

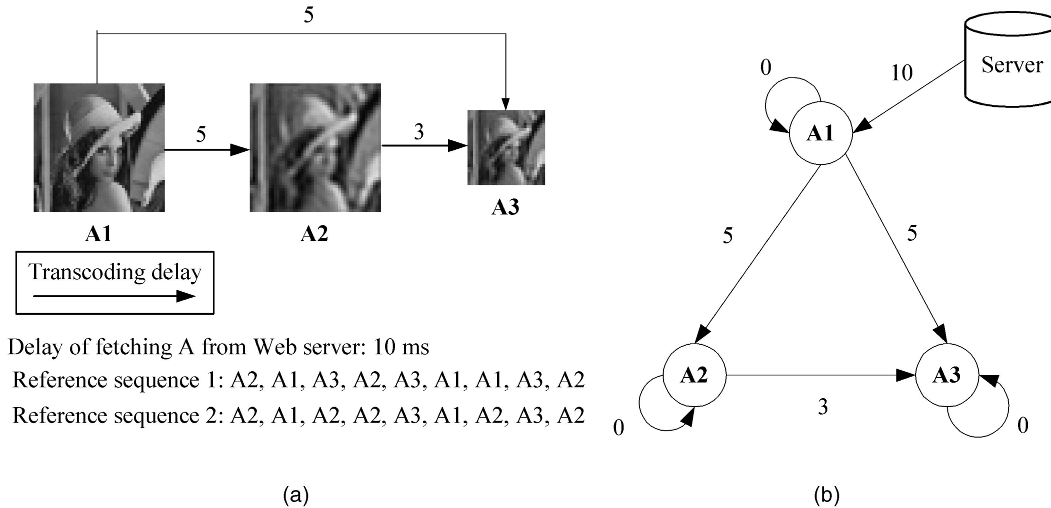
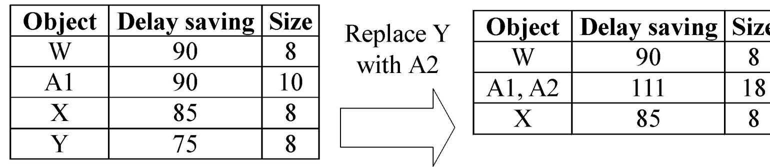


Fig. 1. Two contrary examples illustrating the effect of reference distribution.

[20], [25] to handle the situations in transcoding proxies, extensions to these traditional algorithms are needed. In [7], two extended strategies, which are called *coverage-based* and *demand-based* strategies, are proposed. Note that, when transcoding the requested object to the specified version before sending this object to the client, the transcoding proxy has the opportunity of caching either the original version, the transcoded version, or both versions of the object in the local memory. In view of this, the *coverage-based* strategy in [7] is designed to choose the original version of the object to cache, whereas the *demand-based* strategy is designed to choose the transcoded one to cache. In addition, most cache replacement algorithms employ an *eviction function* to determine the priorities of objects to be replaced. The LRU (Least Recently Used) algorithm, for example, evicts the page with the largest elapsed time since the last access of that page. The LFU (Least Frequently Used) algorithm, on the other hand, evicts the page with the smallest reference rate in the cache. Combining the *eviction function* of the traditional cache replacement algorithm with these two strategies, we can decide the priority of the cached object to be evicted and the version of the new coming object to be cached. For example, the *demand-based* extension to the LRU algorithm in [7] replaces the object of the largest elapsed time since its last access with the transcoded version of the new coming object. The *coverage-based* extension works similarly except the evicted object is replaced with the original version of the new coming object.

Note, however, that both the *coverage-based* and *demand-based* extensions in [7] are designed for efficient use, but not for optimal cache replacement decisions. In contrast, we shall design in this paper a cache replacement algorithm for transcoding proxies that maximizes the performance of Web object caching. The motivation for our study is mainly due to the new emerging factors in the environment of transcoding proxies. First, transcoding incurs additional delays, and this factor should be included in the *eviction functions* for consideration whenever the eviction priority is determined. For example, suppose there are two cached objects, say, A and B, having the same eviction priority according to some eviction function which does not consider the transcoding delay. Clearly, if the transcoding delay of A is longer than that of B, A should be given a higher eviction priority than B so as to shorten the response time. Without considering the transcoding delays, the traditional cache replacement algorithms could make a wrong replacement decision. Second, different versions of the same object are of different sizes. Specifically, the transcoded version is usually of a smaller size than the original one. Thus, the strategy which chooses the transcoded version to cache may admit more objects with the same cache capacity. An *eviction function* is thus required to take this object size factor into consideration. Third, the reference rates to different versions of an object should be considered separately because the distribution of them could affect the caching decision. This can be seen by two contrary examples illustrated in Fig. 1. Assume that the *Lena image* has three versions (i.e., $A1$, $A2$, and $A3$). $A1$ is the original version with full resolution and size. $A1$ can be transcoded to $A2$, which is of the same size, but a lower resolution. $A1$ can also be transcoded to $A3$, which is of a lower resolution and a smaller size. In addition, $A2$ can be transcoded to $A3$, whereas $A3$ is the least detailed version, which cannot be transcoded any more. The transcoding delay for $A1$ to $A2$, and that for $A1$ to $A3$ are both assumed to be 5 ms and the transcoding delay for $A2$ to $A3$ is 3 ms. The transmission delay of fetching object A from the Web server is assumed to be 10 ms. Consider the reference sequence 1 in Fig. 1, where the numbers of references to $A1$, $A2$, and $A3$ are all equal to 3. By caching $A1$, we can get the delay saving equal to 90 ms, which is obtained by $3 * 10 + 3 * (10 + 5 - 5) + 3 * (10 + 5 - 5)$. By caching $A2$, we can, however, get the delay saving only equal to 81 ms, which is obtained by $3 * (10 + 5) + 3 * (10 + 5 - 3)$. As such, caching a more detailed version can get more benefit in this case. However, in reference sequence 2 in Fig. 1, the numbers of references to $A1$, $A2$, and $A3$ are 2, 5, and 2, respectively. By caching $A1$, we can get the delay saving equal to 90, which is obtained by $2 * 10 + 5 * (10 + 5 - 5) + 2 * (10 + 5 - 5)$. In contrast, by caching $A2$, we can get the delay saving equal to 99, which is obtained by $5 * (10 + 5) + 2 * (10 + 5 - 3)$. In this case, caching a less detailed version will get more benefit than

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.



Object	Delay saving	Size
W	90	8
A1	90	10
X	85	8
Y	75	8

Object	Delay saving	Size
W	90	8
A1, A2	111	18
X	85	8

Fig. 2. An example of a wrong cache replacement decision.

caching a more detailed one. It is noted that, although the sum of the reference rates to $A1$, $A2$, and $A3$ in sequence 2 is the same as the sum in sequence 1, the distribution of them indeed affects the caching decision accordingly.

Furthermore, the situation of caching two or more versions of the same object makes the caching problem caused by transcoding proxies more complicated. Consider the example in reference sequence 1 in Fig. 1. The individual delay savings from caching $A1$ and $A2$ are 90 and 81 ms, respectively. However, the aggregate delay saving from caching $A1$ and $A2$ at the same time is not simply the sum of the individual delay savings (i.e., $90 + 80 = 171$ ms), but rather, depends on the transcoding relationship among them. Note that the transcoding relationship in Fig. 1a can be depicted by a directed graph with weights on edges, as shown in Fig. 1b. Such a directed graph is called a *weighted transcoding graph*, which will be formally defined in Section 2. In the example of Fig. 1, when caching $A1$ and $A2$ at the same time, the subsequent references to $A2$ and $A3$ are not necessary to be supported by the transcoding from $A1$ any more. Instead, they can be supported by the transcoding from $A2$ because there needs no transcoding between $A2$s and the transcoding cost from $A2$ to $A3$ is smaller than that from $A1$ to $A3$. Hence, the delay saving from caching $A1$ is overevaluated and should be revised as $30 = 3 * 10$. The aggregate delay saving from caching $A1$ and $A2$ together is thus 111 rather than 171. Such an overevaluation will probably result in a wrong cache replacement decision, as will be explained below.

In the case of Fig. 2, assume that the cache capacity is 34 Kbytes, and the cache is filled with W, $A1$, X, and Y. The computed delay saving and size of each object is listed in the left table of Fig. 2. Suppose here comes a new object $A2$. In the example of Fig. 1, we have calculated the delay saving from caching it is 122. Without considering the aggregate effect of caching $A1$ and $A2$ together, the cache replacement algorithm will choose to replace Y with $A2$ because the delay saving from caching Y is smaller than that from caching $A2$. The resulting cache will thus become the one listed in the right table of Fig. 2. Recall that the aggregate delay saving from caching both $A1$ and $A2$ is 111 rather than 171. Thus, the overall delay saving of the entire cache decreases from 340 to 286 as Y is replaced with $A2$. Obviously, replacing Y with $A2$ is a wrong cache replacement decision.

