ICache: A Size-Aware Cooperative Caching Architecture for Web Images

Ruixuan Li[1], Cuihua Zuo[1], Tony C. Shan[2], Zhengding Lu[1]
[1]*College of Computer Science and Technology,*
Huazhong University of Science and Technology,
Wuhan 430074, P.R.China
[2]*Wachovia Corporation, Charlotte NC 28213, USA*
E-mail: rxli@hust.edu.cn, zuocuihua@smail.hust.edu.cn,
tonycshan@gmail.com, zdlu@hust.edu.cn

Abstract*

This paper designs an efficient cooperative caching architecture for images, named iCache, which introduces super cache servers in caching system and caches images using different policies according to image size. The small-size images are stored in cache server directly, while the large-size images are divided into small blocks which are distributed in multiple cache servers in order to achieve scalability and load balance. Furthermore, the paper proposes corresponding storage policy and replacement mechanism for iCache that stores the least copies of large-size images to save space and giving them higher priority to keep in cache than small ones for increasing the byte hit rate. The simulation experiments are carried out to evaluate the performance of iCache architecture and the results show that the proposed scheme offers good search efficiency and hit rate with cooperation overhead significatively lower than that of other cooperative mechanisms.

1. Introduction

In the past decades, web caching technology has received considerable attention from the research and industry communities [1]. The caching mechanism has the advantage of bandwidth saving and service latency reducing, but existing caching systems often suffer from the limited cache space and the risk of single point of dependency [2]. There are many proposals on cooperative caching among proxies trying to address the above issues, such as [3], [4], but they are still afflicted with the similar problems in traditional cooperative systems. One more important problem is all the existing approaches don't take the object size into account. As we know, images are very popular objects on the web and their size varies from 10 byte to 10MB. Obviously, the cache servers can store more objects if the object size is smaller. We need different policies to cache objects with different sizes. Consequently, there is a need for methodologies and techniques for flexible and size-awareness cooperation among the caches in terms of object lookup, object update, as well as object storage and replacement.

The ultimate goal of our research is to design and develop techniques and architecture-level facilities for efficient cache and delivery of images in a large scale web system. The main contributions of this paper are threefold. First, we propose iCache, a cooperative caching architecture for web images that employs super cache servers to manage general caches and uses different caching strategies according to the image size. Second, we present corresponding storage and replacement policies for iCache through storing the least copies of large-size images to save space and giving them higher priority to keep in caches than small ones for increasing the byte hit rate. Third, we carry out simulation experiments to evaluate the performance of iCache architecture.

The rest of the paper is organized as follows. Section 2 describes the related work of web cache. Section 3 presents the size-aware cooperative caching architecture for images (iCache). Section 4 proposes the storage and replacement policies in iCache. Section 5 evaluates the performance of the proposed architecture with simulation experiments, followed by the conclusion in Section 6.

* This work is partially supported by National Natural Science Foundation of China under Grant 60403027, Natural Science Foundation of Hubei Province under Grant 2005ABA258, Open Foundation of State Key Laboratory of Software Engineering under Grant SKLSE05-07.

2. Related Work

Most of the recent research on cooperative caching can be classified into five categories: cooperative architecture, object retrieval, placement, replacement and consistency [5]. The cache replacement policy is a key algorithm in caching systems. Many approaches have been proposed. [6] discusses LRU algorithm which iteratively removes the least recently used object until there is enough space available. In some situations, this replacement strategy leads to removing a large number of small-size objects to store a larger one. [7] advises LFU algorithm which considers the number of times a client accessed cached objects. This algorithm has the same drawback as the LRU algorithm because it does not take object size into account. Obviously, the combined algorithms would be good choices. [8] presents some combined algorithms, such as LRU-MIN LFU-SIZE, LRU-HOT.

As to the cooperative architecture, [9] address the two predominant types: the hierarchical and distributed systems. In hierarchical architecture, cache servers are placed at multiple levels of the network. Harvest Cache [10] and Internet Caching Protocol (ICP) [11] are used in hierarchical caching architecture. But this architecture has several problems. First, a request may have to travel many hops in a cache hierarchical directory to hit the data, and the data may traverse several hops back to the clients. Second, cache misses are significantly delayed by having to traverse the hierarchy. Third, shared higher-level cache servers may be too far away from the client and the time for the object to reach the client is simply unacceptable.

