개체 타입별 분할공간을 이용한 웹 프락시 캐시의 대체 알고리즘

정회원 이 수 행*, 두 현 재*, 최 상 방*

Web Proxy Cache Replacement Algorithms using Object Type Partition

Soo-haeng Lee*, Hyeon-jae Doo*, Sang-bang Choi* *Regular Memers*

요 약

웹 캐시는 클라이언트와 서버사이에 위치하여, 대리자의 역할을 하는 프락시 서버의 기능적측면의 다른 이름이다. 클라이언트와 프락시 서버사이는 주로 LAN을 통해 연결되므로 넓은 대역폭을 갖게 되지만 웹 캐시의 저장공간은 한정되어 있으므로 웹 캐시내에 존재하는 개체들은 대체 알고리즘이라는 어떤 규칙에 의해 새 개체를 위한 공간확보를 위해 삭제되게 된다. 대부분의 대체 알고리즘들은 성능평가기준인 히트율과 바이트히트율 중 단지 하나의 성능평가기준만을 만족하던지, 때때로 어느 한 기준도 만족시키지 못한다. 본 논문에서 우리는 히트율과 바이트히트율 모두에서 높은 성능을 나타내는 두 가지의 대체알고리즘을 제안한다. 첫 번째 알고리즘은 기본모델로서, 캐시를 파일타입에 따라 적절히 분할시킨다. 두 번째 알고리즘에서는 2-레벨의 캐시구조를 사용한다. 상위레벨캐시는 기본 모델에서처럼 운용되고, 하위레벨캐시는 공유공간으로서 모든 타입의 개체들을 집합적으로 수용하게 된다. 트레이스-드리븐 시뮬레이션을 사용하여 히트율과 바이트히트율을 측정함으로써 제안 알고리즘들의 성능을 평가하였다.

ABSTRACT

Web cache, which is functionally another word of proxy server, is located between client and server. Web cache has a limited storage area although it has broad bandwidth between client and proxy server, which are usually connected through LAN. Because of limited storage capacity, existing objects in web cache can be deleted for new objects by some rules called replacement algorithm. Hit rate and byte-hit rate are general metrics to evaluate replacement algorithms. Most of the replacement algorithms do satisfy only one metric, or sometimes none of them. In this paper, we propose two replacement algorithms to achieve both high hit rate and byte-hit rate with great satisfaction. In the first algorithm, the cache is appropriately partitioned according to file types as a basic model. In the second algorithm, the cache is composed of two levels; the upper level cache is managed by the basic algorithm, but the lower level is collectively used for all types of files as a shared area. To show the performance of the proposed algorithms, we evaluate hit rate and byte-hit rate of the proposed replacement algorithms using the trace driven simulation.

I. Introduction

The fact that objects of the same file type on WWW have low coefficients of size variation was revealed by the study of Martin F. Arlitt [1]. In other words, it means the fact that objects of the

* 인하대학교 전자공학과 컴퓨터구조 및 네트워크 연구실(sangbang@inha.ac.kr)
논문번호 : 010369-1204, 접수일자 : 2001년 12월 04일
※ 이 논문은 2001학년도 인하대학교의 지원에 의하여 연구되었음. (INHA-22006)

same type have similar sizes. In this paper, we suggest two algorithms for web cache replacement, the basic algorithm and the advanced algorithm. The basic algorithm, which is referred to as TYPE algorithm, partitions the whole cache space into several sections to store objects of the same file type instead of simply managing one big cache space for all objects of various types and sizes. The class is defined as the assemblage of object types with similar file sizes, and each class is assigned to a separate section that is appropriately partitioned. The advanced algorithm, which is called 2-LEVEL algorithm, has two-level structure, i.e., upper level cache (L1) and lower level cache (L2). The L1 cache is managed by the TYPE algorithm. However, the lower level L2 cache is collectively used for all types of files as a shadow cache.

Multimedia objects are sometimes several tens of megabytes in size. When they make an entry into the limited cache storage, they can sacrifice thousands of small-sized documents. And those removals usually result in low hit rates. In the TYPE algorithm, sections of reasonable space are assigned exclusively for such small objects to guarantee higher hit rates. For examples, one section is assigned for document class (text, html) and the other section is for graphic class (gif, bmp). In each partitioned section, LRU (least-recently used) algorithm, which removes the object that has been unused for the longest time from a cache, can be used. LRU algorithm was already proved to have very high byte-hit rate in usual file cache system. Since objects in the same class are similar in size and assigned to the same section in proposed TYPE algorithm, LRU replacement algorithm can provide high byte-hit rate for a given amount of space. A different replacement algorithm can be used for each section to optimize the byte-hit rate.