From the illustrative examples in Figs. 1 and 2, it is noted that the new emerging factors in the environment of transcoding proxies indeed affect the replacement decision and, in turn, the performance of caching algorithm. The aggregate effect even makes the caching problem caused by transcoding proxies more complicated and unable to be solved by simple extensions to traditional cache replacement algorithms. Consequently, we propose an efficient cache replacement algorithm for transcoding proxies in this paper. Specifically, we formulate a *generalized profit function* to evaluate the profit from caching a version of an object. This *generalized profit function* explicitly considers several new emerging factors in the transcoding proxy and the aggregate effect of caching multiple versions of the same object. As explained, the aggregate effect is not simply the sum of the delay savings from caching individual versions of an object, but rather, depends on the transcoding relationship among these versions. Thus, the notion of a *weighted transcoding graph* is defined to represent the transcoding relationship among the different versions of an object. In addition, to evaluate the aggregate effect properly, we devise *Procedure MATC (standing for Minimal Aggregate Transcoding Cost)* to find the subgraph of the weighted transcoding graph. This subgraph shows the transcoding relationship which minimizes the aggregate transcoding cost when caching a certain set of versions of the object. Utilizing the *generalized profit function* as the eviction function, we propose, in this paper, an innovative cache replacement algorithm for transcoding proxies. This algorithm is referred to as algorithm *AE (standing for Aggregate Effect)* for the reason that it explores the aggregate effect of caching multiple versions of an object in the transcoding proxy. In addition, an effective data structure is designed to facilitate the management of the large number of different versions of objects cached in the transcoding proxy. Using an event-driven simulation, we evaluate the performance of our algorithm under several circumstances. The experimental results show that algorithm *AE* consistently outperforms the companion schemes in terms of the primary performance metric, delay saving ratio, and also, the secondary performance metrics, hit ratios.

The contributions of this paper are twofold. We not only devise for transcoding proxies an innovative cache replacement algorithm by proposing the notion of a weighted transcoding graph to take the transcoding delays into consideration, but also, in light of the aggregate effect which is obtained by traversing the weighted transcoding graph, optimize the overall effect of cache replacement. To the best of our knowledge, none of the prior work has discovered the aggregate effect of caching multiple versions of the same object, let alone devising a specific cache replacement algorithm for transcoding proxies. These features distinguish this paper from others. With the rapid advances in mobile technologies, users can ubiquitously access the Internet via many emerging mobile appliances with various features. The increasing demand of transcoding proxies justifies the timeliness and importance of this study.

The rest of this paper is organized as follows: In Section 2, we introduce the system environment where the caching issues in the transcoding proxy are considered. We propose

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

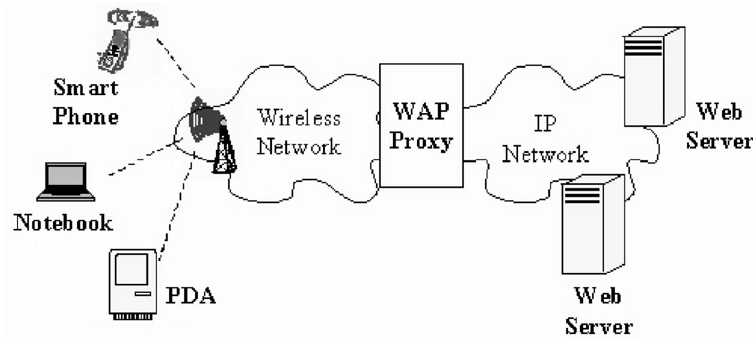


Fig. 3. The WAP architecture.

our cache replacement algorithm for transcoding proxies in Section 3. In Section 4, we empirically evaluate the performance of several algorithms. We conclude this paper with Section 5.

2 System Environment

To facilitate our later discussion in this paper, we describe the system environment in this section. The system architecture is presented in Section 2.1. The Web object transcoding is described in Section 2.2.

2.1 System Architecture

This paper studies cache replacement algorithms for the transcoding proxy, which typically interconnects two heterogeneous networks. A well-known example of such a transcoding proxy is the WAP proxy or a gateway, which interconnects the wireless network and the Internet.

Fig. 3 illustrates the architecture of WAP [22]. It consists of three important components: the mobile clients, the WAP proxy, and the Web servers. The mobile clients connect to the WAP proxy via the wireless network, whereas the WAP proxy connects to the Web servers via the IP network. The mobile clients (e.g., portable notebooks, personal digital assistants, and WAP-enabled cellular phones) are capable of requesting and rendering WAP content. They vary widely in their features such as the screen size and color, the processing power, storage, user interface, and the software. The clients can describe their capabilities via User Agent Profile (UAProf) [24] supported by WAP. UAProf is initially conveyed when a WSP session is established with the WAP proxy.

In addition to several other roles in WAP architecture, a WAP gateway acts as a transcoding proxy for diverse accesses. As assumed by the clients, the WAP proxy will keep the UAProfs for them when the clients initially establish WSP sessions to the WAP proxy. Keeping these UAProfs in the proxy, the WAP proxy knows the preference of each client and will on-the-fly transcode the requested object to a proper version according to the client's UAProf before delivering to it. Moreover, as pointed out in [23], to speed up wireless data access, the WAP proxy also acts as the role of an caching proxy. Note that the proxy in such a circumstance has the opportunity of caching either the original version, the transcoded version, or both versions of the object for future references. This emerging caching model justifies the problem we deal with in this paper. Note that, while being applicable to the WAP architecture, the cache replacement algorithm we shall devise is for general transcoding proxies and by no means restricted to the WAP applications.

2.2 Web Object Transcoding

Transcoding is defined as a transformation that is used to convert a multimedia object from one form to another, frequently trading off object fidelity for size. The original object contains the full information of the content and usually corresponds to the original version or the most detailed version of the object. The original version of the object can be transcoded to a less detailed one and such a transcoded object is called the transcoded version or the less detailed version of the object. Without loss of generality, we assume that each object can be represented in n versions. The original version of object is denoted as V_1 (i.e., version 1), whereas the least detailed version which cannot be transcoded any more is denoted as V_n (i.e., version n). The intermediate versions are, in turn, denoted as $V_2, \ldots, V_{n-1}$, in which V_i is a more detailed version than V_j for each i, j if $i < j$.

It is noted that not every V_i can be transcoded to V_j for $i < j$. It is possible that V_i does not contain enough content information for the transcoding to V_j. Consider the following case: Suppose V_i is a color bitmap cached in the transcoding proxy and a lower resolution color bitmap V_j has to be produced now. If V_j is not a trivial divisor of the number of pixels in V_i, the proxy may spend a lot of time and computing power interpolating V_i to produce a V_j without getting any good result. Thus, the transcoding proxy must a priori know the transcodable relationship of an object whenever it performs the transcoding. To this end, we devise the notion of *weighted transcoding graph* as defined in Definition 1.

Definition 1. *The weighted transcoding graph, G_i, is a directed graph with weight funtion w_i. G_i depicts the transcoding relationship among transcodable versions of object i. For each vertex $v \in V[G_i]$, v represents a transcodable version of object i. Version u of object i is transcodable to version v iff there exists a directed edge $(u, v) \in E[G_i]$. The transcoding cost from version u to v is given by $w_i(u, v)$, which is the weight of the edge from u to v.*

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

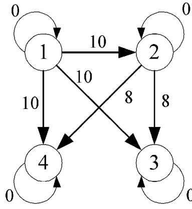


Fig. 4. An example of weighted transcoding graph.

The weighted transcoding graph is, in essence, an extension to the *transcoding relation graph* proposed in [7]. For each object, we maintain a corresponding weighted transcoding graph in the transcoding proxy. Note that, since each version is surely transcodable to itself with null transcoding, there is a directed edge pointed from each version to itself with the corresponding weight being 0. An example of *weighted transcoding graph* is given in Fig. 4, where the original version V_1 can be transcoded to each of the less detailed versions V_2, V_3, and V_4. However, V_3 cannot be transcoded to V_4 due to insufficient content information.

3 Cache Replacement Algorithm for Transcoding Proxies

With the background of the system environment described in Section 2, we present the proposed cache replacement algorithms for transcoding proxies in this section. In Section 3.1, we define the *generalized profit function*, which will be used to determine the eviction priorities of the cached objects in our algorithm. This *generalized profit function* explicitly considers the new emerging factors in the transcoding proxy and the aggregate effect explained in Section 1. To evaluate the aggregate effect properly, we devise *Procedure MATC (standing for Minimal Aggregate Transcoding Cost)* to find the subgraph of the weighted transcoding graph. This subgraph shows the transcoding relationship which minimizes the aggregate transcoding cost when caching a certain set of versions of the object. Utilizing this *generalized profit function* as the eviction function, we design an innovative cache replacement algorithm for transcoding proxies in Section 3.2. This algorithm is referred to as algorithm *AE* (standing for Aggregate Effect) for the reason that it explores the aggregate effect of caching multiple versions of an object in the transcoding proxy. In addition, an effective data structure is designed to facilitate the management of the large number of different versions of objects cached in the transcoding proxy. In Section 3.3, we use an example to illustrate how algorithm *AE* works and to demonstrate the usefulness of the data structure designed.

3.1 Generalized Profit Function

As mentioned in Section 1, most cache replacement algorithms employ an *eviction function* to determine the priorities of objects to be replaced. In our algorithm, we determine the eviction priorities according to the *generalized profit function*, which will be defined later. Initially, we will derive the individual profit function for the calculation of the profit from caching a single version of an object. Then, we derive the aggregate profit function for the calculation of the profit from caching multiple versions of an object at the same time. Based on the aggregate profit function, we formulate the marginal profit function for the calculation of the profit from caching a version of an object, given other versions of the object are already cached. Finally, we conclude these profit functions as the *generalized profit function*.

