



# Proxy Cache Replacement Algorithms: A History-Based Approach

ATHENA VAKALI

*Department of Informatics, Aristotle University, Thessaloniki 54006, Greece*

avakali@csd.auth.gr

## *Abstract*

Accessing and circulation of Web objects has been facilitated by the design and implementation of effective caching schemes. Web caching has been integrated in prototype and commercial Web-based information systems in order to reduce the overall bandwidth and increase system's fault tolerance. This paper presents an overview of a series of Web cache replacement algorithms based on the idea of preserving a history record for cached Web objects. The number of references to Web objects over a certain time period is a critical parameter for the cache content replacement. The proposed algorithms are simulated and experimented under a real workload of Web cache traces provided by a major (*Squid*) proxy cache server installation. Cache and bytes hit rates are given with respect to different cache sizes and a varying number of request workload sets and it is shown that the proposed cache replacement algorithms improve both cache and byte hit rates.

**Keywords:** Web cache replacement, Web-based information systems, Web caching and proxies, cache replacement policies

## 1. Introduction

Web caching has introduced an effective solution to the problems of traffic congestion, bandwidth insufficiency and (distributed objects) accessing over the Web. Cache efficiency depends on the chosen cache update scheme, as well as on the algorithmic approach used to maintain the cache content reliability and consistency. Several approaches have been proposed for effective cache management and the problem of maintaining a consistent cache content has gained a lot of attention recently, due to the fact that many Web caches often fail to maintain a consistent cache. Several techniques and frameworks have been proposed towards a more reliable and consistent cache infrastructure [7,14].

Most Web servers are reinforced with proxy cache servers in order to keep the objects closer to end users by adding specific cache consistency mechanisms and cache hierarchies. *Cache consistency* mechanisms have been included in almost every proxy cache server (e.g., [17]) and their improvement became a major research issue. In [11] a survey of contemporary cache consistency mechanisms in the Internet is presented and the introduction of trace-driven simulation shows that a weak cache consistency protocol reduces network bandwidth and server load more than prior estimates of an objects life cycle or invalidation protocols. Furthermore, prefetching and caching are techniques proposed to reduce latency in the Web. Earlier research efforts have specified several bounds on

the performance improvement rates derived by these techniques under specific workloads (e.g., [13]).

Web cache replacement policies are an actual research topic of high interest, in relation to current proxy cache implementations. In [9] two innovative Web cache replacement policies are implemented in the Squid cache server framework and they show improvement over standard mechanisms, whereas in [22] another set of LRU-based algorithms are proposed for effective cache replacement. In [2] the importance of various workload characteristics for the Web proxy caches replacement is analyzed and trace-driven simulation is used to evaluate the replacement effectiveness. In [12] a Web-based dynamic data caching model is introduced and this model's design and performance are analyzed. A number of Web replacement policies are discussed in [4], and compared on the basis of trace-driven simulations. A Web-based evolutionary model has been presented in [20] where cache content is updated by evolving over a number of successive cache objects populations and it is shown by trace-driven simulation that cache content is improved. A Genetic algorithm model is presented in [21] for Web objects replication and caching. Cache replacement is performed by the evolution of Web objects cache population accompanied by replication policies employed to the most recently accessed objects. Furthermore, performance of Web caching is studied in [1] where a new generalized LRU is presented as an extension to the typical SLRU algorithm. Hit ratios and robustness of the proposed replacement algorithm is compared with other Web replacement policies using both event and trace-driven simulations. A more recent approach proposes randomized algorithms for approximating any existing Web cache replacement scheme, in an effort to tackle with the problem of document replacement in Web caches [18].

This paper presents a new approach to Web cache replacement by proposing a number of cache replacement algorithms extended with a so-called "history" parameter which serves as the basic criterion for a effective cache replacement. The key contributions and main issues of the paper are as follows:

- The proposed algorithms are based on earlier widely adopted cache replacement algorithms, namely, the LRU, the SLRU, the MFU and the LFU.
- A new version of each of these algorithms is defined where a "history" of Web objects requests is preserved based on the idea of page replacement in database disk buffering [16]. The main idea is to keep a record for a number of past references to Web objects, i.e., a history of the times of the last  $h$  requests is evaluated for each cached Web object in order to support a more detailed cache replacement.
- A new set of cache replacement algorithms is defined. These algorithms are named HLRU, HSLRU, HMFU, HLFU in correspondence with the typical LRU, SLRU, MFU, LFU algorithms. An initial version of the HLRU algorithm was introduced by the author in [22].
- The proposed algorithms are evaluated and experimented under Squid proxy cache traces and cache log files. Therefore, the importance of preserving a history record for cached objects is highlighted and commented.

The remainder of the paper is organized as follows. The next section introduces and defines the cache replacement problem. Section 3 presents the proposed history-based

algorithms whereas Web proxies performance and workload characteristics are presented in Section 4. Section 5 has the experimentation details and figures depict the results derived from the trace-driven simulation. Section 6 summarizes the main conclusions and the future research topics are identified.

## 2. The cache replacement problem

Web proxy implementations are based on a pre-specified cache area of limited space, for storage of a bounded number of Web objects. Once the content in the cache area reaches the cache predetermined limits, a cache replacement policy should be employed to update the cache content with more recent requested Web objects. Each cached Web object is characterized by its so-called *staleness* which is related with the need to contact the original server to validate the existence of the cache copy. Web object's staleness is related to the cache server's lack of awareness about the original object's changes. Each proxy cache server implementation must be reinforced with specific staleness confrontation.

The cache replacement problem is defined by introducing a set of parameters that will monitor the Web cache content replacement process. Web cache content can be modeled by an informative hash table of a number of rows, where each row is associated with a particular cached object. Therefore, the number of rows is bounded by the number of cached objects. Each object is identified by its corresponding stored object filename, along with a number of related attributes. The attributes are chosen such that cache replacement could be supported and employed. The most important factors for the cache replacement refer to the object's staleness status, its frequency of access and its retrieval rate.