Separate sections are also assigned for classes of large objects to obtain high byte-hit rates. In general, the high hit rate for each class provides a high byte-hit rate. We also introduce two-level cache structure to enhance the performance of TYPE algorithm. In a certain circumstance, small area can be assigned to a class. However, when various objects are suddenly referenced by clients, the corresponding section will have very high miss rate because of the limited area. In the proposed 2-LEVEL algorithm, an extra space is provided as the shared area to accommodate those removed objects from the L1 cache. The L2 cache is collectively used for all types of files as a shelter space. The goal of this paper is to design algorithms to improve performance in terms of both hit rate and byte-hit rate. In the proposed algorithm, the hit rate is obtained from partitioning of cache and the byte-hit rate is improved by LFU-DA (least-frequently used with dynamic aging) algorithm, which removes the object that has the smallest key value obtained from the frequency of references and an aging factor in each partitioned section [2].

This paper is organized as follows. We explain characteristics of HTTP and performance of cache replacement algorithm for proxy server in Section 2. In Section 3, related researches about replacement algorithms and web access characteristics are discussed. We show limitations of current replacement policies and propose two replacement algorithms to overcome those limitations in Web proxy cache in Section 4. In Section 5, we explain trace driven simulations and compare the performance of proposed algorithms with other ones. Finally, we conclude our paper in Section 6.

Ⅱ. Preliminaries

HTTP is the object unit transfer protocol. Thus, the entire object is sent in a session since whole data are regarded as one unit. Partial transfer function doesnt exist in HTTP. This simplicity of HTTP contributes to popularization of WWW [3]. Since its impossible to transfer a part of object, storage and replacement of a page unit with fixed size have no meaning in this circumstance. Thus, in proxy server, it is inevitable to employ a cache management policy based on variable-sized object instead of cache management for fixed-sized page.

The hit rate has employed as a measure of the performance of the memory hierarchy and is defined as the fraction of memory accesses found in the upper level. Thus cache hit rate is generally used as the most popular metric to evaluate the performance of cache structure or replacement algorithm. And this concept can also be used to measure the performance of web cache replacement algorithm for requested objects. In proxy server, the hit rate is defined as the number of objects found in the cache as a percent of total requested objects since HTTP is the transfer protocol in object unit.

However, since the stored objects in proxy cache have variable sizes, another performance metric has to be employed to evaluate the web cache operation properly. That is, we need byte-hit rate and is defined as the number of bytes that the proxy cache served directly as a percent of the total number of bytes for all requests. We can intuitively realize that these two performance metrics have the same value for general file system or database cache because page or data block is the unit of data with a fixed size.

Ⅲ. Related Researches

Data cache in database or file system has been a subject of many studies. Along the development of WWW, traditional cache algorithms have been adopted for the proxy cache. And a great deal of effort has also been made to obtain more suitable algorithms for web objects. Among earlier studies, S. Williams[4] classified the existing cache algorithms by sorting keys used for the decision of removal order. The four basic sorting keys are considered in the classification of cache replacement algorithm; the size of a cached document in bytes (size), the time that a document entered the cache (etime), the time of last document access (atime), and the number of document references (nref). Table 1 shows the summary of four

Table 1. Removal algorithms defined by sorting key.

Algorithm	Sorting Key	Removing Order
SIZE	Size of a cached document in bytes	Largest file
FIFO	Time of document entered the cache	Oldest entered file
LRU	Time of last document access	Least recently used file
LFU	Number of document references	Least referenced file

representative replacement algorithms classified by sorting key.

The SIZE algorithm that uses the size of document as a sorting key can increase the number of objects in cache by making space for small files. It removes big-sized file first as shown in Table 1. Moreover, most user requests make references for small-sized files as shown in Figure 1. Thus, SIZE algorithm that sacrifices a big file for many small files can have higher hit rate.

The performance metrics of cache replacement algorithms are hit rate and byte-hit rate as we explained in Section 2. The SIZE algorithm has best performance in hit rate as we explained above. However, it has very poor performance for byte-hit rate since one miss of a large object generates so many byte misses. Therefore, we have to design a new replacement algorithm to have a close value to SIZE algorithm for the case of hit rate.

The characteristic of web accesses is very similar to the access pattern of main memory. In other words, it has temporal locality; if an object is referenced, it will be referenced again soon. Thus, the object replaced should be the one that

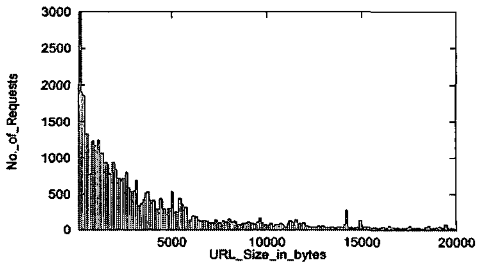


Fig. 1. Distribution of document sizes in Virginia Tech trace data.

has been unused for the longest time. The LRU algorithm, which uses time of last document access as a sorting key, shows high byte-hit rate since it keeps those objects that can be referenced soon with high probability without regard to the size of objects [5]. Therefore, we have to design a new replacement algorithm that has a close value to LRU algorithm for the byte-hit rate.