Let $o_{i,j}$ denote version j of object i. The reference rates to different versions of objects are assumed to be statistically independent and denoted by $r_{i,j}$, where $r_{i,j}$ is the mean reference rate to version j of object i. The mean delay to fetch object i from the original server to the transcoding proxy is denoted by d_i, where we define the delay to fetch an object as the time interval between sending the request and receiving the last byte of response. The size of version j of object i is denoted by $s_{i,j}$. The corresponding *weighted transcoding graph* of object i is denoted by G_i and the transcoding delay of object i from version x to version y is given by the weight on the edge (x, y) in $E[G_i]$, which is denoted by $w_i(x, y)$. A list of the symbols used is give in Table 1. With these given parameters, we can determine the profit from caching version j of object i.

First, we consider the profit from caching a single version of an object in the transcoding proxy (i.e., no other versions are cached). From the standpoint of client users, an optimal cache replacement algorithm is expected to minimize the response time perceived. In other words, an optimal cache replacement algorithm will be designed to maximize the delay saving by caching a certain set of Web objects. Thus, the individual profit from caching version j of object i is defined in terms of the delay saving obtained as below.

Definition 2. *$PF(o_{i,j})$ is a function for the individual profit from caching $o_{i,j}$, while no other version of object i is cached.*

$$PF(o_{i,j}) = \sum_{(j,x)\in E[G_i]} r_{i,x} * (d_i + w_i(1, x) - w_i(j, x)). \quad (1)$$

TABLE 1
A List of the Symbols Used

Symbol	*Description*
$o_{i,j}$	version j of object i
$r_{i,j}$	the mean reference rate to version j of object i
d_i	the delay of fetching object i from the original server to the proxy
$s_{i,j}$	the size of version j of object i
G_i	the corresponding transcoding relation graph of object i
$w_i(u, v)$	the weight on each edge (u, v) in G_i

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

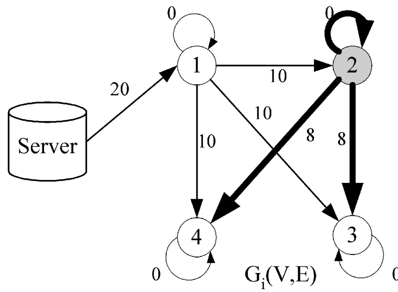


Fig. 5. An example illustrating Definition 1.

Note, in (1), each $(j,x) \in E[G_i]$ represents each transcodable version from version j to version x. Expression $d_i + w_i(1,x)$ represents the delay if $o_{i,j}$ is not cached (i.e., the delay of fetching object i from the original server plus the delay of transcoding from the original version to version x). On the other hand, $w_i(j,x)$ represents the delay if $o_{i,j}$ is cached. Thus, $d_i + w_i(1,x) - w_i(j,x)$ is the delay saving from caching $o_{i,j}$ in terms of the subsequent references to $o_{i,x}$. Consider the scenario in Fig. 5, for example. Suppose we would like to calculate the individual profit from caching version 2 of object i. The transcodable versions from version 2 is pointed out by bold arrows in Fig. 5. Without caching version 2, the transcoding proxy has to fetch object i from the original server and to perform transcoding to satisfy the subsequent references to versions 2, 3, or 4. The delays for them in this circumstance are all equal to 30. On the other hand, with caching version 2, the transcoding proxy only has to perform transcoding from version 2 to versions 2, 3, or 4 to satisfy the subsequent references to them with the corresponding delays being 0, 8, and 8, respectively. Thus, the delay saving from caching version 2 is $30 = 30 - 0$ for each reference to version 2, $22 = 30 - 8$ for each reference to version 3, and also, $22 = 30 - 8$ for each reference to version 4. Multiplying the delay savings by the reference rates, we can get the individual profit from caching version 2. In this example, suppose the reference rates to versions 1, 2, 3, and 4 are all equal to 10. The delay saving from caching version 2 of object i is thus equal to 740 (i.e., $10 * 30 + 10 * 22 + 10 * 22 = 740$).

As a matter of fact, however, the transcoding proxy may cache two or more versions of an object at the same time if the aggregate profit from caching them together is valuable. However, as pointed out in Section 1, the aggregate profit depends on the transcoding relationship among different versions of an object. Hence, we shall devise a procedure, *Procedure MATC (standing for Minimal Aggregate Transcoding Cost)* to find a subgraph $G'_i \subseteq G_i$ that shows the transcoding relationship which maximizes the aggregate profit when caching two or more versions of an object at the same time.

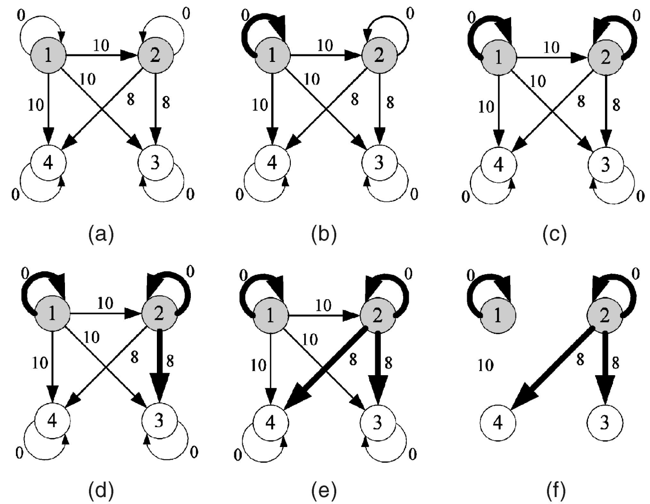


Fig. 6. The execution of *Procedure MATC*.

Procedure MATC(G, w, C)
1 $A \longleftarrow \emptyset$
2 **for** each vertex $u \in C$
3 **do** $color[u] \longleftarrow GRAY$
4 **for** each vertex $v \in V[G]$
5 **do for** each $(x,v) \in E[G]$ **and** $color[x] = GRAY$
6 **do** find the minimum $w(x,v)$
7 $A \longleftarrow A \cup (x,v)$
8 **return** $G'(V[G], A)$

The input of *Procedure MATC* consists of three arguments in which G is the corresponding weighted transcoding graph, w is the weights on the edges in G, and C is the set of the versions that the transcoding proxy tries to cache. The operations of *Procedure MATC* can be best understood by the example of Fig. 6. Suppose G and w are given by the graph in Fig. 4 and the input of C is $\{1,2\}$. As shown in Fig. 6a, lines 1-3 initialize the set A to the empty set and color vertex 1 and 2 gray, where A is the resulting edge set of the subgraph to be returned. The **for** loop in lines 4-7 finds, for each vertex $v \in V[G]$, the edge (x,v) with minimum weight $w(x,v)$ and adds (x,v) into A. In our example, since (1, 1) and (2, 2) are the only edges pointed to from gray vertexes to themselves, the iterations for vertex 1 and vertex 2 add (1, 1) and (2, 2) in A, as shown in Figs. 6b and 6c, respectively. As shown in Fig. 6d, the iteration for vertex 3 adds (2, 3) in A because $w(2,3) < w(1,3)$. The iteration for vertex 4 works similarly and adds (2, 4) in A as shown in Fig. 6e, leading to the resulting subgraph of *Procedure MATC* in Fig. 6f. It can be verified that the time complexity of *Procedure MATC* is $O(VE)$, mainly contributed by the **for** loop in lines 4-7.

With *Procedure MATC*, we can thereafter define the aggregate profit from caching multiple versions of an object.

Definition 3. *$PF(o_{i,j1}, o_{i,j2}, \ldots, o_{i,jk})$ is a function for the calculation of the aggregate profit from caching $o_{i,j1}$, $o_{i,j2}$, ..., $o_{i,jk}$ at the same time.*

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

$$PF(o_{i,j1}, o_{i,j2}, \dots, o_{i,jk}) = \sum_{v \in V[G']} \sum_{(v,x) \in E[G']} r_{i,x} * (d_i + w_i(1, x) - w_i(v, x)), \quad (2)$$

where G' is the resulting subgraph of Procedure MATC $(G, w, \{j1, j2, \dots, jk\})$.

Following the example in Fig. 5 and with the resulting subgraph in Fig. 6f, the aggregate profit from caching versions 1 and 2, $PF(o_{i,1}, o_{i,2})$, can be determined as 940 (i.e., $10 * (20 - 0) + 10 * (20 + 10 - 0) + 10 * (20 + 10 - 8) + 10 * (20 + 10 - 8) = 940$). After the aggregate profit is calculated, we can determine the marginal profit from caching $o_{i,j}$, given $o_{i,j1}, o_{i,j2}, \dots, o_{i,jk}$ are already cached.