The most important parameters in relation to attributes of each cached object are summarized in Table 1. Definitions for each of these factors are given next:

*Definition 1.* The *popularity* of a Web object  $i$  is defined by

$$pop_i = \frac{hits_i}{hits_{tot}},$$

Table 1. The most useful attributes of each cached object  $i$

Parameter	Description
$C$	total available cache area size
$N$	number of objects in cache
$s_i$	server on which object resides
$b_i$	object's size in kBytes
$t_i$	time the object was logged
$c_i$	time the object was cached
$l_i$	time of object's last modification
$af_i$	number of cache accesses since the last time object $i$ was accessed
$key_i$	objects original copy identification (e.g., its URL address)
$pop_i$	popularity of a cached object $i$
$df_i$	dynamic frequency of cached object $i$

where  $hits_i$  refer to the number of hits for the cached object  $i$  out of the total number of hits  $hits_{tot}$  on the considered cache area. The popularity values are a percentage metric, since it is always true that  $0 \leq pop_i \leq 1$ , for all  $i$ .

*Definition 2.* The cached object's *staleness ratio* is defined by

$$StRatio_i = \frac{c_i - l_i}{now - c_i},$$

where the numerator corresponds to the time interval between the time of object being cached and the time of the object's last modification and the denominator is the cache "age" of the object, i.e., it determines the time that the object has remained in cache. It is always true that  $StRatio_i \geq 0$  since  $c_i - l_i \geq 0$  and  $now - c_i > 0$  ( $now$  is the current time). Since  $c_i - l_i$  is a fixed value and  $now - c_i$  increases as the time goes on, the lower the value of  $StRatio_i$  the more stale the object  $i$  is (since  $is$  is the main indication that this object has remained in cache for longer period).

*Definition 3.* The *dynamic frequency* of cached object  $i$  is defined by

$$df_i = \frac{pop_i}{af_i},$$

where  $af_i$  is the metric to identify the number of accesses to other objects since object  $i$  was last referenced (Table 1). We assume that  $af_i \neq 0$  since we consider objects  $i$  which have been already cached so they reside in cache after at least another object's reference has occurred. Therefore, it is true that the higher the values of  $df_i$ , the most popular and recently the object  $i$  was accessed.

A Web cache server considers has to support mechanisms which will determine whether an object could be cached or not. In case that there is not enough cache space, there is a need to remove one or more objects from the cache in order to free sufficient space. The cache replacement process must guarantee enough space for the incoming objects. Therefore, there are two actions related to the replacement process, either the object will remain *stored* in cache or it will be *purged* from cache. A function is needed to identify the action that should be taken for each cached object.

*Definition 4.* The cached object's *action function* is defined by

$$act_i = \begin{cases} 0 & \text{if object } i \text{ will be purged from cache,} \\ 1 & \text{otherwise.} \end{cases}$$

Here, we formulate the cache replacement problem in terms of mathematical programming modeling under our definition of dynamic frequency. A similar approach has been introduced in [1], where cache replacement was defined as an optimization problem in the set of NP-hard algorithms. Here, we consider both the staleness ratio and the dynamic frequency parameters in the problem statement in order to well define the cache replacement problem.

*Problem Statement.* Suppose that  $N$  is the number of objects in cache and  $C$  is the total capacity of the cache area. The cache replacement problem is to:

$$\begin{aligned} \text{Maximize} \quad & \sum_{i=1}^N act_i \cdot StRatio_i \cdot df_i \\ \text{subject to} \quad & \sum_{i=1}^N act_i \cdot b_i \leq C. \end{aligned}$$

In the optimization formula the  $StRatio_i$  and the  $df_i$  are used as the “weight” factors characterizing each cached object, since they involve both the object’s popularity and updating status. At all times, the basic goal of the proposed cache replacement problem is to maintain in cache the most non-stale, frequently accessed Web objects.

### 3. The history-based cache replacement

#### 3.1. Cache replacement policies

A typical cache replacement approach involves updating the cache content under a certain criterion or over a considered time period. Figure 1 presents a random cache replacement policy as proposed here in order to serve as an indicative cache replacement policy for comparisons. Furthermore, the LRU is presented in Figure 2 since the LRU algorithm is the most popular cache replacement algorithm in most current proxy servers. As presented in this figure, the LRU (Least Recently Used) is based on the Squid proxy cache replacement and also the case of emergency purging is considered. In case of an emergency purge the cached objects are sorted and the most stale cached objects are purged.

*Definition 5* (Temporal locality rule). The Web objects which were not referenced in the recent past, are not expected to be referenced again in the near future.

LRU cache replacement is based on the temporal locality rule (Definition 5) and the least recently requested objects are purged from cache. The LRU cache replacement occurs when either there is a need for cache disk space or periodically in order to keep the cache area in an appropriate usage level.

*Definition 6* (Threshold parameter). A value identified as *threshold* is needed for estimating the expected time needed to fill or completely replace the cache content. This threshold is dynamically evaluated based on current cache size and on the low and high watermarks. When current cache size is closer to low watermark the threshold gets a higher value, otherwise when current cache size is closer to high watermark the threshold value is smaller.

---

```

number of objects that will be erased = RandomNumber ( size of bucket );

while ( ( k < number of objects that will be erased ) && ( not an emergency purge ) )
{
    k++;
    position of the object that will be erased = RandomNumber ( size of bucket );

    if ( position ( left + position of the object that will be erased ) is not empty )
    {
        purge the object;
        time object stayed in cache = CurrentTime - hashTable.timeOfFirstAccess;
        current cache swap - = object's size;
    }
}

if ( an emergency purge )
{
    for ( int t = 0; t <= 8; t ++ )
    {
        if ( position ( left + position of the object that will be erased ) is not empty )
        {
            purge the object;
            time object stayed in cache = CurrentTime - hashTable.timeOfFirstAccess;
            current cache swap - = object's size;
        }
    }
}

```

---

Figure 1. The random cache replacement algorithm.

### 3.2. The history-based approach

Here we introduce a scheme to support a “history” of the number of references to a specific Web object.