The LFU (least frequently used) algorithm requires that the object with the smallest reference count be replaced. The reason for this selection is that an actively used object should have a large reference count value. However, this algorithm suffers from the situation in which an object is used heavily during the initial phase, but then is never used again. Since it was used heavily, it has a large count and remains in a cache even though it is no longer needed [6].

Ⅳ. Proposed Replacement Algorithms

4. 1 Characteristics of File Access Pattern on WWW

The fact that WWW objects have variable sizes has been found in many preceding studies. And the variable sizes stem from diverse kinds of file type provided by servers. For an extreme example, several hundred bytes of HTML document file can be compared with several hundred megabytes of MPEG multimedia file.

The characteristic of web objects was studied more systematically by Martin F. Arlitt [1] in 1996. In this paper, the class is defined as the assemblage of some related object types with similar file sizes. Tables 2 and 3 show CoV (Coefficient of Variation) values classified by file types for the web documents from the two university trace data.

Table 2. Classification of documents by file types.

Class	File Type
HTML	.html file types
Images	.gif, .jpeg, or .xbm file types
Sound	.au or .wav file types
Video	.mpeg, .avi, or .mov file types
Dynamic	.cgi or .perl file types
Formatted	.ps, .dvi, or .doc file types

Table 3. Coefficient of variation of web documents from Saskachewan and Calgary University traces.

Saskachewan data

	Html	Image	Sound	Video
Mean Size CoV	5,447 2.19	5,980 2.77	84,154 2.62	3,602,176 2.29
	Dynamic	Formatted	Others	All Files
Mean Size CoV	3,969 2.91	36,055 0.08	22,441 11.30	5,970 11.19

Calgary data

	Html	Image	Sound	Video
Mean Size CoV	3,929 1.86	13,971 3.95	258,196 1.49	496,992 1.60
	Dynamic	Formatted	Others	All Files
Mean Size CoV	4,702 1.26	305,444 2.77	27,112 2.77	13,997 8.01

(CoV : Coefficient of variation)

From Table 3, we can find that the CoV of each file type has small value although the CoV of all files has quite large value. In other words, we can say that files that are classified into the same class are similar in size. And it is a very important characteristic for the proposed replacement algorithms since we partition cache space into sections of appropriate sizes according to file classes.

4. 2 TYPE Algorithm

Partitioning of cache has already been used to expand the bandwidth between CPU and memory. In most high performance microprocessors, the cache is partitioned into two sections, i.e., instruction cache and data cache, according to their usage. Our approach to new replacement algorithm also partitions the proxy cache. However, we partition cache space into several sections on the basis of file type instead. The logical ground for our approach is that each partition can provide a space of reasonable size for the class assigned to it. And the LRU algorithm, which has high performance for objects of similar size, is employed for each section. As a result, the proposed algorithm can achieve both very high hit rate and byte-hit rate. In this paper, partition and section will be used interchangeably. To the best of our knowledge, there is no such replacement algorithm proposed thus far. Figure 2 shows the visual description of the proposed replacement algorithm.

The proposed strategy is simply referred to as TYPE algorithm in this paper since the cache is partitioned according to file type. And, in order to employ this TYPE replacement algorithm, we must consider following several factors; the number of partitioned sections, the size of each section, file types assigned to each section, and replacement algorithm used in each section.

The procedure of TYPE algorithm can be described in detail as follows. The classification of the file types that will be located in each partition has to be formed. Some related file types can be grouped into 7 classes such as graphic, text/html, audio, video, CGI, formatted, and unknown. And each class is assigned to one section exclusively. The size of each partition can be derived from the amount of transfers in bytes for responses of each class as follows.

$$transfer_{class\ i} = request\ rate\ per\ second_{class\ i} * mean\ transfer\ size_{class\ i}$$

The initial partition size for class i can be assigned according to its popularity or an average of $transfer_{class\ i}$ values for some time. The advantage of using class-based partition is that big-sized files like sound or multimedia do not delete many small-sized files unconditionally unlike SIZE algorithm.

TYPE algorithm can guarantee high hit rate close to that of SIZE algorithm because reasonable cache spaces are always allocated to the classes of small-sized files like text/html. The LRU algorithm removes the object that has not been used for the longest time from the corresponding section. The LRU algorithm was

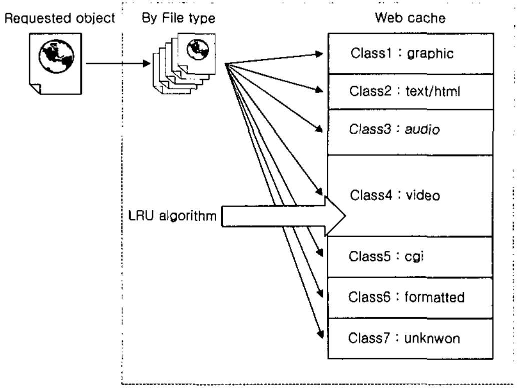


Fig. 2. Logical structure and operation flowof web proxy cache using TYPE replacement algorithm.

already proved to have very high hit rate in conventional file cache systems. And objects in the same class are similar in their sizes and assigned into the same partition. The maximized hit rate in each partition for objects of similar sizes produces the optimal byte-hit rate in general, even though it is not always guaranteed. And the division ratio for each partitioned section can be determined by accumulated decision factors when they reach critical points, or periodically.