While in distributed caching systems, caches are placed at the bottom level of the network and there are no intermediate caches. Only institutional caches at the edge of the network cooperate to serve for each other's missing. Query-based ICP protocol [11], digest /summary [12] based approach and hash function approach are proposed for this problem. However, using a query-based approach may significantly increase the bandwidth consumption and the experienced latency by the client, since a cache needs to poll all cooperating caches and wait for the slowest one to respond. In the second mechanism above, content digests / summaries need to be periodically exchanged among the institutional caches. The third mechanism is limited since each object has only one single copy among all cooperating caches.

3. iCache: Cooperative Caching Architecture for Images

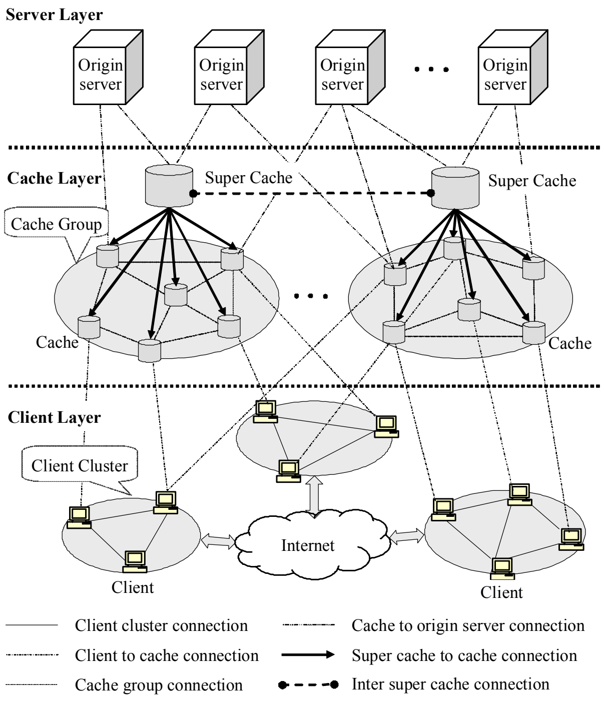

Figure 1. iCache architecture

3.1. iCache Architecture Overview

In this paper, we propose a new cooperative caching architecture for images (iCache), as shown in Figure 1. There are three layers in iCache architecture: server layer, cache layer and client layer. Origin servers are located in server layer that are usually the central sites of the network. Clients are located on the edge of the network and form different client clusters according to their geographical location. Cache layer connects the clients to origin servers. In general, most of the requests of clients will be delivered to the caches first instead of the origin servers. When it can not be satisfied in cache layer, the cache server will connect the origin servers to get the requested objects.

In iCache architecture, we employ two types of cache servers: super cache servers and general cache servers. Super cache server acts as the parent of some general cache servers. Each super cache server and its children form a group. Super cache server is an index server, storing the indices of its children which are called local indices and the indices of its neighbors which are called group indices. In each group, the super cache server will store the evicted small objects if the cache servers in this group are full. As to large-size object, it will be divided into small blocks (see Section 3.3) that are stored in the same group of cache servers in order to perform searching of large-size objects efficiently.

```
Algorithm 1. Processing requests in cache servers
 1: issue a request (object I);
 2: check local cache;
 3: if local_hit then
 4:   retrieve the object I;
 5: else
 6:   select a super cache server for transferring;
 7:   search in super cache servers (Algorithm 2)
 8:   if remote_hit then
 9:     get the object from remote cache servers;
10:   else
11:     get the object from the origin server;
12:   end if
13:     cache the object I;
14: end if
```

Figure 2. Request processing algorithm in cache servers

```
Algorithm 2. Searching in super cache servers
 1: receive a request (object I);
 2: check the super cache server;
 3: if object is found then
 4:   send remote_hit;
 5:   send the object I;
 6: else
 7:   check its local indices;
 8:   if object I is found then
 9:     send remote_hit;
10:     send the object I;
11:   else
12:     while terminating conditions are not
          satisfied do
13:       check its group indices;
14:       select a possible neighbor super cache
            server;
15:       execute 2~10 steps above;
16:     end while
17:   end if
18: end if
```