**Definition 7** (History function). Suppose that  $r_1, r_2, \dots, r_n$  are the requests for cached Web objects as logged at the time units  $t_1, t_2, \dots, t_n$ , respectively. A history function for a specific cached object  $x$  is defined as follows:

$$\text{hist}(x, h) = \begin{cases} t_i & \text{if there are exactly } h - 1 \text{ references between times } t_i \text{ and } t_n, \\ 0 & \text{otherwise.} \end{cases}$$

The above function  $\text{hist}(x, h)$  is a time metric and defines the time of the past  $h$ th reference to a specific cached object  $x$  (based on the idea of [16]). Furthermore, the time  $t_i$  identifies the first of the last  $h$  references to  $x$ .

**The HLRU algorithm.** One of the disadvantages of the LRU is that it only considers the time of the last reference and it has no indication of the number of references for a certain Web object. Therefore, the proposed HLRU (History LRU) algorithm will replace

---

```

qsort (hashTable)

for ( i = left bucket boundary; i<= right bucket; i++ )
{   if ( not an emergency purge )
    {   if ( position i is not empty )
        if ( age of object i > LRU threshold )
        {   purge i object from the cache;
            time the object stayed in the cache =
                CurrentTime - hashTable.timeOfFirstAccess;
            current cache swap - = object's size;
        }
    }
}
else // the case of an emergency purge
{   if ( i <= left bucket boundary + 8 )
    {   if ( position i is not empty )
        {   purge the object;
            time the object stayed in the cache =
                CurrentTime - hashTable.timeOfFirstAccess;
            current cache swap - = object's size;
        }
    }
}
}

```

---

Figure 2. The LRU cache replacement algorithm.

Table 2. The main LRU and HLRU data structure

LRU	HLRU
struct <i>HashTable</i>	struct <i>HashTable</i>
{ long <i>LRU_age</i>	{ int <i>OldTimeOfAccess</i>
long <i>positionInFile</i>	long <i>positionInFile</i>
Boolean <i>empty</i>	Boolean <i>empty</i>
long <i>timeOfFirstAccess</i>	long <i>timeOfFirstAccess</i>
} <i>hashTable</i> [ ]	} <i>hashTable</i> [ ]

the cached objects with the maximum *hist* value. In case there are many cached objects with *hist* = 0, the typical LRU is considered to decide on which object will be purged from cache. The same idea of the threshold value (to decide when the cache replacement will occur) still holds. In Table 2 the main structure of the cache hash table is presented and the difference between the conventional LRU and the proposed HLRU is highlighted by the presented data structures. For the typical LRU, each cached object is assigned an *LRU\_age* to indicate the time since its last reference. Under HLRU, the old time of

---

```

for (i=left boundary; i<=right boundary; i++)
{
    if (hashTable[i]. OldtimeOfAccess == 0)
    {
        ++counterOfObjectswithOneAccess;
        if (counterOfObjectswithOneAccess > 1)
            break;
    }
}
if (there is >1 object with only 1 access)
    for (i=left boundary; i<=right boundary; i++)
        if (the object i had more than one accesses)
            age[i] = CurrentTime - hashTable[i]. OldtimeOfAccess;
        else
            age[i] = CurrentTime - hashTable[i]. NewtimeOfAccess;
else
    for (i=left boundary; i<=right boundary; i++)
        if (the object i had more than one accesses)
            age[i] = CurrentTime - hashTable[i]. OldtimeOfAccess;
        else
            hashTable[i].empty = true;

qsort(hashTable);

for (i=left boundary; i<=right boundary; i++)
{
    if (not an emergency purge)
    {
        if (position is full)
            if (the age of object i > LRU threshold)
            {
                object is purged;
                time the object stayed in cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap - = object's size;
            }
    }
    else // the case of an emergency purge
    {
        if ( i <= left boundary of the bucket + 8 )
            if ( position i is not empty )
            {
                purge the object;
                time the object stayed in cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap - = object's size;
            }
    }
}

```

---

Figure 3. The HLRU cache replacement algorithm.

access as identified by the *hist* value is the criterion to characterize an object's remaining or purging from the cache area. Variable *positionInFile* declares the position the specific object has in the file, whereas the Boolean type variable *empty* indicate whether the specific cache location is empty or not. Finally, the variable *timeOfFirstAccess* is used for the

Table 3. The main HSLRU and MFU/LFU data structure

HSLRU	HMFU/HLFU
<pre> struct HashTable { long timeOfFirstAccess   long positionInFile   Boolean empty   long frequency } hashTable[] </pre>	<pre> struct HashTable { int frequencyOfAccess   long positionInFile   Boolean empty   long timeOfFirstAccess } hashTable[] </pre>

time the specific object was cached. Figure 2 presents the implemented LRU algorithm in pseudocode.

The HLRU data structure is quite similar, there are two different times kept for each cached object. *OldTimeOfAccess* is the time the cached object was first referenced whereas *NewTimeOfAccess* is the time of the last reference to the cached object. Similarly, Figure 3 presents the implemented HLRU algorithm in a pseudocode format, for the case of two ( $h = 2$ ) past references. The HLRU for  $hist(x, h)$ , where  $h > 2$ , i.e., for preserving a record of a more detailed history supports a linked list for each cached object in order to keep track of the times of past references.

**The HSLRU algorithm.** The conventional SLRU (Segmented LRU) algorithm has been used in order to overcome earlier LRU problems such as not considering the number of hits for the cached objects and so on. Under SLRU cache is divided into two segments with sorted objects by the most recent to most old referenced objects. One of the two segments (so-called protected segment) maintains the objects that have been referenced at least twice and there is a boundary pointer for pointing at the most frequently referenced object in this section. The proposed history based HSLRU algorithm will replace the cached objects of the protected segment based on the *hist* values. These *hist* values determine the objects accesses as checked in order to determine the object's purging action (Figure 4). Similarly, the threshold value (to decide when the cache replacement will occur) is still used. In Table 3 the main structure of the cache hash table is presented for the HSLRU algorithm. Figure 4 presents the proposed HSLRU algorithm (for  $h = 2$ ) whereas Figure 5 has the algorithm used for employing the sorting in the HSLRU in order to determine the HSLRU cache boundary pointer.