Definition 4. *$PF(o_{i,j} | o_{i,j1}, o_{i,j2}, \dots, o_{i,jk})$ is a function for the calculation of the marginal profit from caching $o_{i,j}$, given $o_{i,j1}, o_{i,j2}, \dots, o_{i,jk}$ are already cached, where $j \neq j1, j2, \dots, jk$.*

$$PF(o_{i,j} | o_{i,j1}, o_{i,j2}, \dots, o_{i,jk}) = PF(o_{i,j}, o_{i,j1}, o_{i,j2}, \dots, o_{i,jk}) - PF(o_{i,j1}, o_{i,j2}, \dots, o_{i,jk}).$$

From the above example, we can obtain that the marginal profit from caching version 1, given version 2 is already cached, is only 200 (i.e., $940 - 740 = 200$). Finally, we can integrate all the profit functions into a *generalized profit function* as

$$PF^G(o_{i,j}) = \begin{cases} \frac{PF(o_{i,j})}{s_{i,j}} & \text{if no other version of object } i \text{ cached;} \\ \frac{PF(o_{i,j} | o_{i,j1}, \dots, o_{i,jk})}{s_{i,j}} & \text{if there are other versions } o_{i,j1}, \dots, o_{i,jk} \text{ cached.} \end{cases} \quad (3)$$

It is noted that the *generalized profit function* is further normalized by the size of version j of object i to reflect the object size factor. The rationale behind this normalization originates from one fast heuristic for solving the *knapsack problem*: Order the objects by the ratio of profit to size. Then, choose the objects with the best profit-to-size ratio, one by one, until no more objects can fit into the knapsack. Since it is shown that the caching problem is equivalent to the *knapsack problem* [4], we normalize the *generalized profit function* by the object size based on the same heuristic. The *generalized profit function* defined in (3) explicitly considers the new emerging factors in the transcoding proxy and the aggregate effect of caching multiple versions of an object. As such, we can evaluate the profit from caching a certain version of an object and, then, in view of the cost model, develop the optimal cache replacement algorithm.

3.2 Design of Algorithm *AE*

In Section 3.1, we have formulated the *generalized profit function*, PF^G. Based on this *generalized profit function*, we design algorithm *AE* in this section. The main idea behind algorithm *AE* is to first order the objects according to their values of the *generalized profit function* and then select the object with the highest value, one by one, until there is not enough space to hold any object more. The objects selected are thus the objects to be cached in the transcoding proxy and the rest of the objects are to be evicted.

Note, however, that some assumptions made in Section 3.1 to facilitate our presentation may have to be relaxed when an algorithm is designed. First, the mean reference rates of Web objects are not static, but may change as time advances. This phenomenon is quite common, especially in a news Web site where the reference rates to the news documents decrease as time goes by. Second, the delays of fetching Web objects from the original servers are not a priori known and may also change with the network condition from time to time. Third, the mean reference rates to different Web objects are not independent of one another. For instance, if there is a hyperlink from one Web object to another, the mean reference rates to them are somewhat dependent. To address these issues, we shall employ the concept of *sliding average* functions [20], [26] to practically estimate the running parameters in the Web proxy in this paper. Specifically, the *sliding average* functions we use are

$$d_i = (1 - \alpha) \cdot d_i + \alpha \cdot d_i^{new},$$
$$r_{i,j} = \frac{K}{t_{i,j} - t_{i,j}^K},$$

where d_i^{new} is the newly measured delay of fetching object i from the original server, $t_{i,j}$ is the time when the new request to version j of object i is received by the transcoding proxy and $t_{i,j}^K$ is the time when the last K request is received. Note that $r_{i,j}$ is refreshed whenever a reference to $o_{i,j}$ is received by the transcoding proxy, whereas d_i is refreshed whenever a cache miss caused by the reference to object i occurs and object i is fetched from the Web server. The effects of such estimates on the performance of algorithm *AE* depend on the selection of the fine-tuning knobs, K and α. The knob K determines how many samples should be used to estimate $r_{i,j}$, whereas the knob α determines how fast d_i adapts to the most recent sample. Clearly, the larger the value of K, the more reliable the estimation of $r_{i,j}$. However, large value of K results in significant spatial cost to keep track of the reference samples. Thus, an adequate trade off has to be made. On the other hand, the result in [20] showed that d_i can be predicted with relatively high precision (around 10 percent error) and the more adaptive the sliding average function, the better the precision of the estimate. Based on these observations, we believe that, with careful selection of the knobs K and α, algorithm *AE* is able to approximate the optimal offline algorithm, where the values of d_i and $r_{i,j}$ can be calculated from the access log in advance.

With the estimated running parameters, we can devise the pseudocode of algorithm *AE*. As listed, the algorithm takes four arguments as the inputs. C is a priority queue, which is used to hold the objects cached in the transcoding proxy. The cached objects in C are maintained by a heap data structure in nondecreasing order according to their values of the *generalized profits*. S_{now} is the accumulated size of the objects currently cached in the priority queue. $o_{i,1}$ and $o_{i,j}$ are the original and transcoded versions to be cached.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

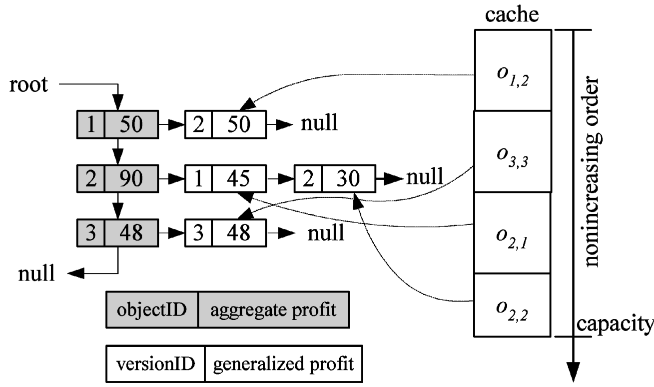


Fig. 7. Data structure for management of multiple versions of objects in cache.

Algorithm AE(C, S_{now}, $o_{i,1}$, $o_{i,j}$)
1 insert $o_{i,1}$ and $o_{i,j}$ into C
2 **for** each version of object i in C
3 **do** calculate or revise its generalized profit
4 **BuildHeap**(C)
5 **while** $S_{now} > cacheCapacity$
6 **do** exclude the first object $o_{m,n}$ from C
7 $S_{now} \longleftarrow S_{now} - s_{m,n}$
8 **for** each version of object m
9 **do** revise its generalized profits
10 **BuildHeap**(C)

Theorem 1. *The time complexity of Algorithm AE is* $O(n \log n)$.

Proof. The running time of algorithm *AE* mainly depends on two components. One is the time to execute BuildHeap() and the other is the time to calculate the generalized profit. The time complexity of BuildHeap() is $O(n \log n)$, where n is the number of objects cached in the transcoding proxy. The time complexity of the calculation of the generalized profit is $O(VE)$, where V is the number of the vertexes and E is the number of the edges in the weighted transcoding graph. Since $V, E << n$, the total time complexity of algorithm *AE* is $O(n \log n)$. □

Recall that the calculation of *generalized profit* is dependent on the fact of whether there are other versions of the object cached already. Thus, whenever we insert or delete an object into/from the cache, we have to search the entire cache for the other versions of the object and then revise the *generalized profits* for them. Such operations are, in general, very costly. Thus, it is important to design a data structure to facilitate the management of the large number of different versions of objects cached in the transcoding proxy. Consequently, we design a data structure to facilitate the operations of searching related objects and revising the *generalized profits* for them. With an illustrative example given in Section 3.3, we show the data structure proposed in Fig. 7, where the gray and white linked lists form a two-dimensional linked list. The gray linked list records the object identities and *aggregate profits* for different objects in the cache. The white linked list headed by a gray node records the version identities and *generalized profits* for distinct versions of the same object. In addition, each object in the cache has a pointer pointing to the corresponding node in the white linked list. Hence, the cached objects can be efficiently prioritized according to their *generalized profits* recorded in the white nodes.

3.3 An Illustrative Example

We use an example to illustrate how algorithm *AE* works and to demonstrate the usefulness of such a two-dimensional linked list devised. The profile of the objects used in the example is shown in Table 2. By (1), (2), and (3), we can obtain the aggregate profits and the generalized profits (e.g., $PF(o_{2,1},\ o_{2,2}) = 6 * (20 + 0 - 0)\ +\ 14 * (20 + 10 - 8)\ +\ 14 * (20 + 10 - 8)\ +\ 6 * (20 + 10 - 8) = 980$, $PF(o_{2,2}) = 14 * (20 + 10 - 0) + 14 *\ (20 + 10 - 8) + 6 *\ (20 + 10 - 8) = 840$, $PF^G(o_{2,1}) = (980 - 840)/10 = 14$, etc.). Suppose Fig. 8a is the snapshot of the current cache status in the transcoding proxy and then a new reference to $o_{3,2}$ arrives. Since no object in the cache can satisfy this reference, a cache miss occurs. Then, the transcoding proxy fetches object $o_{3,1}$ from the original server, transcodes it to $o_{3,2}$, and sends $o_{3,2}$ to the client. Meanwhile, the transcoding proxy calls algorithm *AE*, and tries to cache $o_{3,1}$ and $o_{3,2}$. The execution scenarios of algorithm *AE* are shown in Figs. 8b, 8c, and 8d. Referring to the algorithmic form of algorithm *AE* presented in Section 3.1, in line 1, the object $o_{3,1}$ and $o_{3,2}$ are inserted to the cache and the corresponding nodes in the white linked list are also created. The for loop in lines 2-3, for each version of object 3, calculates or revises the corresponding generalized profit. In line 4, since the generalized profits of some objects are changed, the heap C is rebuilt. The resulting cache is shown in Fig. 8b. It is observed that the cumulative size of the cached objects exceeds the cache capacity. The for loop in lines 5-10 starts to evict the object with the smallest generalized profit. In the first iteration, $o_{3,1}$ is evicted and, thereafter, the for loop in lines 8-9 revises the aggregate profit in gray node 3 and the generalized profits in white nodes 2 and 3. It is noted that the generalized profits of $o_{3,2}$ and $o_{3,3}$ have changed. Hence, the cache is rebuilt again in line 10. The resulting cache is then shown in Fig. 8c. In the second iteration, the algorithm works

