LRU-SP: A Size-Adjusted and Popularity-Aware LRU Replacement Algorithm for Web Caching

Kai Cheng and Yahiko Kambayashi
Graduate School of Informatics, Kyoto University
Sakyo Kyoto 606-8501, Japan
{chengk, yahiko}@isse.kuis.kyoto-u.ac.jp

Abstract

This paper presents LRU-SP, a size-adjusted and popularity-aware extension to Least Recently Used (LRU) for caching web objects. The standard LRU, focusing on recently used and equal sized objects, is not suitable for the web context because web objects vary dramatically in size and the recently accessed objects may possibly differ from popular ones. LRU-SP is built on two LRU extensions, namely Size-Adjusted LRU and Segmented LRU. As LRU-SP differentiates object size and access frequency, it can achieve higher hit rates and byte hit rates. Furthermore, an efficient implementation scheme is developed and trace-driven simulations are performed to compare LRU-SP against Size-Adjusted LRU, Segmented LRU and LRV caching algorithms.

1. Introduction

With the exponential growth of the World Wide Web, techniques for alleviating the bottlenecks to network performance have gained increasing importance. Caching is one of such techniques which stores frequently used data near the user to diminish unnecessary remote accesses. Web caching is effective because a few resources are requested often by many users, or repeatedly by a specific user. This phenomenon is known as *locality of reference*.

Cache replacement algorithm plays a key role in capturing such locality to maximize *hit rate* and other performance metrics. LRU (least recently used) is the most widely used and important replacement algorithm ever developed for main memory and disk caching. LRU exploits *temporal locality of reference*, keeping the recently used while dropping the least recently used objects. LRU is simple to implement, robust and effective in paging scenarios.

However, LRU is not suitable for web caching due to several limitations. LRU focuses on the recently used and equal sized objects, which corresponds to *R* in **Figure 1**, the recently used subset of objects .

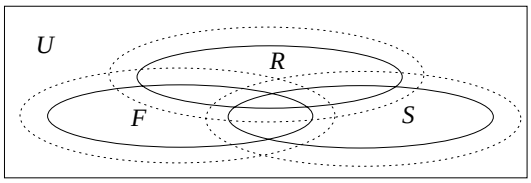

Figure 1. Relationship between the recently accessed, frequently accessed and smaller-sized objects R, F and S respectively

Web caching is on the basis of documents that vary dramatically in size. It is more profitable to cache smaller sized objects, because a larger document may take up a space for several smaller ones. Thus, it is highly desirable to keep as many as possible smaller sized objects, i.e. the subset of object *S* in **Figure 1**. On the other hand, the recently accessed data may be just temporarily referenced, so *R* may possibly differs from the frequently accessed objects *F*. It is ideal to keep the objects in $(R \cap S \cap F)$.

A number of efforts are made to extend the standard LRU [1, 2, 6, 5]. Unfortunately, up to date, there is still no efficient approach to achieving the complete goal, that is, to explicitly exploits both access frequency and object size in LRU. This is partly because of the complications in implementing such an extension in current LRU framework.

In this paper, we propose a size-adjusted and popularity-aware LRU (LRU-SP). LRU-SP is built upon two existing replacement algorithms: *Size-Adjusted LRU* and *Segmented LRU*. Size-Adjusted LRU is an $(R \cap S)$-based extension which evaluates objects in terms of both size and recency of reference. Segmented LRU is an $(R \cap F)$-based extension, in whihc objects with different access frequency are managed in different LRU queues.

The paper is organized as follows: Section 2 briefly re-

views the related work; In Section 3, we describe a couple of typical LRU extensions as the basis of our new replacement algorithm; Section 4 presents the LRU-SP replacement algorithms, containing accumulative cost-to-size model, implementation scheme; In Section 5, we experimentally evaluate LRU-SP with its predecessors; Section 6 describes some directions of future work and concludes this paper

2. Previous Work

We briefly review previous studies on replacement algorithms, especially those based on LRU.

GreedyDual-Size [3] is an interesting algorithm that combines recency of reference with object size and retrieval cost. Based on its size and cost, an object is given a initial value when it first gets in or gets a new reference. Then the value decreases little by little with time if it gets no more reference. Object with least value is the candidate to be replaced. One problem with GreedyDual-Size is that the difference of popularity has not been used. Additionally, it is required to maintain priority at a cost of $O(\log k)$ (k is number of objects in cache).