Figure 3. Searching algorithm in super cache servers

3.2. Searching Algorithms

In iCache architecture, the searching process for images in cache layer is as follows. When a client *C* requests an image, the request will be delivered to its local cache server first. If the object is not found, the request will be forwarded to a super cache server *S*. If *S* has a copy of the requested object, it sends the object back to the client. Otherwise, it will check its local indices. If there isn't, it will check its group indices and choose super cache servers which may possibly have the requested object to forward the query. This process will continue until the requested object is located or some terminating conditions are satisfied.

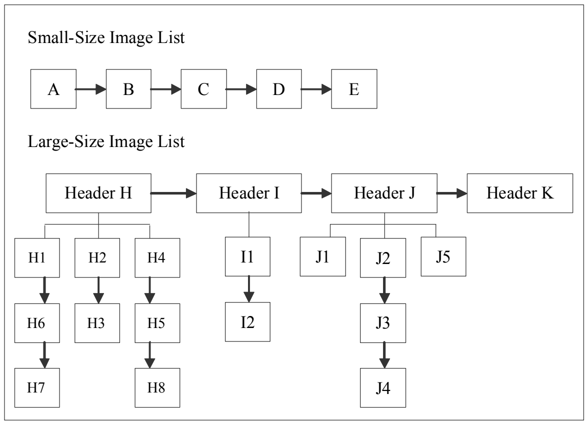

Figure 4. SSIL and LSIL

For example, we can define *n* hops to control the forwarding. If the requested object can not be found in the cache servers, the query will be redirected to the original web server. The algorithms of processing requests in cache servers (Algorithm 1) and searching in super cache servers (Algorithm 2) are shown as Figure 2 and Figure 3 respectively.

3.3. Size-Aware Image Classification

Images are one of the most popular objects on the web, and the size of most images varies from 10 bytes to 10M bytes or even more. In iCache architecture, we use different caching policies to process different images according to their sizes. The small-size images are stored in cache servers without any changes, but the large-size images will be decomposed into small blocks that are distributed in cache servers. Defining an appropriate threshold to justify whether an image is large or small is an important challenge. The choice of the threshold is up to some specific conditions of the real world circumstances. We define *m* (KB) as the threshold that varies with the average image size, the number of images, the image size differentia and etc. We will carry out experiments to verify the appropriate range for the threshold.

In this paper, we divide the large-size image into blocks with *r***r* pixels for each block, where *r* is generally equal to the power of 2 (e.g., 64*64, 128*128, 256*256 and so on). We call *r* as block size. We introduce two lists to manage the cached images: Small-Size Image List (SSIL) and Large-Size Image List (LSIL). The SSIL is a simple list storing the information of the small-size images. The LSIL is a list of the header blocks for large-size images. Each header block has one or more sub-lists linking different blocks of the large-size image. The blocks in the same sub-list are stored in the same cache server, while those in different sub-lists are stored in different cache servers.

Figure 4 shows an example of SSIL and LSIL. In Figure 4, we can see header *H* has three sub-lists, that is, the blocks of image *H* are distributed in three cache servers. The block *H*1, *H*6 and *H*7 in the same sub-list are stored in the same cache server, while *H*1, *H*2 and *H*4 in different sub-lists are stored in different cache servers respectively. Furthermore, the blocks of the same image are stored in the same cache group, which will simplify the searching of the large-size images accordingly.

4. Storage and Replacement Mechanism in iCache

4.1. Storage Policy

As mentioned above, there are two kinds of cache servers in iCache architecture: cache server and super cache server. Each super cache server stores two kinds of index tables: local indices and group indices. Each record in an index table includes two lists: SSIL and LSIL. Besides this, the super cache server will store the evicted small-size images from general cache servers in its group, and these evicted images are only stored but not indexed in super cache server in order to avoid periodical update. The super cache servers need exchange information periodically to update their group indices. In each group, the general cache servers need to update the local indices to their parent periodically. The structure of super cache server is shown in Figure 5.