**The HMFU algorithm.** The conventional MFU (Most Frequently Used) algorithm performs a cache replacement by purging the most frequently requested objects. The basic idea of MFU is that cache objects are sorted based on their *frequencyOfAccess* and the objects with a value greater or equal than the MFU threshold are purged from the cache area. Here, we introduce an extension of the typical MFU which involves the history function *hist* in the cache replacement process. We use a respective MFU threshold which is a linear function on the amount of cache currently in use. The proposed history-based, so called the HMFU algorithm, will perform cache replacement by assessing the cached objects based on their *hist* value as compared to the defined MFU threshold. In Table 3 the

---

```

for (i=left boundary; i<=right boundary; i++)
{
    if (hashTable[i]. OldtimeOfAccess == 0)
    {
        ++counterOfObjectswithOneAccess;
        if (counterOfObjectswithOneAccess > 1) break;
    }
}
if (there is more than one object with only one access)
{
    for (i=left boundary; i<=right boundary; i++)
    {
        if (the object i had more than one accesses)
            age[i] = CurrentTime - hashTable[i]. OldtimeOfAccess;
        else
            age[i] = CurrentTime - hashTable[i]. NewtimeOfAccess;
    }
}
else
{
    for (i=left boundary; i<=right boundary; i++)
    {
        if (the object i had more than one accesses)
            age[i] = CurrentTime - hashTable[i]. OldtimeOfAccess;
        else
            hashTable[i].empty = true;
    }
}

boundary_pointer=qsorSLRU(age,hashTable.positionInFile,hashTable.empty,hashTable.frequency, hashTable.timeOfFirstAccess);

for (i=boundary pointer; i<=right boundary of the bucket; i++)
{
    if (not an emergency purge)
    {
        if (position is full)
        {
            if (age[i] > LRU threshold)
            {
                the object is purged;
                time the object stayed in the cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap - = object's size;
            }
        }
    }
    else // the case of an emergency purge
    {
        if (k <= left boundary of the bucket + 8 )
        {
            if (position k is not empty )
            {
                purge the object;
                time the object stayed in the cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap - = object's size;
            }
        }
    }
}

```

---

Figure 4. The HSLRU cache replacement algorithm.

---

```

int qsortSLRU (age[], position[], empty[], frequency[], timeOfFirstAccess[],
int leftboundaryOfbucket, int rightboundaryOfbucket, int boundaryPointer)

{
    if (it first application of SLRU to the bucket)
    {
        for (i=leftboundaryOfbucket; i<=rightboundaryOfbucket; i++)
        {
            if (object i had more than one accesses)
            {
                for (k=leftboundaryOfbucket; k<i; k++)
                if (object k was accessed for only one time)
                    swap (object k with object i);
                    break;
            }
        }
    }
    else
    {
        for (l = boundaryPointer; l <= rightboundaryOfbucket; l++)
        {
            if (object l had more than one accesses)
            {
                index=l; //keeps the index of the last new hit
                numofnewHits++;
                for (k= boundaryPointer; k<l ; k++)
                if (object k was accessed only one time)
                    swap (object k with object i);
                    break;
            }
        }

        if (Boundary!=0)&&(there were new hits at the probationary segment)
        {
            qsort(age,position,empty,frequency,timeOfFirstAccess,leftbound
            daryOfbucket,
                boundaryPointer+numofnewHits-1);
            qsort(age,position,empty,frequency,timeOfFirstAccess,boundaryPoin
            ter+numofnewHits,
                rightboundaryOfbucket);
        }
    }
    if (number of elements of the protected part of the array > 7)
    {
        for(t=7;t<=numofnewHits+boundary pointer-1,t++)
            frequency[t] = 0;
    }
    for (i=leftboundaryOfbucket; i<=rightboundaryOfbucket; i++)
    if (object k was accessed only one time)
        break;
        boundary pointer++;

    return boundary pointer;
}

```

---

Figure 5. The qsort algorithm for the HSLRU.

---

```

input h // history parameter

qsort(hashTable.frequencyOfAccess,hashTable.positionInFile,hashTable.empty, hashTable.timeOfFirstAccess);

for (k = left boundary of the bucket; k <= right boundary of bucket; k++)
{
    if (not an emergency purge)
    {
        if (position k is not empty)
        {
            if (hist(h,k) >= MFU threshold)
            {
                purge the object from the cache;
                time the object stayed in the cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap - = object's size;
            }
        }
    }
    else // the case of an emergency purge
    {
        if (k <= left boundary of the bucket + 8)
        {
            if (position k is not empty)
            {
                purge the object;
                time the object stayed in the cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap - = object's size;
            }
        }
    }
}

```

---

Figure 6. The HMFU cache replacement algorithm.

main structure of the cache hash table is presented for the HMFU algorithm whereas the proposed HMFU algorithm is presented in Figure 6.

**The HLFU algorithm.** The typical LFU (Least Frequently Used) algorithm performs a cache replacement by replacing the least frequently requested objects. Here, we introduce an extension of the typical LFU which considers the history function in the cache replacement process. We use a respective LFU threshold which is a linear function on the amount of cache currently in use. The proposed history based HLFU algorithm will replace the cached objects based on the *hist* value as compared to the defined LFU threshold. HLFU has the same structure of the cache hash table with the HMFU algorithm (Table 3). Similarly, the proposed HLFU algorithm is presented in Figure 7.

#### 4. Web proxy cache servers

Caching was initially introduced to provide an intermediate storage space between the main memory and the processor, relying on locality of reference by assuming that the most recently accessed data has the highest potential of being accessed again soon. Caching was extended to Web servers in order to improve client latency, network traffic and server load.

```

input h // history parameter

qsort(hashTable.frequencyOfAccess,hashTable.positionInFile,hashTable.empty, hashTable.timeOfFirstAccess);

for (k = left boundary of the bucket; k <= right boundary of bucket; k++)
{
    if (not an emergency purge)
    {
        if (position k is not empty)
        {
            if (hist(k,h) <= LFU threshold)
            {
                purge the object from the cache;
                time the object stayed in the cache =
                    CurrentTime - hashTable.timeOfFirstAccess;
                current cache swap -= object's size;
            }
        }
    }
}
else // the case of an emergency purge
{
    if (k <= left boundary of the bucket + 8)
    {
        if (position k is not empty)
        {
            purge the object;
            time the object stayed in the cache =
                CurrentTime - hashTable.timeOfFirstAccess;
            current cache swap -= object's size;
        }
    }
}
}

```

Figure 7. The HLFU cache replacement algorithm.