The realization of the proposed TYPE algorithm is intuitively clear and not difficult. Cache manager keeps the list of cached objects and searches the list when a new request comes in. If the requested object is not found, a cache miss occurs and the manager directly requests the object from a web server. This process causes a file transfer and the manager is able to know the file type. Transferred file must be located in a

```
TYPE Algorithm
Input: Requested object from a client;
Output: Retrieved object from a cache;
Determine the class of requested object;
if the new object is not in the corresponding section;
    then retrieve the requested object from the web server;
        if there is no enough room in the section
            then while (no enough room for the new object)
                Remove the LRU object;
Add the new object to the corresponding section;
Return requested object to the client;
```

Fig. 3. Pseudo code describing TYPE algorithm.

section that is responsible for its class. If the section needs to make a space for the newly entered object, the LRU algorithm can be applied to the section. The TYPE algorithm can be described by pseudo code as shown in Figure 3.

4. 3 2-LEVEL Algorithm

In the second algorithm, we enhance TYPE algorithm by employing two-level cache structure and an individual replacement algorithm for each level. In the TYPE algorithm, a small area can be assigned to a class in a dynamic web environment. When various objects of the same class are suddenly referenced by many clients in a local domain, the corresponding section will create very high miss rates because of the limited space assigned to the class. In the proposed 2-LEVEL algorithm, an extra space called L2 cache is provided as the shared buffer to accommodate those objects removed from the L1 cache. The L2 cache is collectively used for all types of files as a shared area. Such a cache is often referred to as a victim cache in a conventional cache of microprocessor system.

In the proposed 2-LEVEL algorithm, the hit rate is obtained from partitioning of cache according to object classes in the L1 cache, and the byte-hit rate is improved by the frequency- based algorithm with an dynamic aging factor employed in both L1 and L2 caches. The proposed algorithm can be explained in detail as follows.

Two-level structure: The cache is composed of two levels; the L1 cache is appropriately partitioned for each file class and objects of the same class are assigned into the corresponding portion of the L1 cache as in the TYPE algorithm. The L2 cache is the combined cache that accepts all types of files in it.

Type-based partitioning in the L1 cache: The L1 cache is partitioned into seven partitions for the same number of classes, i.e., graphic, text/html, audio, video, CGI, formatted, unknown. As revealed by the study of Martin F. Arlitt [1], objects of the same file type on WWW have low coefficients of size variation. In other words, objects of the same type have similar sizes. Therefore, if sections of reasonable space are assigned for each class, it guarantees high hit rates.

LFU-DA for each section of L1 cache and for L2 cache: We employ the least-frequently used with dynamic aging (LFU-DA) algorithm for each section of L1 cache since it shows great performance for byte-hit rate. LFU-DA is also used in L2 cache.

The LFU policy maintains a frequency count for each object in the cache, and replaces the least frequently used object. In the LFU policy, the objects with initial high access counts can kept in the cache for a long time and pollute the cache, even though they are not popular any more. The LFU-DA is also frequency-based policy and retains the most popular objects regardless of its size [2]. However, to solve the cache pollution problem, the LFU-DA calculates the key value of an object using the frequency count and the inflation factor for aging objects. The key value for each object i, key[i], can be obtained from the following formula.

$$key_{LFU-DA}[i] = cost[i] * frequency\ count[i] + inflation\ factor[i],$$

where, the cost[i] is the cost to bring object i, the frequency count [i] is the reference count of the object that has been accumulated since it entered the cache, and the inflation factor[i] is the key value of the previous object that has been replaced by this object (i.e., object i). If object i has not replaced any object when it enter the cache, inflation factor[i] is set to 0. The LFU-DA policy may prove useful in caching environment where the frequency is an important characteristic, and achieve high byte-hit rates.

On the other hand, the GreedyDual-Size (GDS) is size-based algorithm and replaces the object with the smallest key value for a certain utility function, where the function considers cost associated with bringing object, object size, and aging factor in the calculation of key values [5]. However, it does not take into account how many times the object was accessed in the past. Greedy-dual size with frequency (GDSF) considers the reference frequency in the utility function to enhance the performance of GDS [2]. The GDSF calculates the key value for each object i as follows.

$$key_{GDSF}[i] = (cost[i] * frequency\ count[i]) / size[i] + inflation\ factor[i]\ ,$$

where the size[i] is the size of object i in bytes. Both GDS and GDSF provide very high hit rate since they are size-based algorithm.