In general cache servers, we store as few copy as possible for large-size images. Only if the image is always requested, the copies will be created and distributed to cache servers. As for small-size images, copies will be stored in caches in order to reduce searching time. Certainly, we need a proper algorithm to control the number of copies. Comparing with the past algorithms, we propose a moderate method for storing image copies. A hop value *k* is defined to control the number of small image copies. If a request for an image has been transferred through *n* super cache servers before the object is found, and *n*>*k*, a copy will be stored in a cache server having the sparest space in the first hop super cache server group. Otherwise, we will not store the copy of the image.

4.2. Replacement Mechanism

Since cache storage often limits, the appropriate replacement algorithm is important. Generally, the cache servers intend to evict large-size objects as soon

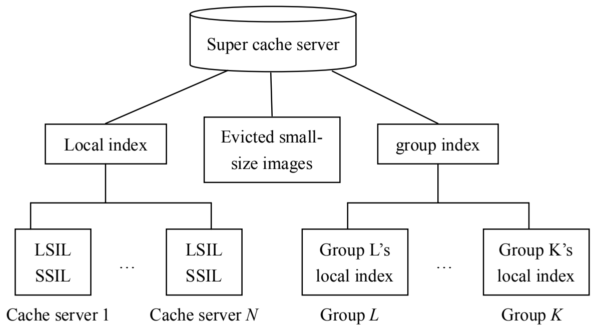

Figure 5. The structure of super cache server

as possible in order to leave more space for small objects. This makes higher hit rate, but smaller byte hit rate. In our storage policy, to prevent degradation of the byte hit rate, large-size images are given more chances to reserve. That is, if new storage is needed for storing new object, we will consider evicting the small-size images firstly. The evicted small-size images will be stored in the backup storage, super cache servers. Of course, only considering the object size is not enough. If the large images in web caches are out of date, they need to be replaced in time. Combining the most popular algorithm LRU (Least Recently Used), we suggest a new algorithm LRU-SS (Least Recently Used with Small Size) as follows.

As for every image in general cache server, an attribute value called expiring age is defined to decide which one will be evicted. Considering an image *M*, on which the last hit occurred at time *TL*, and suppose that *M* is removed at time *TR*. *w* ($0<w\leq 1$) is a parameter which gives superiority to the large-size image blocks in order to ensure that they can be preserved longer than small ones which have the same LRU. If *M* is a block of a large-size image, its expiring age equals to $(TR-TL)*w$. If *M* is a small-size image, its expiring age equals to $(TR-TL)$. Once a new storage is needed in general cache servers, we will choose the image with maximum expiring age to remove. In super caches, it stores the local indices, group indices and the evicted small-size images. While it doesn't have enough space to store new evicted small images, FIFO (First In, First Out) policy will be used to remove small-size images.

5. Performance Evaluation

To evaluate the performance of the proposed schemes, we develop a web cache system simulator for images. The simulator can be configured to simulate different caching architectures such as distributed, hierarchical and iCache architectures. It can also simulate

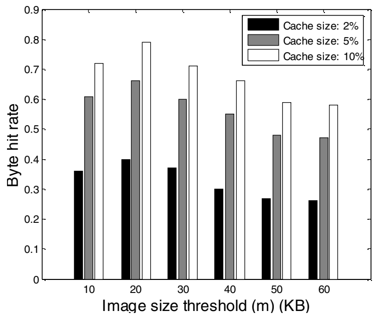

Figure 6. Impact of image size threshold on byte hit rate

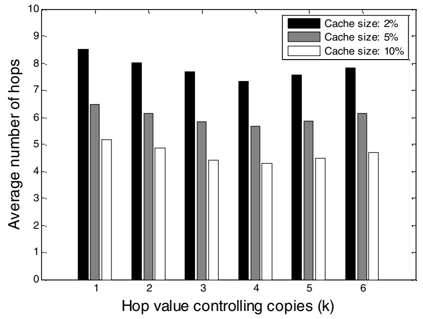

Figure 7. Impact of different hop value controlling copies on average number of hops