TABLE 2
The Profile of Objects Used in Fig. 8

		reference rates				sizes				
object	d_i	$r_{i,1}$	$r_{i,2}$	$r_{i,3}$	$r_{i,4}$	$s_{i,1}$	$s_{i,2}$	$s_{i,3}$	$s_{i,4}$	G_i
1	20	10	10	10	10	12	10	8	6	Figure 5
2	20	6	14	14	6	10	8	6	4	Figure 5
3	20	6	16	12	6	10	8	6	4	Figure 5

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

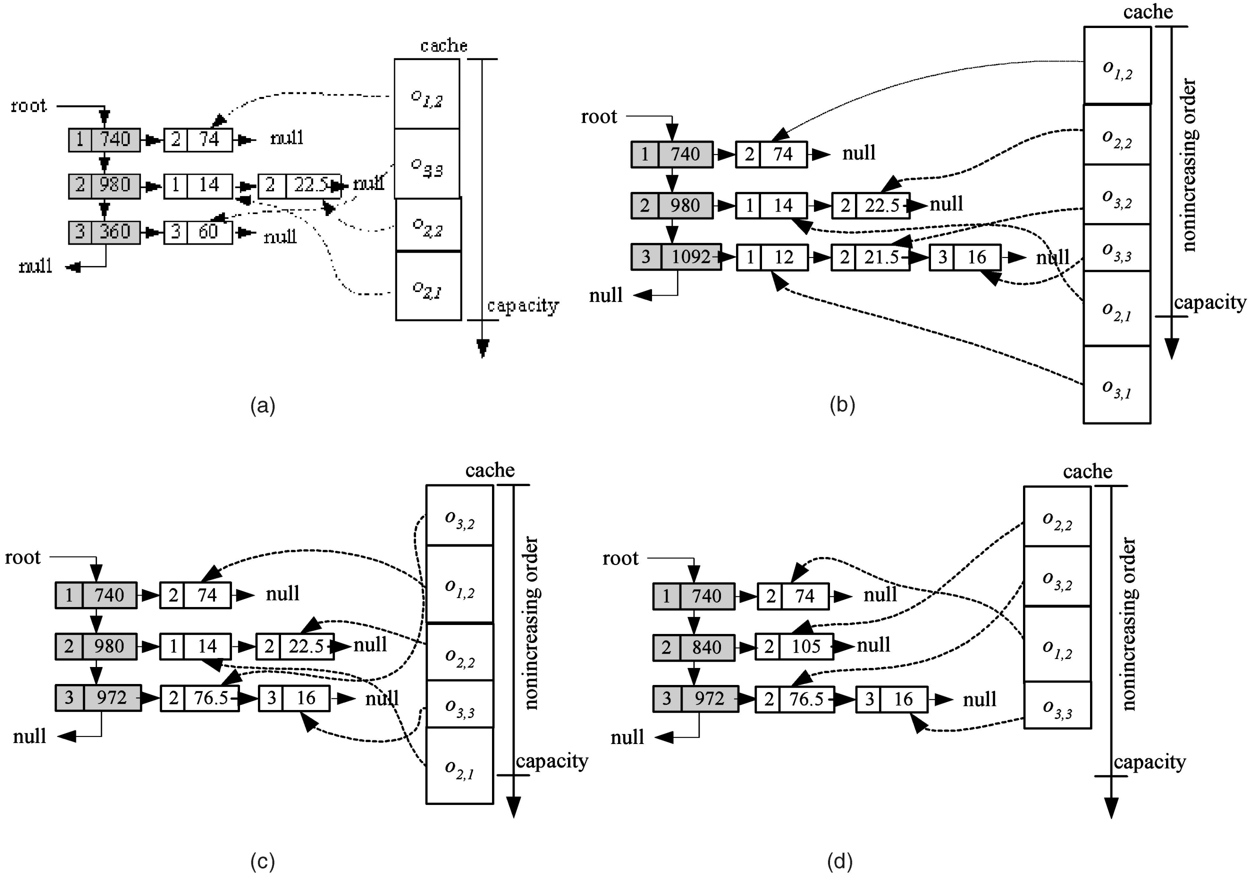


Fig. 8. The execution of algorithm *AE*.

similarly except the evicted object is $o_{2,1}$ and the cache becomes the one as shown in Fig. 8d. It is worth mentioning that algorithm *AE* inserts both the original and transcoded versions of the new coming object into the cache before enough space in the cache is freed. There are two reasons for this preinsertion. First, if we do not preinsert the new coming objects into the cache, we are not able to calculate the aggregate profit accurately. Second, by the preinsertion, we unify the admission control and space-allocation into one for loop in lines 5-10 and let the generalized profit be the criterion of the admission.

4 Performance Analysis

In this section, we will evaluate the performance of our cache replacement algorithm by conducting an event-driven simulation. The primary goal of performing an event-driven simulation is to assess the effect on the performance of our cache replacement algorithm by varying different system parameters. We will describe the simulation model in Section 4.1. The experimental results will be shown in Section 4.2.

4.1 Simulation Model

The simulation model is designed to reflect the system environment of the transcoding proxy as described in Section 2. For the client model, as in [7], we assume that the mobile appliances can be classified into five classes. The distribution of these five classes of mobile clients is modeled as a device vector of $< 15\%, 20\%, 30\%, 20\%, 15\% >$.

To generate the workload of clients' requests, we assume that the number of total Web objects is N and these N objects are divided into two types, *text* and *image*. The sizes of the *text* and *image* objects follow a heavy tailed (Pareto) distribution with the mean value of 5 KBytes and 14 KBytes, respectively [19]. Since there exists five classes of mobile clients, without loss of generality, the sizes of the five versions of each object are assumed to be 100 percent, 80 percent, 60 percent, 40 percent, and 20 percent of the original object size. The transcoding relationships among these five versions are modeled by the transcoding relation graphs as shown in Fig. 9. The transcoding delay is determined as the quotient of the object size to the

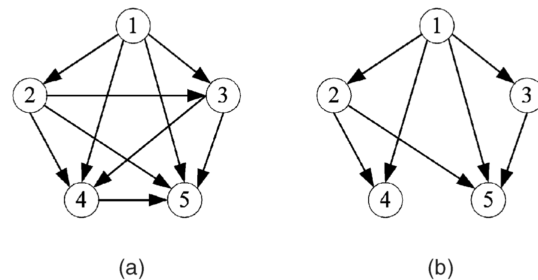


Fig. 9. Transcoding relation graph for (a) *text* object and (b) *image* object.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

TABLE 3
Parameters of the Simulation Model

Parameter	*Distribution/Default value*
Number of Web objects	$N = 10,000$
Object size	Pareto dist. ($\mu = 5$ KB for text;14 KB for image)
Object popularity (reference rate)	Zipf-like dist. ($\alpha = 0.7$)
Number of users' access sequences	$100,000$
Length of an access sequence	Inverse Gaussian dist. ($\mu = 3.5$)
Interarrival time of access sequences	Exp. dist. ($\mu = 0.5$ sec)
Client's silent time	Pareto dist. ($\mu = 30$ sec)
Cache capacity	$0.05 * (\sum$object size)
Object fetching delay	Exp. dist. ($\mu = 1.5$ sec)
Transcoding rate	20 KB/sec

transcoding rate. The popularity of the Web objects follows a Zipf-like distribution, which is widely adopted to be a model for real Web traces [3], [6], [18], [20]. As shown in [11], [14], since small files are much more frequently referenced than large files, we assume that there is negative correlation between the object size and its popularity.

After setting up the attributes of every Web object, we generate several synthetic client workloads based on serveral recent results on the Web workload characterization [11], [14], [19]. Each generated workload consists of 100,000 users' access sequences. For each access sequence, we determine the length of the sequence according to a heavy tailed (Inverse Gaussian) distribution with the mean value of 3.5. The interarrival time between two consecutive access sequences is modeled by an exponential distribution with the mean value of 0.5 seconds. The client's silent time between two successive requests within an access sequence is modeled by a Pareto distribution with the mean value of 30 seconds. The self-similarity of Web traffic requests is explained with the superimpositions of heavy tailed ON-OFF periods.

As to the model of the transcoding proxy, we set the object transcoding rate, as in [7], to be 20 KBytes per second. The default cache capacity is assumed to be $0.05 * (\sum \text{object size})$. The delays of fetching objects from Web servers are given by an exponential distribution. Since it is shown in [20] that the correlation between object size and fetching delay is relatively low, we assume that there is no correlation between Web object size and the delay of fetching it from the Web server. Table 3 provides a summary of the parameters used in the simulation model.