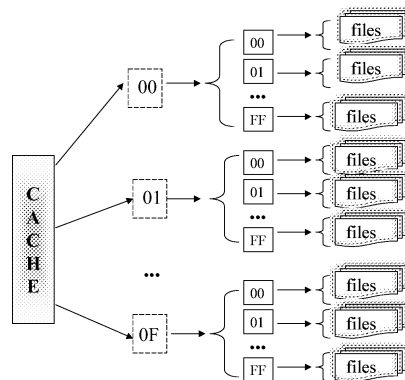


Figure 8. Structure of the Squid proxy cache area.

4.1. Web proxies—a brief overview

A Web cache is an application residing between Web servers and clients such that it watches requests for information objects identified as html pages, images, documents and

files. Web cache servers reply to the users request by sending the requested Web object and by (at the same time) saving a copy for the cache itself. If there is another request for the same object, cache will use the copy it has, instead of asking the original server for it again [15]. As pointed out in Section 1, the two main Web caches advantages are the reduce in both latency (request is satisfied by the cache which is closer to the client) and traffic (each object is retrieved from the server once, thus reducing the bandwidth used by a client).

Nowadays a variety of cache servers are available for the World-Wide Web caching, most of them freely-distributed on the Internet. Most of the recent Web servers application include caching modules (for example, Apache, Spinner, Jigsaw, Purveyor). A brief description of the three most wide-spread proxy cache servers follows:

- *CERN proxy server* has been widely adopted since there was a large infrastructure of CERN Web servers already installed. A heuristic known as *time-to-live* (TTL), was used to manage object's staleness. TTL is implemented by using the last date modification header included in every reply from a Web server. A TTL timing frame based on that date, accompanies each document in cache [11,23].
- *Netscape Proxy Server* has been available commercially since 1995 and checks object's staleness by supporting TTL frame based on object's age when it is cached. This server also supports preemptive fetch groups of linked Web pages according to a schedule and has a variety of filtering options for use as a firewall proxy.
- *Harvest cache* software was developed with the aim of making effective use of the information available on the Internet, by sharing the load of information gathering and publishing between many servers. Harvest produced the ICP protocol for co-operation between individual caches. Newest Harvest developments are available commercially whereas a team from the N.L.A.N.R. (National Laboratory for Advanced Networking Research) has continued to provide a free version under the name Squid [19]. Squid has evolved by additional features for objects refreshment and purging, memory usage and hierarchical caching. Harvest and Squid have been adopted widely by many institutions and research organizations as a new proposal for efficient caching.

The Squid Proxy Cache is further discussed since the present paper develops a simulation environment based on the Squid cache model and experiments are made by the use of Squid trace log files. Squid caching software has gained a lot of attention lately, since it is used on an experimental network of seven major co-operating servers across U.S.A., under a project framework by NLANR [10]. These servers support links to collaborating cache projects in other countries. Aristotle University has installed Squid proxy cache for main and sibling caches and supports a Squid mirror site. The present paper uses data from this cache installation for experimentation.

#### 4.2. *The Squid proxy cache*

Figure 8 represents the organization of Squid cache hierarchy storage-wise, consisting of a two-level directory structure. Assuming approximately 256 objects per directory there is

a potential of a total of 1,048,576 ( $= 16 \times 256 \times 256$ ) cached objects. Squid uses a lot of memory for performance reasons since an amount of metadata for each cached object is kept in memory. Squid switched from the TTL base expiration model to a Refresh-Rate model. Objects are no longer purged from the cache when they expire. Instead of assigning TTLs when the object enters the cache, now a check of freshness requirements is performed when objects are requested. The refresh parameters are identified as *min\_age*, *Percent* and *max\_age*. *Age* is how much the object has aged since it was retrieved whereas *lm\_factor* is the ratio of age over the how old was the object when it was retrieved. *expires* is an optional field used to mark an object's expiration date. *Client\_max\_age* is the (optional) maximum age the client will accept as taken from the http cache-control request header. The following algorithm is used by Squid to determine whether an object is stale or fresh:

```

if Age > Client_max_age then
  Return "STALE"
else if Age <= min_age then
  Return "FRESH"
else if (expires) then // expires field exists
  if (expires <= NOW) then Return "STALE"
  else Return "FRESH"
else if Age > max_age then
  Return "STALE"
else if lm_factor < Percent then
  Return "FRESH"
else Return "STALE"

```

Squid keeps size of the disk cache relatively smooth since objects are removed at the same rate they are added and object purging is performed by the implementation of a Least-Recently-Used (LRU) replacement algorithm. Objects with large LRU age values are forced to be removed prior objects with small LRU ages. Squid cache storage is implemented as a hash table with some number of hash “buckets” and store buckets are randomized so that same buckets are not scanned at the same time of the day [19]. For example, in Squid, the LRU is used along with certain parameters such as a *low watermark* and a *high watermark* to control the usage of the cache. Once the cache disk usage is closer to the low watermark (usually considered to be 90%) fewer cached Web objects are purged from cache, whereas when disk usage is closer to the high watermark (usually considered to be 95%) the cache replacement is more severe, i.e., more cached Web objects are purged from cache. There are several factors as of which objects should be purged from cache.

The performance metrics used in the presented approach focus on the cached objects cache-hit ratio and byte-hit ratio:

- *Cache hit ratio* represents the percentage of all requests being serviced by a cache copy of the requested object, instead of contacting the original object's server.
- *Byte hit ratio* represents the percentage of all data transferred from cache, i.e., corresponds to ratio of the size of objects retrieved from the cache server. Byte hit ratio provides an indication of the network bandwidth.