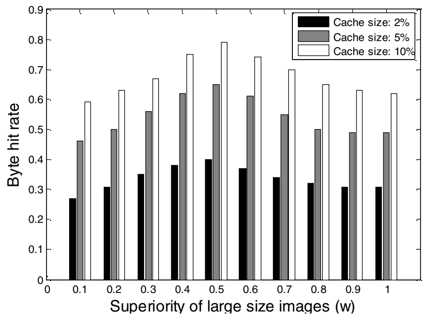

Figure 8. Impact of superiority parameter on byte hit rate

size sensitive image replacement schemes. The data set is the captured images from the campus web proxy of Huazhong University of Science and Technology. We get 10,000 images and 100,000 requests as records of our data set. For each record, we extract the URL and the image size. The URL includes the path and image name so that the images can be uniquely identified. The average image size of these 10,000 images is about 10KB. The data set is used on the architecture given in Figure 1 and the performances of the three caching systems are compared according to the average number of hops, hit rate and byte hit rate. The image size threshold, the hop value controlling copies, the superiority of large-size image, the number of caches and the cache size in the iCache system are parameters and can be varied.

5.1. The Effectiveness of iCache Scheme

In the first set of experiments we study the parameter properties of the iCache mechanism. As we analyzed in Section 3.3, the image size threshold (m) varies with different data sets and relates to the average image size. As to our data set used in experiments, we test different value of the threshold on byte hit rate and the result is shown in Figure 6. From the figure we can see, with different cache sizes 2, 5 and 10 percent, it performs better while the image size threshold (m) is 20KB, twice as large as the average image size (10KB) in our experiment. We carry out the following experiments by using 20KB as the choice for threshold m and 256*256 as the block size for dividing large-size images (r=256).

The second experiment studies the effect of the hop value controlling copies (k) on the average number of hops delivering requests. The bar graphs in Figure 7 show the average number of hops getting the target image with different value of k. We can see the average number of hops will be the least when k equals 4, which is the parameter setting for the following experiments. The frequently image copying among caches will not only increase the load of the cache system, but also increase the average number of hops. This is because the capacity of the caches is limited and the intended image may be replaced before it is accessed due to frequently copying.

In the third experiment, we study the impact of superiority parameter (w) on byte hit rate in iCache scheme, as shown in Figure 8. At the value of w=0.5 iCache scheme yields the highest byte hit rate values. While increasing the value of w, more large-size images will be probably replaced and the byte hit rate will decrease accordingly. On the contrary, when the value of w is smaller than 0.5, large-size images are intended to be preserved in caches so that more small-size images will be replaced. At the same time, those infrequently visited large-size images will hold the rare cache space and thereby it will reduce the byte hit rate. In the following experiments, we set w=0.5.

5.2 The Performance of Different Schemes

To compare the performance of iCache scheme with the two popular caching systems: distributed and hierarchical systems, we carry out the following experiments and use the parameter settings as described above. We use LRU as the replacement algorithm in distributed and hierarchical systems, and LRU-SS algorithm in iCache system.

The graph in Figure 9 shows the average number of hops for 100,000 requests at various *cache* sizes. We can see that, as the cache size increases from 1 to 10 percent of the total size of all images, the average number of hops decreases from 8.2 to 4.3 in iCache scheme. It is clear that the decrease is more obvious than the hierarchical and distributed systems. This is because there are only two levels - super cache level and general cache level in iCache architecture, and all

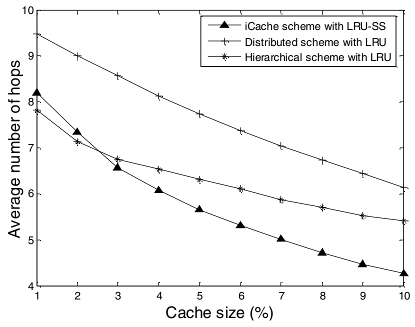

Figure 9. Average number of hops under different cache schemes

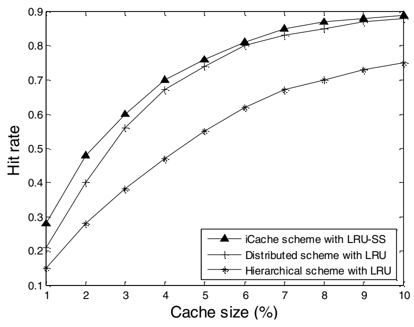

Figure 10. Hit rate under different cache schemes

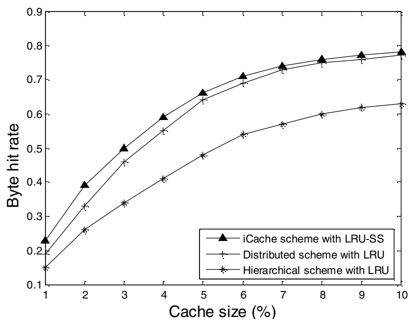

Figure 11. Byte hit rate under different cache schemes

requests from clients are transferred to a super cache server and routed among super cache servers, which will reduce the number of hops considerably.