A recent technical report [4] presents *GDSP*, an popularity-based extension to GreedyDual-Size. Although similar somehow to our approach, however, just like GreedyDual-Size, this algorithm requires to maintain large priority queue .

Least Relative Value (LRV [7]) is another replacement algorithm for proxy cache. Although it has taken into account size, recency and frequency, however, it has been criticized as heavy parameterization and high overhead in implementation [3].

There are several frequency-based extensions to LRU in research of traditional paging or buffering scenarios, including *FBR* (Frequency-Based Replacement) [8], *LRU-K* and *2Q* [6], and *Segmented LRU* [5]. However, none of them take into account the variable sizes of web objects.

3. Extended LRU Policies

3.1. Segmented LRU

Segmented LRU is a frequency-based extension to basic LRU especially designed for disk caching where all pages are identical in size [5]. Segmented LRU is based on the observation that objects with at least two accesses are much more popular than those with only one access during a short interval. In Segmented LRU, cache space is partitioned into two segments: *probationary segment* and *protected segment*.

New objects (with only one access) are first faulted into the probationary segment, whereas objects with two or more accesses are kept in the protected segment. When a probationary object gets one more reference, it will change to the protected segment. When the whole cache space becomes full, the least recently used object in the probationary segment will first be replaced. The protected segment is finite in size. When it gets full, the overflowed will be *re-cached* in probationary segment. Since objects in protected segment have to go a longer way before being evicted, popular object or an object with more accesses tends to be kept in cache for longer time.

Although Segmented LRU can differentiate objects with different popularity, while keeping track of the recency of reference, it is not suitable for web caching where object size is a critical factor. When extended to web caching, another problem with this algorithm is the parameterization of segment sizes and the number of segments.

3.2. Size-Adjusted LRU

To deal with variable sizes of web objects, Charu Aggarwal et al proposed a generalized LRU replacement algorithm, namely Size-Adjusted LRU [2]. The Size-Adjusted LRU sorts all objects in cache in terms of the cost-to-size ratio, $1/(S_i \cdot \Delta T_{it})$, where S_i is the size of object i, ΔT_{it} is the elapsed time from last access to current time t. It then greedily discards those with least cost-to-size ratios from the cache. In practice, the objects are reindexed in order of nondecreasing values of $S_i \cdot \Delta_{it}$:

$$S_1 \cdot \Delta_{1t} \leq S_2 \cdot \Delta_{2t} \leq \cdots S_k \cdot \Delta_{kt}$$

Objects with highest index are one by one greedily picked and purged from the cache to make space for more potential object. Furthermore, to avoid expensive calculation of the cost-to-size ratios for all objects, a so called Pyramidal Selection Scheme (PSS) approximate scheme was developed. In this scheme, objects are classified into a limited number of groups based on $\lfloor log_2(size) \rfloor$, so that objects within a same group are similar in sizes. Each group is maintained using a LRU mechanism: a hit will make the hit object move to the most recently used end of this LRU list. The basic Size-Adjusted LRU policy only applies to a limited set of least recently used objects from all nonempty groups to make final decision. The object with largest $(S_i \cdot \Delta T_{it})$ will be purged from the cache. In Size-Adjusted LRU, objects with similar size are treated equally no matter how popular they are. This can be explained using the *cache state transition graph* or *CSTG*.

An CSTG describes how an object being cached in a lifecycle. Let s_q indicates the state an object is out of cache and s_i represents object is cached in subcache i. **Figure 2** depicts the CSTG of Size-Adjusted LRU in which access frequency does not affect the cache states.

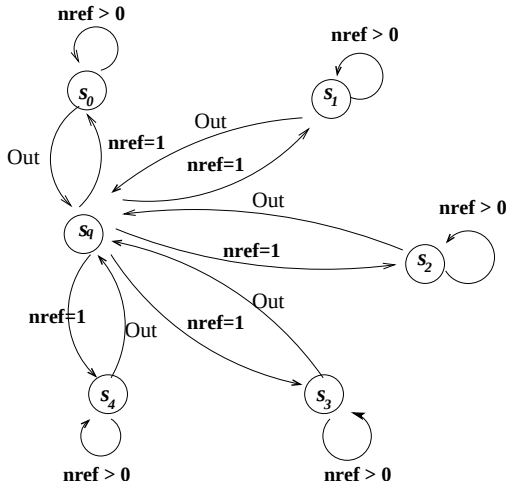

Figure 2. CSTG of Size-Adjusted LRU: access frequency does not affect cache states

4. Size-Adjusted and Popularity-Aware LRU

The basic idea behind Segmented LRU is to separate objects with different access frequency in different segments, while the key idea underlying Size-Adjusted LRU is to differentiate objects with different sizes or size-levels by keeping them in different LRU queues. The major objective of LRU-SP is to incorporate both object sizes and access frequency so as to handle long-term popularity and adjust to variable object sizes.