In the proposed 2-LEVEL algorithm, we employ the LFU-DA policy to replace an object in each section. And each section is assigned to one class that has objects with similar sizes. Thus, the value of *key*$_{LFU\text{-}DA}$*[i]* is very similar in relative size to that of *key*$_{GDSF}$*[i]* in each section. In the GDSF policy, the *size[i]* does not play an important role in calculating values of *key*$_{GDSF}$ for objects of similar sizes. In other words, the order of LFU-DA key values is very similar to that of GDSF key values for the whole objects of one section. As a result, the 2-LEVEL can achieve very high hit rate and byte-hit rate simultaneously.

Figure 4 describes the logical cache structure and operation of web proxy cache that employs 2-LEVEL replacement algorithm. In the first step,

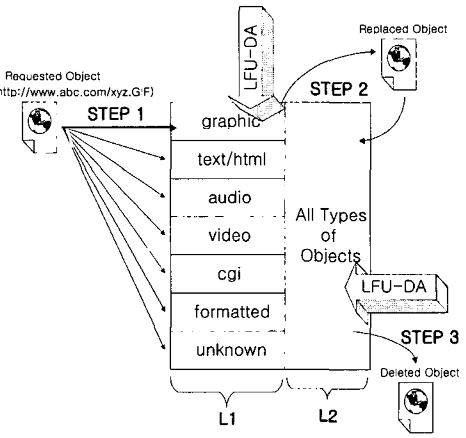


Fig. 4. Logical structure and operation flow of web proxy cache using 2-LEVEL replacement algorithm.

```
2-LEVEL Algorithm
Input: Requested object from a client;
Output: Retrieved object from a cache;
Determine the class of requested object;
if the new object is not in the corresponding section in L1 // L1: upper level cache
  then if the new object is not in L2 // L2: lower level cache
    then retrieve the requested object from the web server;
      if there is no enough room in the section of L1
        then while (no enough room for the new object in the section of L1)
          Select the object with the smallest key using LFU-DA;
          if there is no enough room in L2 for the object selected from L1
            then while (no enough room in L2 for the object from L1)
              Remove a object with the smallest key in L2 using LFU-DA;
          Transfer the selected object from L1 to L2;
Add the new object to the corresponding section;
Return requested objected to the client;
```

Fig. 5. Pseudo code describing 2-LEVEL algorithm

the newly entered object is stored into the corresponding partition of the L1 cache according to its file class. In the next step, when there is no enough room for the newly entered object, the replacement algorithm (LFU-DA) is applied to the corresponding section of the L1 cache in order to prepare a space for the new object. Then, the object removed from the L1 cache is stored into the L2 cache if there is an enough space in L2 cache. If the L2 cache doesnt have enough space to store the removed object, the LFU-DA algorithm is employed again to make a space.

If an object requested by a client host has been found in the L2 cache, the object is sent to the client and, at the same time, moved to the corresponding section of the L1 cache. The L2 cache is used as a shared area for the needs of an additional space when user requests are concentrated in a certain file type or when the spaces of the L1 cache is not adequately assigned for some classes of objects. In other words, the L2 cache prevents the high miss rate caused by conflicts of objects in a small space. The 2-LEVEL algorithm can make a maximal utilization of cache space. The procedure of the 2-LEVEL algorithm can be described by pseudo code as shown in Figure 5.

V. Simulations and Analyses

We use Saskachewan and Calgary trace data available in a public domain of Internet to improve the reliance on simulations instead of using our own trace data [1].

Saskachewan: It includes the valid 2,165,415 requests among a total of 2,408,625 accesses during 214 days from June 1. 1995 through December 31, 1995.

Calgary: It includes the valid 567,795 requests among a total of 726,739 accesses during 353 days from October 24. 1994 through October 11, 1995.

The size-based policies have better hit rates than frequency-based policies. And, among size-based policies, the GDSF algorithm has better performance than SIZE in hit rate. On the other hand, the frequency-based policies provide better byte-hit rates than size-based policies. The LFU-DA achieves higher byte-hit rate than any other frequency-based policies such as LFU. In the 2-LEVEL algorithm, LFU-DA or LRU can be used to choose a victim object in each section since both of them take into account object access times and are not size-based policies. However, we employ the LFU-DA for the 2-LEVEL algorithm to compare its performance with the TYPE algorithm, which already has used LRU policy.

We have used the Visual C++ to make the simulator and MS Excel to draw the following graphs. We have performed excessive simulations to obtain the dependable results about performance of web cache replacement policies. We have simulated the proposed TYPE, 2-LEVEL algorithms and existing algorithms that have best performance in each metric using the two trace data, i.e., Saskachewan and Calgary trace data. It has been shown that the GDSF algorithm showed the best performance in the hit rate, and the LFU-DA algorithm showed the best performance in the byte-hit rate. In the following figures, we will show the changes of hit rates, byte-hit rates, and the size of the lower level cache (L2) over the sequence of user request. We will also give in depth analyses about the simulation results.