The above metrics are considered to be the most typical ones in order to capture and analyze the cache replacement policies (e.g., [1,2,4]).

Furthermore, the performance of the proposed cache replacement algorithms is studied by estimating the strength of the cache content. This strength is evaluated by the consideration of the cached objects retrieval rates as well as their frequency of access and their “freshness”. In order to evaluate the HLU algorithms we have devised a function in order to have a performance metric for assessing the utilization and strength of the cache content. The following formula  $F(x)$  considers the main Web cache factors as identified in the Cache replacement problem statement in Section 2. Here, we consider a cache content  $x$  of  $N$  individual cached objects:

$$F(x) = \sum_{i=1}^N act_i \times StRatio_i \times df_i. \quad (1)$$

The above function has been introduced in the present research effort in order to consider the effect of staleness, access frequency and retrieval cost in the overall cache replacement process.

## 5. Experimentation—results

Aristotle University has installed Squid proxy cache for main and sibling caches and supports a Squid mirror site. The present paper uses for experimentation the traced information provided by this cache installation. A simulation model was developed and tested by Squid cache traces and their corresponding log files. Traces refer to the period from May to August 1999, regarding a total of almost 70,000,000 requests, of more than 900 GB content. A compact log was created for the support of an effective caching simulator, due to extremely large access logs created by the proxy. The reduced simulation log was constructed by the original Squid log fields needed for the overall simulation runs.

Squid (in its default configuration) produces four logfiles:

- *logs/access.log*: requests posed to proxy server with information regarding how many people use the cache, how much each one requested, etc.
- *logs/cache.log*: information Squid wants to know such as errors, startup messages, etc.
- *logs/store.log*: information of what is happening with our cache diskwise; it shows whenever an object is added or removed from disk.
- *cache/log*: contains the mapping of objects to their location on the disk.

The notations *HSLRU*, *HMFU*, *HLFU* correspond to the proposed algorithms with history parameter  $h = 4$ . A track of the proposed HLRU algorithms has been tested. More specifically, the notations *HLRU(2)*, *HLRU(4)* and *HLRU(6)* refer to the HLRU implementations for 2, 4 and 6 past history references, respectively. Furthermore, both random and the typical LRU cache replacement policies applied in most proxies (e.g., Squid), have been simulated in order to serve as a basis for comparisons and discussion. The performance metrics used in this simulation model focus on the cached objects cache-hit ratio, byte-hit ratio and the “weight” function  $F(x)$  (Equation (1)) normalized to the interval  $[0, 1]$ .

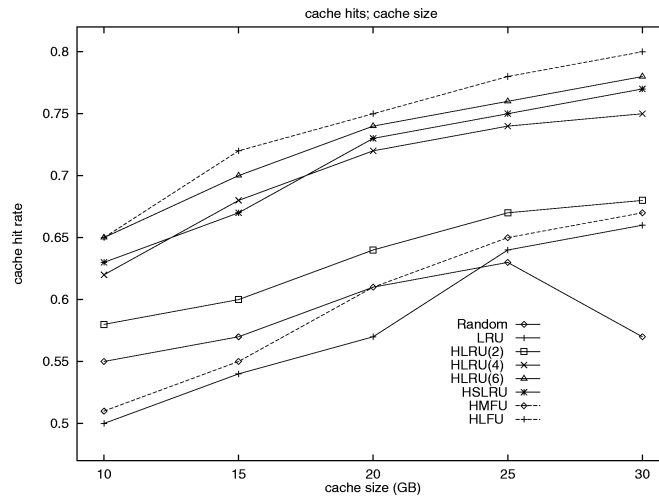


Figure 9. Cache hit rate/cache size.

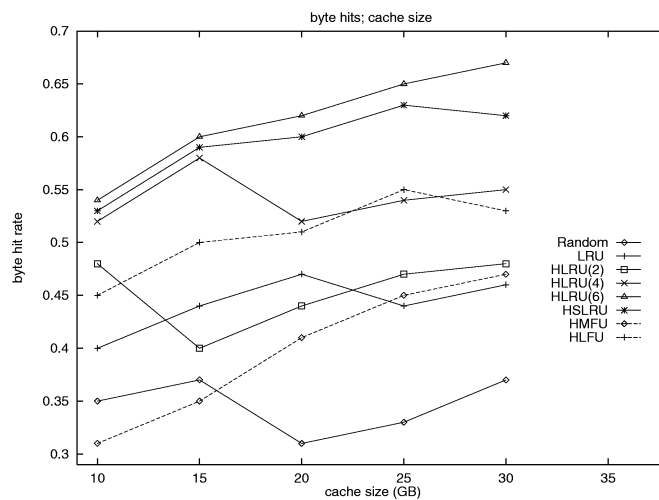


Figure 10. Bytes hit/cache size.

Figures 9 and 10 depict the cache hit and the bytes hit ratio for all of the proposed algorithms with respect to cache size. More specifically, Figure 9 presents the cache hit ratio for a cache size of 10, 15, . . . , 30 GBytes. The cache hit under *HLFU* policy outperforms the corresponding metric of all other policies, with *HLRU(6)* having similar cache hit rates. The increase in the number or past histories seem to be rather beneficial to the cache hits rates since *HLRU(6)* has better hit rates than the corresponding *HLRU(4)* and *HLRU(2)*. Overall, all *HLRU* algorithms result in better cache hits than their corresponding typical LRU approach (as expected). Furthermore, *HSLRU* presents a quite stable curve of hits

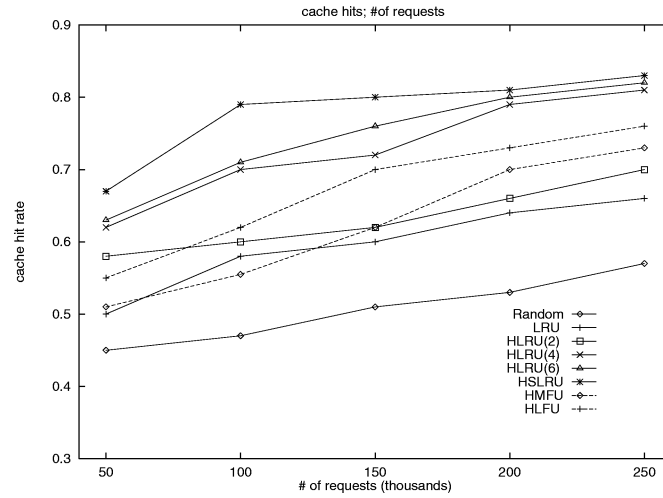


Figure 11. Cache hit rate; # of requests.