4.2 Experimental Results

Based on the simulation model devised, we implement a simulator using Microsoft Visual C++ package on an IBM compatible PC with Pentium III 450 CPU and 128 Mbytes RAM. Each set of the experimental results is obtained by 10 simulation runs with different seeds, where each simulation run is warmed up with 20,000 of the 100,000 users' access sequences. Confidence intervals were estimated for all simulation results and the 90 percent confidence interval was estimated to be within 10 percent of the mean.

The primary performance metric employed in our experiments is the *delay saving ratio*, which is defined as the ratio of total delay saved from cache hits to the total delay incurred under the circumstance without caching. In addition, we also use the *exact hit ratio*, the *useful hit ratio*, and the *overall hit ratio* as the secondary performance metrics in our experiments. The exact hit ratio corresponds to the fraction of requests which are satisfied by the exact versions of the objects cached (i.e., no transcoding is needed for such requests), whereas the useful hit ratio is the fraction of requests which are satisfied by the more detailed versions of the objects cached (i.e., additional transcoding is required for them). Our yardsticks are the *coverage-based* and *demand-based* extensions to the traditional LRU and LRUMIN cache replacement algorithms. We denote the *coverage-based* LRU as $\mathrm{LRU_{CV}}$ and the *demand-based* LRU as $\mathrm{LRU_{DM}}$. The *coverage-based* and *demand-based* extensions to LRUMIN are denoted as $\mathrm{LRU_{CV}^{MIN}}$ and $\mathrm{LRU_{DM}^{MIN}}$, respectively. In addition, to understand the benefit of algorithm *AE* achieved by considering the aggregate effect, we also compare algorithm *AE* with algorithm AE_0, where algorithm AE_0 is a modified version of algorithm *AE* in which the eviction priorities are determined by the profit function in Definition 1 instead of the *generalized profit function*. Recall that the profit function in Definition 1 did not consider the aggregate effect in the environment of transcoding proxy. By comparing algorithm *AE* with AE_0, the advantage of considering the aggregate effect can be observed. A list of the denotations used is given in Table 4.

4.2.1 Impact of Cache Capacity

In the first experiment set, we investigate the performance of our cache replacement algorithm by varying the cache capacity. The simulation results are shown in Fig. 10. As presented in Fig. 10a, our algorithm outperforms the other ones in terms of the primary performance metric, i.e., the delay saving ratio (DSR). Specifically, the mean improvements of the delay saving ratios over the $\mathrm{LRU_{CV}}$ and $\mathrm{LRU_{DM}}$ algorithms are 160.2 percent and 218.4 percent, respectively, whereas the mean improvements over the $\mathrm{LRU_{CV}^{MIN}}$ and $\mathrm{LRU_{DM}^{MIN}}$ are 21.3 percent and 34.8 percent. The main reason that the extensions to the LRU algorithm did not perform well is that the LRU algorithm considers neither the sizes of objects nor the delays of fetching objects from original servers. This drawback is more severe when it

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

TABLE 4
A List of the Denotations Used in Experimental Results

Symbol	*Description*
$\mathrm{LRU_{CV}}$	*coverage-based* LRU
$\mathrm{LRU_{DM}}$	*demand-based* LRU
$\mathrm{LRU_{CV}^{MIN}}$	*coverage-based* LRUMIN
$\mathrm{LRU_{DM}^{MIN}}$	*demand-based* LRUMIN
AE	Algorithm AE
$\mathrm{AE_0}$	Modified algorithm AE without consideration of the aggregate effect
γ_{FD}	The ratio of the mean object fetching delay to the mean transcoding delay

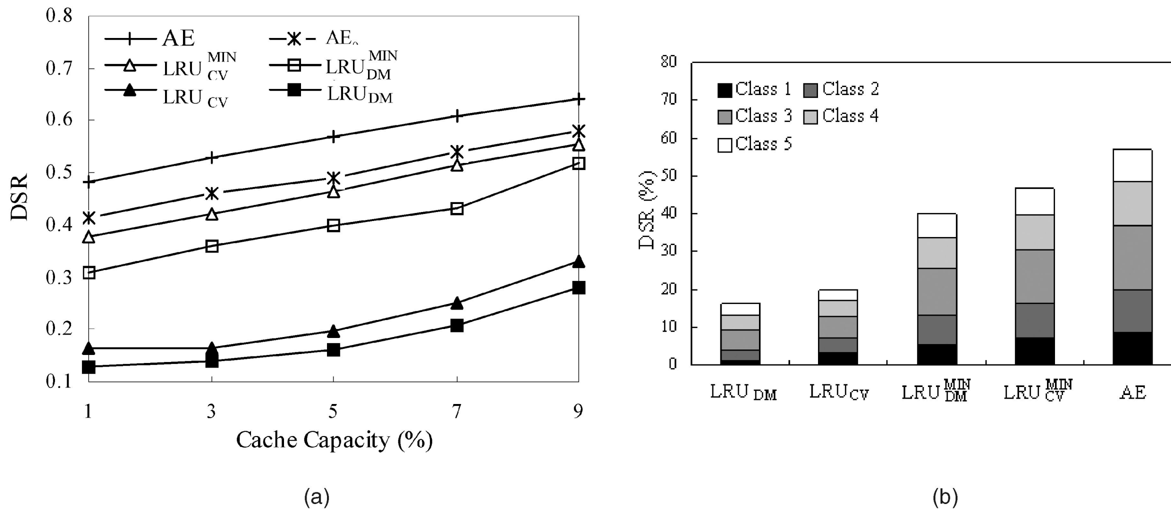


Fig. 10. (a) DSR under various cache capacities and (b) DSR of each class of client.

comes to the circumstance of the transcoding proxy. On the other hand, the advantage of algorithm *AE* over LRU_{CV}^{MIN} and LRU_{DM}^{MIN} is contributed by the devised *generalized profit function*, which explicitly considers the new emerging factors in the environment of transcoding proxies, especially the aggregate effect when observing the difference between AE and $\mathrm{AE_0}$. Fig. 10b further details the delay saving ratio obtained from each class of mobile client. It is observed that the distribution of the delay saving ratio over these five classes of mobile clients corresponds with our default setting of the device vector.

Fig. 11 provides more insights into the relationship between algorithms and performance. Observe that the exact hit ratios of the *demand-based* algorithms are higher than those of the *coverage-based* algorithms. Conversely, the useful hit ratios of the *coverage-based* algorithms are higher than those of the *demand-based* algorithms. This can be explained by the reason that *demand-based* algorithms always cache the transcoded versions to avoid repeating transcoding operations, thus leading to higher exact hit ratios. However, the *coverage-based* algorithms always cache the original versions to provide maximum coverage on subsequent requests, accounting for the reason that they perform better in terms of useful hit ratios. Although algorithm *AE* is designed to maximize the DSR rather than the hit ratios, algorithm *AE* still outperforms others in terms of the overall hit ratio. In addition, we observe in Fig. 11b, that the client class which consumes a more detailed version of object (i.e., Class 1) obtains a higher exact hit ratio than the one which consumes a less detailed version (i.e., Class 5). This phenomenon reveals that the transcoding proxy caches more more detailed versions of objects than less detailed ones. In Fig. 11c, it is noted that, since client class 1 can be only satisfied with the original version of an object, every request by client class 1 leads to either a cache miss or an exact cache hit. Consequently, the useful hit ratio of the client class 1 is 0.

4.2.2 Impact of Object Popularity

The second experiment set examines the performance of algorithm *AE* by varying the parameter α of the Zipf-like distribution. Though Zipf-like distribution is widely adopted to model the real Web traces, the parameter α of the distribution is not fully explored. As pointed out in [6], for traces from a homogeneous environment, α appears to be larger (about 0.8), whereas, for traces from a more diversified user population, α appears to be smaller (around 0.7). Further, as observed in [18], α appears to be larger than 1 at a busy Web site. Therefore, we examine the impact of various distributions of object popularity in this experiment set.