We allocate appropriate area to each section according to the amount of transfers in bytes per second for responses of the corresponding class. In simulations, the initial partition size for class i is assigned according to the average of $transfer_{class\ i}$ values for the first 1000 requests. We allocate the initial size for each section as shown in Table 4 for Saskachewan and Calgary trace data. The sizes of L2 cache are fixed as 530 megabytes for Saskachewan data and as 330 megabytes for Calgary data, respectively.

Figure 6 shows the variation of hit rates along the sequence of user requests in the Saskachewan and Calgary web traces. From the two graphs, we know that the GDSF algorithm has the best performance for the hit rate just as we have expected. However, the LFU-DA algorithm provides the worst performance for it. The proposed TYPE algorithm and 2-LEVEL algorithm achieve much better performance in the hit rate than the LFU-DA policy. Among the algorithms, the 2-LEVEL algorithm provides better performance in terms of hit rate. In some situations, the 2-LEVEL algorithm generates higher hit rate than the GDSF. In general, the 2-LEVEL shows smoother curves than the other policies. This stems from the fact that a proper area is provided to each partition for the assigned class and the objects in the partition have similar sizes to achieve satisfactory high hit rate. Especially, the 2-LEVEL algorithm provides a back-up space that can be used for any class. And this section provides a kind of shelter area for those objects of a class that become popular

Table 4. Initial size of each partition for simulations of TYPE and 2-LEVEL algorithms on Saskachewan and Calgary trace data.

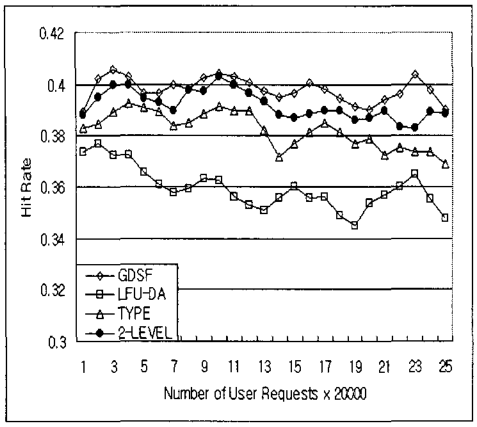
Saskachewan data

	Html	Images	Sound	Video	Dynamic	Formatted	Other
TYPE	892M	644M	27M	46M	77M	2M	72M
2-LEVEL	625M	450M	18M	32M	54M	1M	50M

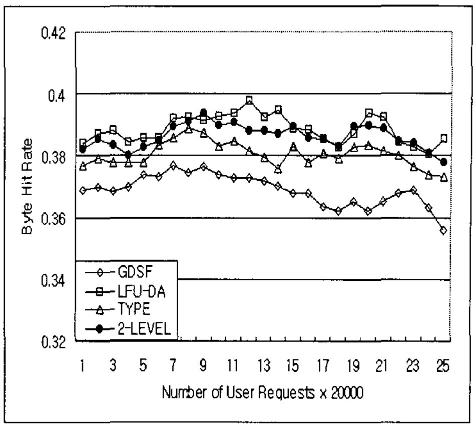
Calgary data

	Html	Images	Sound	Video	Dynamic	Formatted	Other
TYPE	144M	546M	14M	124M	1M	237M	24M
2-LEVEL	100M	380M	10M	86M	1M	165M	18M

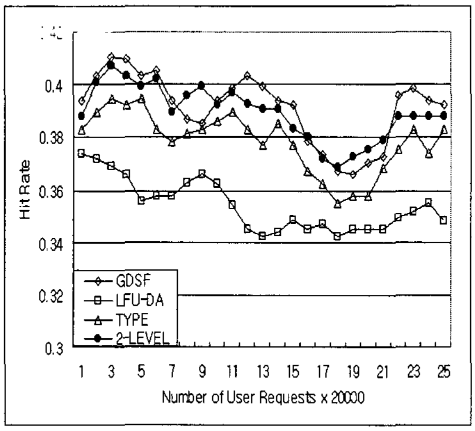
(M: Megabytes)



(a) Hit rate for Saskachewan

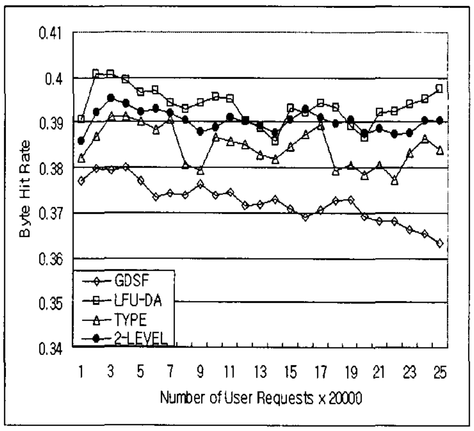


(a) Byte-hit rate for Saskachewan



(b) Hit rate for Calgary

Fig 6. Hit rates for two workloads Saskachewan the Calgary over the sequence of requests.