As shown in Figure 10 and Figure 11, the value of hit rate and byte hit rate increase logarithmically as cache size increasing from 1 to 10 percent in three systems. With the cache size from 1 to 4 percent, the hit rate increases rapidly from less than 0.3 to nearly 0.7, while the byte hit rate from 0.2 to 0.6, in iCache and distributed architectures. The hit rate and byte hit rate change a little when the cache size is larger than 8 percent. The hit rate and byte hit rate of the hierarchical system are always the lowest in the three caching systems.

6. Conclusion

In this paper, we investigate the efficient caching policies and present iCache architecture of cooperative caching for images according to their size. The small-size images are stored in cache servers unchanged, while the large-size images will be divided into small blocks and stored in distributed caches. The experiments show that iCache system is more efficient than the existing distributed and hierarchical schemes. In the future work, we intend to take hot images into account since images on the Web are accessed with different frequency.

References

[1] G. Barish and K. Obraczka, "World Wide Web Caching: Trends and Techniques", *IEEE Communications Magazine*, Vol.38, No.5, 2000, pp.178-185.

[2] J. Wang, "A Survey of Web Caching Schemes for the Internet," *ACM Computer Communication Review*, 1999, 29(5): 36-46.

[3] J.-M. Menaud, V. Issarny, and M. Banatre, "A Scalable and Efficient Cooperative System for Web Caches", *IEEE Concurrency,* Vol.8, No.3, 2000, pp. 56-62.

[4] H. Che, Y. Tung, and Z. Wang, "Hierarchical Web Caching Systems: Modeling, Design and Experimental Results", *IEEE Journal on Selected Areas in Communications,* Vol. 20, No. 7, September 2002, pp. 1305-1314.

[5] L. Ramaswamy, L. Liu and A. Iyenagar, "Cache Clouds: Cooperative Caching of Dynamic Documents on Edge Networks". *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS 2005)*, Columbus, OH, USA, June 2005, pp.229-238.

[6] K. Psounis, B. Prabhakar, and D. Engler, "A Randomized Cache Replacement Scheme Approximating LRU", *Proceedings of 34th Annual Conference on Information Sciences and Systems*, Princeton University, 2000.

[7] K. Psounis, B. Prabhakar, "Efficient randomized web-cache replacement schemes using samples from past eviction times," *IEEE/ACM Transactions on Networking*, Vol. 10, No. 4, August 2002, pp. 441-454.

[8] M. Rabinovich and O. Spatscheck, "Web Caching and Replication", *Addison Wesley Professional*, December 2001.

[9] P. Rodriguez, C. Spanner, and E. W. Biersack, "Analysis of Web Caching Architectures: Hierarchical and Distributed Caching", *IEEE/ACM Transactions on Networking*, Vol. 9, No. 4, August 2001, pp. 404-418.

[10] A. Chankhunthod, P. Danzig, and C. Neerdaels, et al, "A Hierarchical Internet Object Cache", *Proceedings of the USENIX 1996 Annual Technical Conference*, San Diego, California, Jan. 1996, pp. 153-163.

[11] D. Wessels and K. Clasffy, "Application of Internet Cache Protocol (ICP), Version 2", *Internet Engineering Task Force, Internet Draft: draft-wessels-icp-v2-appl-00. Work in Progress*, May 1997.

[12] L. Fan, P. Cao, J. Almeida, and A. Z. Broder, "Summary Cache: A Scalable Wide-Area Web Cache Sharing Protocol", *IEEE/ACM Transactions on Networking*, Vol. 8, No. 3, June 2000, pp. 281-293.