The simulation results are shown in Fig. 12. As the parameter α increases, the delay saving ratios of all algorithms increase. Clearly, as the object popularity becomes more concentrated, the overall hit ratios of all algorithms increase, which, in turn, increases the delay saving ratios. It is noted that the performance differences

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

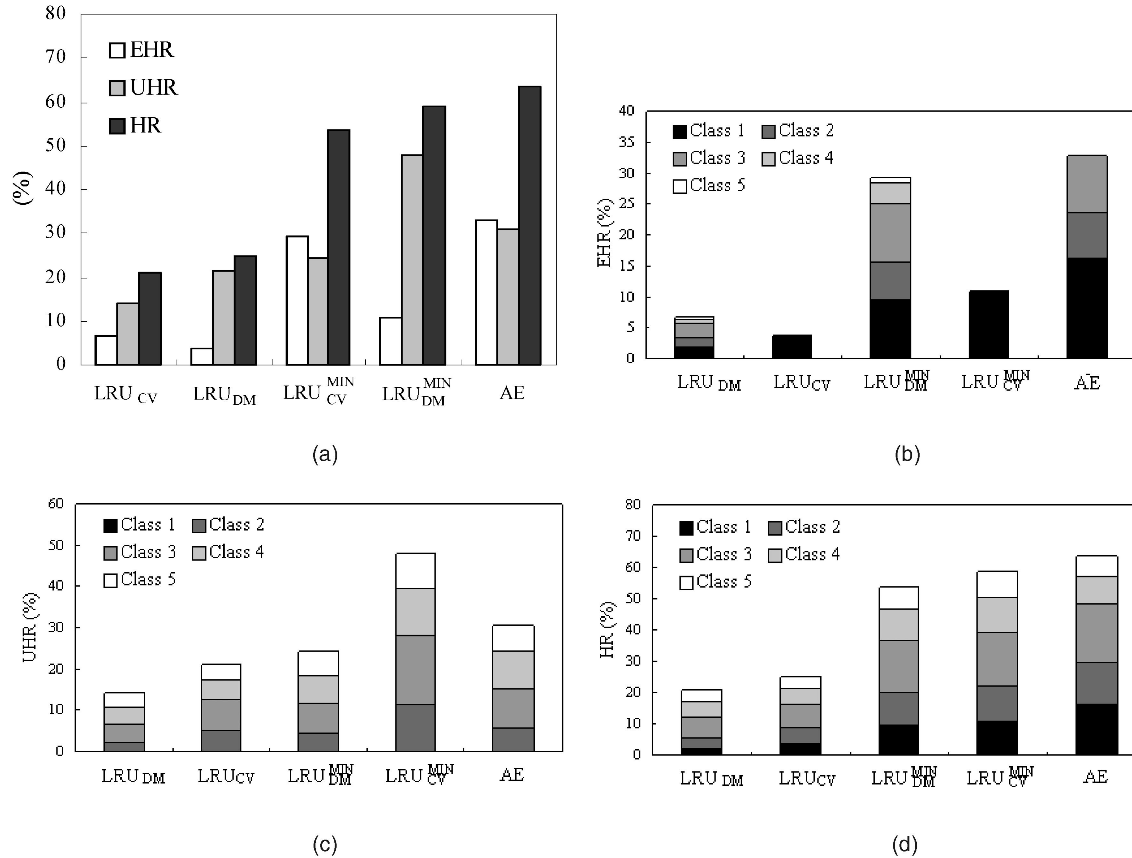


Fig. 11. (a) Hit ratios under various algorithms, (b) EHR of each class of client, (c) UHR of each class of client, and (d) HR of each class of client.

are conspicuous when the parameter of the Zipf-like distribution is low. The reason is that, when the value of α is small, the references are dispersed to different objects. Thus, the extensions to LRU and LRUMIN perform relatively poorly in this situation. On the other hand, algorithm *AE* takes the reference rate to each object into consideration and, hence, consistently outperforms the other algorithms. The mean improvements over $\mathrm{LRU}_{\mathrm{CV}}^{\mathrm{MIN}}$ and $\mathrm{LRU}_{\mathrm{DM}}^{\mathrm{MIN}}$ achieve 21.5 percent and 34.2 percent, respectively.

4.2.3 Impact of γ_{FD} Ratio

We define the ratio of the mean object fetching delay to the mean transcoding delay as the γ_{FD} ratio. Formally, the value of γ_{FD} can be determined by

$$\gamma_{FD} = \frac{mean\ fetching\ delay}{mean\ object\ size/transcoding\ rate}.$$

The third experiment set examines the performance of our algorithm by varying the value of γ_{FD}. The simulation results are given in Figs. 13a and 13b. As shown in Fig. 13a, the overall hit ratio of each algorithm almost remains the same for various γ_{FD} ratios. This is because all simulation parameters except the mean fetching delay are fixed and the mean fetching delay will not affect the overall hit ratio. However, the delay saving ratio increases as the γ_{FD} ratio increases in Fig. 13b. In addition, the performance difference between the *coverage-based* algorithm and the *demand-based* algorithms (e.g., $\mathrm{LRU}_{\mathrm{CV}}$ and $\mathrm{LRU}_{\mathrm{DM}}$) increases at the same time. The former phenomenon is more intuitive because the delay saving ratio is, of course, affected by the mean fetching delay. We explain the latter phenomenon as follows: The delay saving ratio is contributed by cache hits which can be classified into exact cache hits and useful cache hits. Although the ratios of exact and useful cache hits do not change by the γ_{FD} ratio, the weights of their contribution to DSR do change. When the γ_{FD} ratio is small, the delay incurred by the transcoding is more significant than the delay incurred by fetching from the original server. In other words, the contribution of the exact hit ratio is more weighted than that of the useful hit ratio. Thus, the *demand-based* algorithm which avoids repeating transcoding operations performs relatively

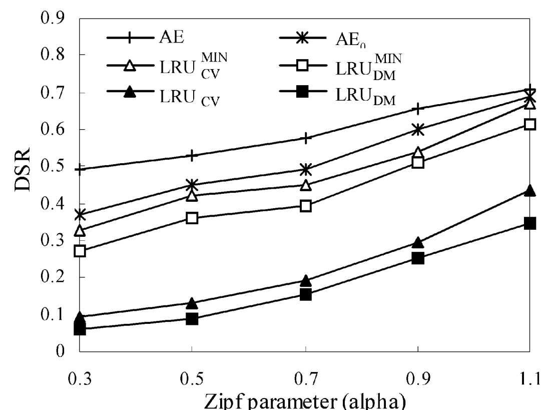


Fig. 12. DSR under various values of Zipf parameter.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

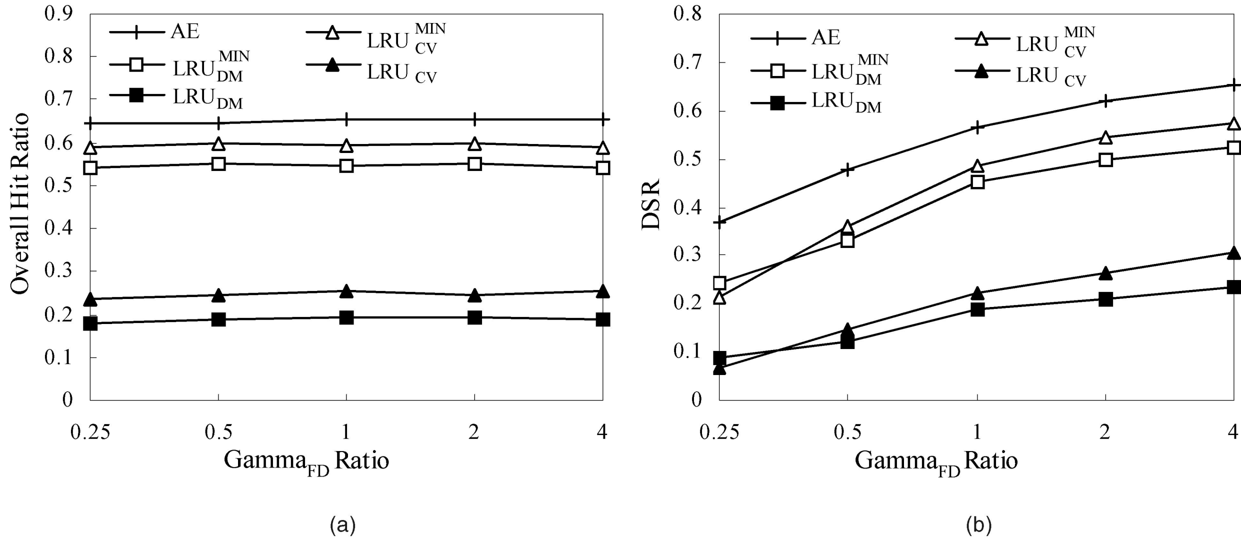


Fig. 13. (a) Overall hit ratio under various γ_{FD} ratios and (b) DSR under various γ_{FD} ratios.

well. The situation is reversed when the value of the γ_{FD} ratio is large. Unlike either the *demand-based* algorithms, which maximize the exact hit ratio, or the *coverage-based* ones, which maximize the useful hit ratio, algorithm *AE* deals with the caching problem of transcoding proxies in a comprehensive way. Consequently, the performance of algorithm *AE* remains robust against various values of γ_{FD}.

4.2.4 Impact of the Number of Client Classes

In the fourth experiment set, we examine the performance of our algorithm by varying the number of client classes, i.e., the number of the transcodable versions required. To achieve this, we vary the device vector as

$$< 100\%,\ 0,\ 0,\ 0,\ 0 >,\ < 25\%,\ 0,\ 50\%,\ 0,\ 25\% >,$$

and $< 15\%, 20\%, 25\%, 20\%, 15\% >$, meaning that the number of required transcodable versions is 1, 3, and 5, respectively.

The simulation results are shown in Fig. 14. It is observed that the delay saving ratios decrease as the number of client classes increases. This phenomenon is due to the fact that as the number of the client classes increases, the users' references will tend to be dispersed. As was explained in Section 4.2.2, as the references are more dispersed, the delay saving ratio decreases. In addition, it is noted that the delay saving ratios of the *demand-based* extensions drop more drastically than those of the *coverage-based* extensions. The reason is that, when the number of client classes is one, the *demand-based* extensions become the same as the *coverage-based* ones. However, as the number of client classes increases, the differences between them become prominent. Though, with its delay saving ratio decreasing as the number of client classes increases, algorithm *AE* still outperforms others. In fact, the slope of the degradation of algorithm *AE* is relatively smooth, indicating the robustness of the algorithm proposed.