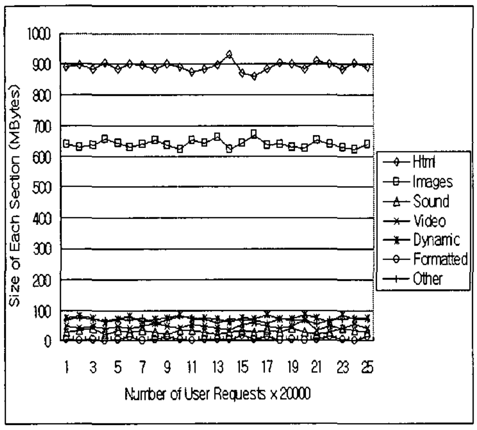


(b) Byte-hit rate for Calgary

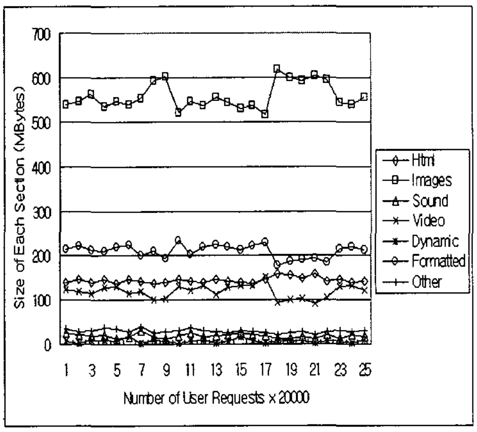
Fig. 7. Byte-hit rates for two workloads Saskachewan and Calgary over the sequence of requests.

and suddenly referenced by many client hosts.

Figure 7 shows the variation of byte-hit rates over the sequence of user requests using the same two traces. The LFU-DA algorithm shows the best byte-hit rates as we have expected. And the GDSF algorithm gives the worst performance in terms of the byte-hit rate among four replacement policies. The TYPE algorithm provides byte-hit rates that are between those of LFU-DA and GDSF. However, the byte-hit rates of the 2-LEVEL algorithm are almost equivalent to those of LFU-DA algorithm. Especially, when user requests are concentrated on objects whose average sizes are large during relatively short period of time, the 2-LEVEL algorithm achieves outstanding performance. Even for small-sized objects, it provides an excellent performance although the improvement is not very remarkable. The high byte-hit rates of the 2-LEVEL algorithm can be explained by the following two reasons. First, the cache pollution problem in each partition is resolved by employing the aging factor in the replacement algorithm. Second, a shelter area is provided for some large objects that are removed from their original sections because of the conflict with each other. And those large objects saved in the shelter area provide very high byte-hit rates.

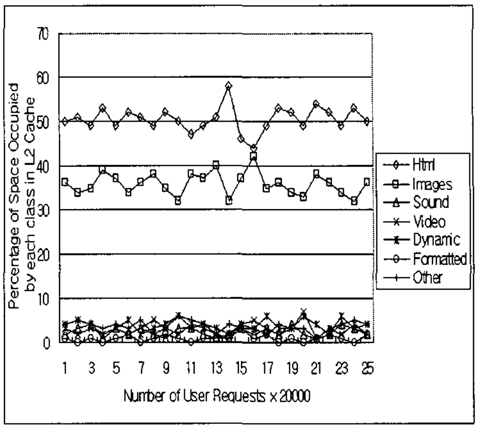


(a) The size of each section for Saskachewan

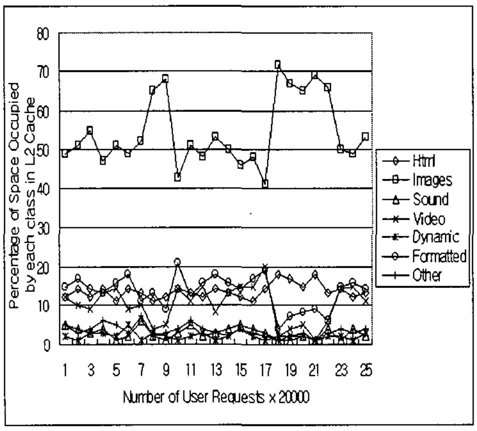


(b) The size of each section for Calgary

Fig. 8. Variations of the size of each section for Saskachewan and Calgary trace data.



(a) The size of L2 cache for Saskachewan



(b) The size of L2 cache for Calgary

Fig. 9. Variations of the size of L2 cache for Saskachewan and Calgary trace data.