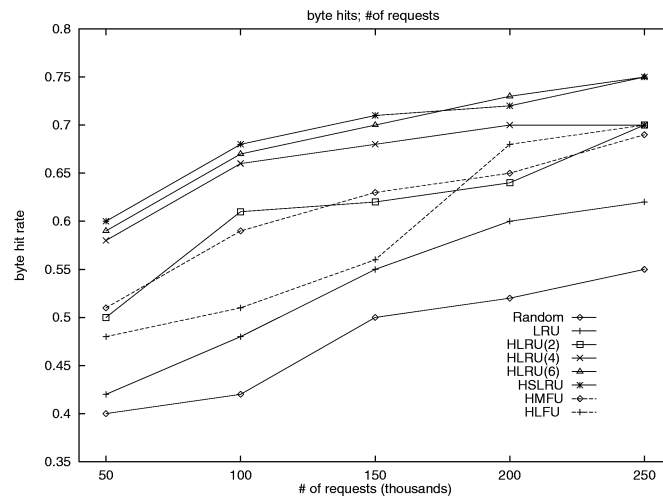


Figure 12. Bytes hit; # of requests.

rates. All algorithms converge to a cache hit rate value which is justified since the larger the cache size, the less replacement actions are performed.

Figure 10 depicts the byte hit ratio for the proposed cache replacement policies with respect to the caches sizes depicted in the cache hit rates figures. Here, the byte hit ratios of *HLRU(6)* and *HSLRU* are the top best in all experimentation runs, whereas *HMFU* and *HLRU(2)* result in more unstable curves of bytes hits. convergent byte hits. Again the *HLRU(6)* outperforms the other other *HLRU* and *LRU* implementations proving that the increase in the number of past references is beneficial to the byte hits rates as well.

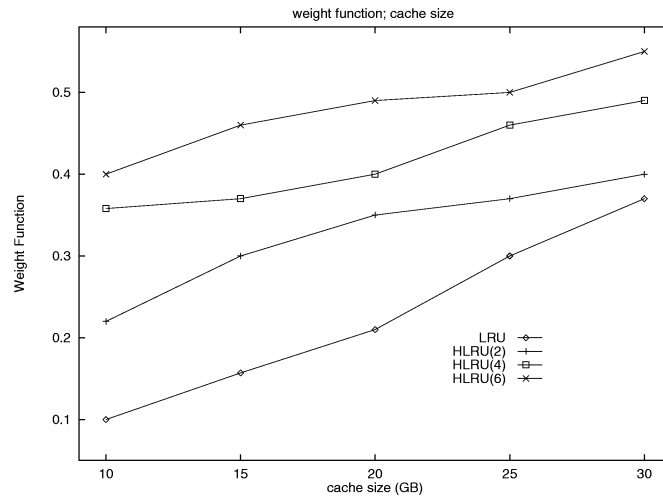


Figure 13. Weight function/cache size.

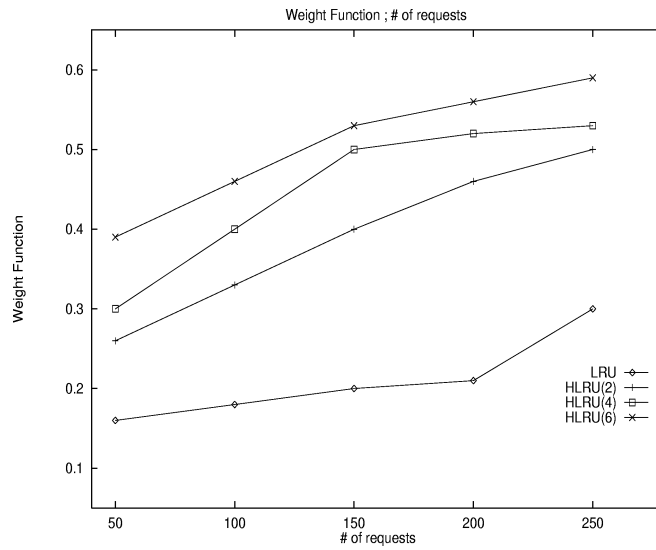


Figure 14. Weight function/# of requests.

Figures 11 and 12 depict the cache hit and the bytes hit ratio for the proposed algorithms with respect to the number of requests. More specifically, Figure 11 presents the cache hit ratio for a cache of 50, 150, . . . , 250 thousands of requests. Again cache hits under *HSLRU*, *HLFU* and *HLRU* policies result in the best cache hit rates for all number of requests used in the experimentation. Similarly, all *HLRU* algorithms result in improved cache hits as compared to the corresponding typical LRU approach and overall *HLRU*(6)

outperforms all other LRU approaches. It is important to note that all cache replacement algorithms produce better results as the number of requests increases and the cache hit rates do reach considerable hit rates (of almost 80%).

Figures 13 and 14 depict the (normalized) weight function (Equation (1)) for all four algorithms with respect to the above used cache size and the number of requests, respectively. These figures focus on the *HLRU* “track” of algorithms performance in relation to the *LRU* in order to also comment on the history parameter value. It is important to note that the values of the weight function for all *HLRU* algorithms lie in a specific region, whereas the typical *LRU* has significantly lower weight values. The importance of the specified value for the history parameter is highlighted by these results since the higher the history value (i.e., the number of past history references) the better the performance. The weight function gets to higher values for all algorithms as there is an increase in both the cache size and the number of requests.