5 Conclusions

In this paper, we developed an efficient cache replacement algorithm for transcoding proxies. We formulated a *generalized profit function* to evaluate the profit from caching each version of object. This *generalized profit function* explicitly considered the new emerging factors in the transcoding proxy, including the reference rate of each version of object, the delay of fetching each object to the transcoding proxy, the delay of transcoding from the more detailed version to the less detailed version, the size of each version of the object and, most importantly, the aggregate effect of caching multiple versions of the same object at the same time. Based on this *generalized profit function*, we proposed algorithm *AE* as the new cache replacement algorithm for transcoding proxies. In addition, an effective data structure was designed to facilitate the management of the multiple versions of different objects cached in the transcoding proxy. Using an event-driven simulation, we showed that our algorithm consistently outperforms companion schemes.

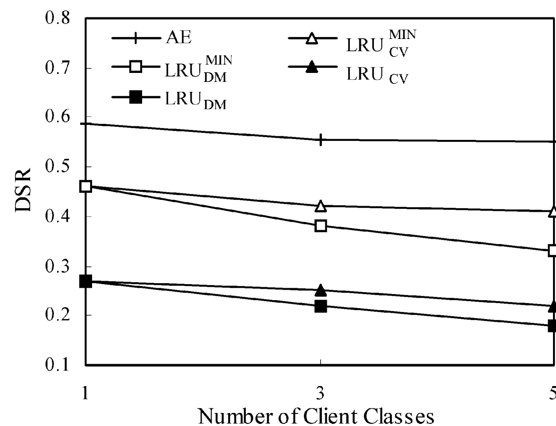


Fig. 14. DSR under various numbers of client classes.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.

REFERENCES

[1] Akamai Technologies, Inc., http://www.akamai.com/, 2002.

[2] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching Proxies: Limitations and Potentials," *Proc. Fourth Int'l World Wide Web Conf.,* 1995.

[3] C. Aggarwal, J.L. Wolf, and P.-S. Yu, "Caching on the World Wide Web," *IEEE Trans. Knowledge and Data Eng.,* vol. 11, no. 1, pp. 94-107, 1999.

[4] R. Ahuja, T. Magnanti, and J. Orlin, *Network Flows: Theory, Algorithms, and Applications.* Prentice Hall, 1993.

[5] C. Asakawa and H. Takagi, "Annotation-Based Transcoding for Nonvisual Web Access," *Proc. Fourth Int'l ACM SIGCAPH Conf. Assistive Technologies,* 2000.

[6] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-Like Distributions: Evidence and Implications," *Proc. IEEE INFOCOM,* Mar. 1999.

[7] V. Cardellini, P.-S. Yu, and Y.-W. Huang, "Collaborative Proxy System for Distributed Web Content Transcoding," *Proc. ACM Int'l Conf. Information and Knowledge Management,* pp. 520-527, 2000.

[8] S. Chandra, C. Ellis, and A. Vahdat, "Application-Level Differentiated Multimedia Web Services Using Quality Aware Transcoding," *IEEE J. Selected Areas in Comm.,* 2000.

[9] S. Chandra and C.S. Ellis, "JPEG Compression Metric as a Quality-Aware Image Transcoding," *Proc. USENIX Second Symp. Internet Techonlogy and Systems,* pp. 81-92, 1999.

[10] S. Chandra, A. Gehani, C.S. Ellis, and A. Vahdat, "Transcoding Characteristics of Web Images," *Proc. Multimedia Computing and Networking,* 2001.

[11] C. Cunha, A. Bestavros, and M. Crovella, "Characteristics of WWW Client-Based Traces," technical report, Boston Univ., Apr. 1995.

[12] R. Floyd and B. Housel, "Mobile Web Access Using eNetwork Web Express," *IEEE Personal Comm.,* vol. 5, no. 5, pp. 47-52, 1998.

[13] A. Fox, S.D. Gribble, Y. Chawathe, E.A. Brewer, and P. Gauthier, "Cluster-Based Scalable Network Services," *Proc. 16th ACM Symp. Operating Systems Principles,* pp. 78-91, 1997.

[14] S. Glassman, "A Caching Relay for the World Wide Web," *Computer Networks and ISDN Systems,* no. 27, 1994.

[15] R. Han and P. Bhagwat, "Dynamic Adaption in an Image Transcoding Proxy for Mobile Web Browsing," *IEEE Personal Comm. Magazine,* pp. 9-17, Dec. 1998.

[16] J.C. Mogul, "Server-Directed Transcoding," *Proc. Fifth Int'l Web Caching and Content Delivery Workshop,* 2000.

[17] R. Mohan, J.R. Smith, and C.-S. Li, "Adapting Multimedia Internet Cotent for Universal Access," *IEEE Trans. Multimedia,* vol. 1, no. 1, pp. 104-114, 1999.

[18] V. Padmanabhan and L. Qiu, "The Content and Access Dynamics of a Busy Web Site: Findings and Implications," *Proc. ACM SIGCOMM,* 2000.

[19] J. Pitkow, "Summary of WWW Characteristics," *World Wide Web,* vol. 2, nos. 1-2, pp. 3-13, 1999.

[20] J. Shim, P. Scheuermann, and R. Vingralek, "Proxy Cache Algorithms: Design, Implementation, and Performance," *IEEE Trans. Knowledge and Data Eng.,* vol. 11, no. 4, pp. 549-561, July/Aug. 1999.

[21] W3C Note, Annotation of Web Content for Transcoding, http://www.w3.org/TR/annot/, 1999.

[22] WAP Forum, Wireless Application Protocol Architecture Specification, 1998.

[23] WAP Forum, Wireless Application Protocol Cache Model Specification, 1998.

[24] WAP Forum, Wireless Application Group User Agent Profile Specification, 1999.

[25] S. Williams, M. Abrams, C.R. Standridge, G. Abdulla, and E. Fox, "Removal Policies in Network Caches for World Wide Web Documents," *Proc. ACM SIGCOMM,* pp. 293-304, 1996.

[26] J. Xu, Q. Hu, D.-L. Lee, and W.-C. Lee, "SAIU: An Efficient Cache Replacement Policy for Wireless On-Demand Broadcasts," *Proc. ACM Int'l Conf. Information and Knowledge Management,* pp. 46-53, 2000.

Cheng-Yue Chang received the MS and PhD degrees in electrical engineering from National Taiwan University, Taipei, Taiwan, in 1998 and 2003, respectively. He is currently a senior researcher at Philips Research East Asia-Taipei. His research interests include World Wide Web systems, multimedia networks, and home networks.

Ming-Syan Chen received the BS degree in electrical engineering from National Taiwan University, Taipei, Taiwan, and the MS and PhD degrees in computer, information, and control engineering from The University of Michigan, Ann Arbor, in 1985 and 1988, respectively. Dr. Chen is currently a professor in the Electrical Engineering Department, National Taiwan University, Taipei, Taiwan. He was a research staff member at the IBM T.J. Watson Research Center, Yorktown Heights, New York, from 1988 to 1996. His research interests include database systems, data mining, mobile computing systems, and multimedia networking and he has published more than 150 papers in his research areas. In addition to serving as a program committee member for many conferences, Dr. Chen served as an associate editor of the *IEEE Transactions on Knowledge and Data Engineering* on data mining and parallel database areas from 1997 to 2001, an editor of the *Journal of Information Science and Engineering*, an editor of the *Journal of the Chinese Institute of Electrical Engineering*, a distinguished visitor of the IEEE Computer Society for Asia-Pacific from 1998 to 2000, and program chair of PAKDD-02 (Pacific Area Knowledge Discovery and Data Mining), program vice-chair of VLDB-2002 (Very Large Data Bases), general chair of Real-Time Multimedia System Workshop in 2001, program chair of IEEE ICDCS Workshop on Knowledge Discovery and Data Mining in the World Wide Web in 2000, and program cochair of the International Conference on Mobile Data Management (MDM) in 2003, International Computer Symposium (ICS) on Computer Networks, Internet and Multimedia in 1998 and 2000, and ICS on Databases and Software Engineering in 2002. He was a keynote speaker on Web data mining at the International Computer Congress in Hong Kong, 1999, a tutorial speaker on Web data mining DASFAA-1999 and on parallel databases the 11th IEEE International Conference on Data Engineering in 1995, and also a guest coeditor for the *IEEE Transactions on Knowledge and Data Engineering* for a special issue for data mining in December 1996. He holds, or has applied for, 18 US patents and seven Republic of China patents in the areas of data mining, Web applications, interactive video playout, video server design, and concurrency and coherency control protocols. He is a recipient of the NSC (National Science Council) Distinguished Research Award in Taiwan, and the Outstanding Innovation Award from IBM Corporation for his contribution to a major database product, and also received numerous awards for his research, teaching, inventions, and patent applications. He coauthored with his students for their works which received the ACM SIGMOD Research Student Award and the Long-Term Thesis Award. Dr. Chen is a senior member of the IEEE and a member of the IEEE Computer Society and ACM.

▷ **For more information on this or any other computing topic, please visit our Digital Library at** http://computer.org/publications/dlib.

Authorized licensed use limited to: National Taiwan University. Downloaded on January 16, 2009 at 01:50 from IEEE Xplore. Restrictions apply.