Figure 8 shows the variations of the size for each section on Saskachewan and Calgary trace data. From two graphs of the figure, we can find the fact that there is very little variation from the initial allocated size for each class along the sequence of requests. This fact shows that the implementation of the proposed 2-LEVEL algorithm can be very simple. Even if we fix the size of each section using statistical data for some time period, a competitive performance can be obtained from the proposed algorithm. Figure 9 depicts the dynamical distribution of each area occupied by files in L2 cache. In Saskachewan trace data, HTML and image files take the most of the L2 space. In Calgary trace, HTML files occupy relative small area, but image files take up most space.

Ⅵ. Conclusions

In a cache, the performance of replacement algorithm is often measured by the hit rate. In proxy server, the hit rate is defined as the number of objects found in the cache as a percent of total objects requested since HTTP is the transfer protocol in object unit. However, the web cache is different from traditional file cache or virtual memory system. The web cache must deal with objects of variable sizes ranging from several hundreds of bytes to several tens of megabytes in WWW. Thus, the byte-hit rate, which is a percent of the total number of bytes that the proxy cache directly serves for all requests, is also used for the proper evaluation of cache performance. Since it's impossible to transfer a part of object in HTTP, it is inevitable to employ a cache management policy based on variable-sized object in proxy server.

In this paper, we propose two novel replacement algorithms to achieve both high hit rate and byte-hit rate with great satisfaction. In the proposed TYPE algorithm, sections of reasonable space are assigned exclusively for each class of objects to guarantee high hit rates. Since objects in the same class are similar in size and assigned to the same section in proposed TYPE algorithm, LRU replacement algorithm can also provides high byte-hit rate for a given amount of cache space. In the proposed 2-LEVEL algorithm, an extra L2 cache is provided as the shared area to accommodate objects removed from the L1 cache. The L2 cache is collectively used for all types of files as a shelter area. In the 2-LEVEL algorithm, the hit rate is obtained from partitioning of cache and the byte-hit rate is improved by LFU-DA replacement policy.

From the simulations, we know that the GDSF algorithm has the best performance for the hit rate, but the LFU-DA algorithm provides the worst performance. Whereas, the LFU-DA algorithm shows the best byte-hit rates, but the GDSF algorithm gives the worst byte-hit rates among four replacement policies. The proposed TYPE and 2-LEVEL algorithms provide byte-hit rates that are between those of LFU-DA and GDF. However, the hit rates and byte-hit rates of the 2-LEVEL algorithm are almost equivalent to those of GDSF and LFU-DA, respectively. In summary, the 2-LEVEL algorithm achieves outstanding performance in most cases.

References

[1] M. F. Arlitt and C. L. Williamson, "Web server workload characterization: The search for invariants," *Proc. of the ACM SIGMETRICS*, pp.126-137, Philadelphia, PA, ACM, Apr. 1996.

[2] M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich and T. Jin, "Evaluating content management techniques for web proxy caches," *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, no. 4, pp. 3-11, March 2000.

[3] T. Berners-Lee, R. Fielding, and H. Frystyk, Hypertext transfer protocol - HTTP/1.0, RFC 1945, May 1996.

[4] S. Williams, M. Abrams, C. R. Standridge, G. Abdulla, and E.A. Fox, "Removal policies in network caches for world wide web documents," *ACM SIGCOMM 96*, pp.293-305, Aug. 1996.

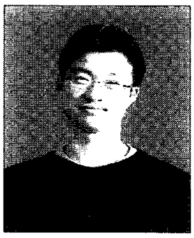
[5] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," *Proceedings of the 1997 USENIX Symposium*, pp.193-206, Dec. 1997.

[6] A. Silberschartz and P. B. Galvin, Operating system concepts, The 5th edition, Addison-Wesley Publishing Company, Nov. 1998.

이 수 행(Soo-haeng Lee) 정회원

2000년 명지대학교 정보통신공학과(학사).
2002년 인하대학교 정보통신공학과(석사).
<주관심분야> 컴퓨터 네트워크, 컴퓨터 구조, 네트워크 관리시스템

두 현 재(Hyeon-jae Doo) 정회원



1995년 명지대학교 제어계측공학과(학사)
2001년 인하대학교 전자공학과(석사)
2001년 ~ 현재 LG전자 핵심망 연구소 핵심망 S/W 2 그룹 근무중
<주관심분야> 컴퓨터 구조, 컴퓨터 네트워크, SS7 통신 프로토콜 , Web Cache Algorithms

최 상 방(Sang-bang Choi) 정회원

1981년 한양대학교 전자공학과 (학사) University of Washington(석사)
1990년 University of Washington(박사)
1981년 ~ 1986년 LG 정보통신 (주) 근무
1991년 ~ 현재 인하대학교 전자공학과 교수
<주관심분야> 컴퓨터 구조, 컴퓨터 네트워크, 병렬 및 분산처리 시스템, Fault-tolerant computing