## 6. Conclusions—future work

This paper has presented a study of applying a history based approach to the Web-based proxy cache replacement process. A history of past references is associated to each cached object and a number of cache replacement algorithms has been simulated based on a number of past “histories”. Trace-driven simulation was employed to evaluate and comment on the performance of the proposed cache replacement techniques. The simulation model was based on the Squid proxy cache server implementation and experimentation was based on workloads of a Squid proxy cache server. Results have indicated that all of the proposed history-based approaches outperform the random cache replacement and the typical popular Least-Recently-Used (LRU) policy as adopted by most currently available proxies. More specifically, results have shown that the *HSLRU*, *HLFU* and the *HLRU* algorithms significantly improve cache hit and byte hit ratios.

Further research should extend the proposed work in order to introduce replication in the Web caching process and study on the effect of the proposed cache replacement algorithms on a caching and replication scheme. Furthermore, Web cache content replacement in a hierarchical cache topology is a quite evolving and interesting research topic. Introducing the proposed algorithms in a hierarchical cache subsystem could be very helpful towards studying the impact and effectiveness of the proposed history-based algorithms.

## Acknowledgements

The author thanks the referees for their valuable contribution and remarks which have significantly improved the paper’s presentation and readability. Also, the author acknowledges the contribution of Dina Tzimorota (senior student) at the implementation part of the proposed algorithms and also thanks are ought to Panayotis Junakis (System administrator) and Savvas Anastasiades (technical staff) of the Network Operation Center at the Aristotle University, for providing access to the Squid cache traces and trace log files.

## References

- [1] C. Aggarwal, J. Wolf, and P. S. Yu, "Caching on the World Wide Web," *IEEE Trans. Knowledge Data Engrg.* 11(1), 1999, 94–107.
- [2] M. Arlitt, R. Friedrich, and T. Jin, "Performance evaluation of Web Proxy cache replacement policies," Hewlett-Packard Technical Report HPL 98-97, to appear in *Performance Evaluation J.*
- [3] M. Baentsch et al., Enhancing the Web's infrastructure: From caching to replication," *IEEE Internet Comput.* 1(2), 1997, 18–27.
- [4] A. Belloum and L. O. Hertzberger, "Document replacement policies dedicated to Web caching," in *Proc. of ISIC/CIRA/ISAS'98 Conf.*, Maryland, USA, September 1998.
- [5] A. Bestavros, R. L. Carter, and M. Crovella, "Application-level document caching in the Internet," in *Proc. of the 2nd Internat. Workshop in Distributed and Networked Environments*, SDNE, 1995.
- [6] R. Caceres, F. Douglis, A. Feldmann, C. Glass, and M. Rabinovich, "Web Proxy caching: The devil is in the details," in *Proc. of the SIGMETRICS Workshop on Internet Server Performance*, June 1998.
- [7] P. Cao, J. Zhang, and K. Beach, "Active cache: Caching dynamic contents on the Web," in *Proc. of the IFIP Internat. Conf. on Distributed Platforms and Open Distributed Processing*, Middleware, 1998, pp. 373–388.
- [8] A. Chankhunthod, P. Danzig, and C. Neerdaels, "A hierarchical Internet object cache," in *Proc. of the USENIX 1996 Annual Technical Conf.*, San Diego, CA, January 1996, pp. 153–163.
- [9] J. Dilley and M. Arlitt, "Improving proxy cache performance: Analysis of three replacement policies," *IEEE Internet Comput.* 3(6), 1999, 44–50.
- [10] "Distributed testbed for national information provisioning," <http://ircache.nlanr.net/>, 1998.
- [11] J. Gwertzman and M. Seltzer, "World Wide Web cache consistency," in *Proc. of the USENIX 1996 Annual Technical Conf.*, San Diego, CA, January 1996, pp. 141–151.
- [12] A. Iyengar and J. Challenger, "Improving Web server performance by caching dynamic data," in *Proc. of the USENIX Symposium on Internet Technologies and Systems, USITS'97*, Monterey, CA, December 1997.
- [13] T. Kroeger, D. D. E. Long, and J. Mogul, "Exploring the bounds of Web latency reduction from caching and prefetching," in *Proc. of the USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, December 1997, pp. 13–22.
- [14] S. Michel, K. Nguyen, A. Rosenstein, and L. Zhang, "Adaptive Web caching: Towards a new global caching architecture," in *Proc. of the 3rd Internat. WWW Caching Workshop*, Manchester: England, June 1998.
- [15] M. Nottingham, "Web caching documentation," [http://mnot.cbd.net.au/cache\\_docs/](http://mnot.cbd.net.au/cache_docs/), November 1998.
- [16] E. J. O'Neil, P. E. O'Neil, and G. Weikum, "The LRU-K page replacement algorithm for database disk buffering," in *Proc. of the ACM SIGMOD Conf.*, Washington, DC, USA, 1993, pp. 297–306.
- [17] O. Pearson, "The Squid cache software, Squid users guide," <http://www.auth.gr/SquidUsers/>, 1998.
- [18] K. Psounis and B. Prabhakar, "A randomized Web-cache replacement scheme," in *Proc. of the IEEE INFO-COM Conf.*, Anchorage, AL, USA, 2001.
- [19] Squid Internet object cache, mirror site, Aristotle University, <http://www.auth.gr/Squid/>, 1999.
- [20] A. Vakali, "A Web-based evolutionary model for Internet data caching," in *Proc. of the 2nd Internat. Workshop on Network-Based Information Systems, NBIS'99*, Florence, Italy, August 1999, IEEE Computer Soc. Press: Silver Spring, MD.
- [21] A. Vakali, "A genetic algorithm scheme for Web replication and caching," in *Proc. of the 3rd IMACS/IEEE Internat. Conf. on Circuits, Systems, Communications and Computers, CSCC'99*, Athens, Greece, July 1999, World Scientific/Engineering Soc. Press.
- [22] A. Vakali, "LRU-based algorithms for Web cache replacement," *First Internat. Conf. on Electronic Commerce and Web Technologies*, Lecture Notes in Computer Science, Springer: New York, 2000, pp. 409–418.
- [23] D. Wessels, "Intelligent caching World-Wide Web objects," in *Proc. of the INET'95 Conf.*, January 1995.