4.1. Handling Long-Term Popularity

Size-Adjusted LRU does not distinguish objects with different access frequency. When an object gets a hit, it just moves to the most recently used end of that LRU queue. So the objects in a same LRU queue have a similar opportunity to survive, in other words, they have nearly the same lifetime before aged out. In fact, among these objects, those with more accesses are very likely to be more popular in the near future.

As a result, even in the case that two objects have the same access frequency during a long period only the distributions are different, the cache results will be quite different. For example, if requests occur periodically (**Figure 3(a)**) and each subsequent request occurs before the object ages out, it will stay in the cache. However, if accesses are locally concentrated, the object may have aged out from cache before it gets the next access (**Figure 3(b)**) even if the average occurrence is the same as the previous case. This is not the case in Segmented LRU, because objects with more accesses are protected in a separate segment.

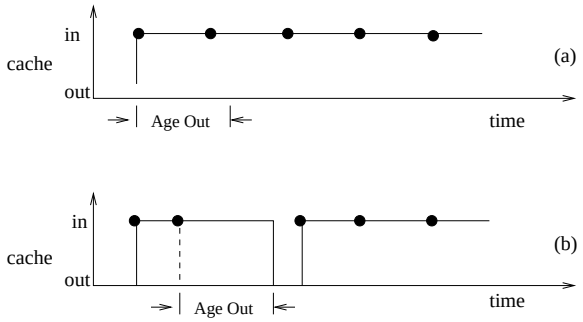

Figure 3. Popular objects being evicted early

In summary, to distinguish objects with different access frequency, cache state of an object should change with the increment of access times. This can be implemented by moving objects between different LRU queues as done in Segmented LRU, instead of being fixed in one LRU queue as in Size-Adjusted LRU.

4.2. Handling Variable Object Sizes

One of the key complications in implementing cache replacement policies for Web objects is that the objects to be cached are not necessarily identical in size. Although we can use a function to score objects with different sizes and access recency and frequency, however, this approach has proven too expensive and not suitable for Web caching. Fortunately, size is a relatively stable property compared to other characteristics of Web objects. Thus a static strategy is suitable for handling the non-identity of object sizes. Size-Adjusted LRU uses *size-based static classification* to reduce the complexity derived from non-homogeneous sizes. By this strategy, objects with similar sizes are maintained together, the differences of sizes can be overlooked. This idea can be generalized to handle more sophisticated classification.

4.3. Incorporating Size and Frequency in LRU-SP

The basic idea of LRU-SP is to incorporate the frequency-based extension of Segmented LRU into the Size-Adjusted LRU scheme. Here an important issue is how to change the cache state when an object gets one more access. To answer this problem, we should develop an extended cost-to-size model like in Size-Adjusted LRU.

New Cost-To-Size Ratio Model

We use a new cost-to-size ratio model to handle access frequency. It is been explored by different researchers that web

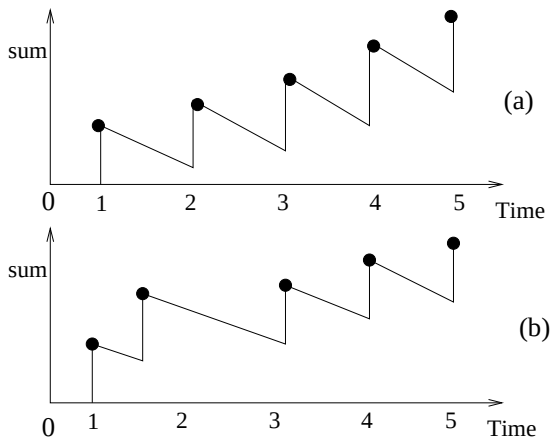

Figure 4. Popular objects remain in cache

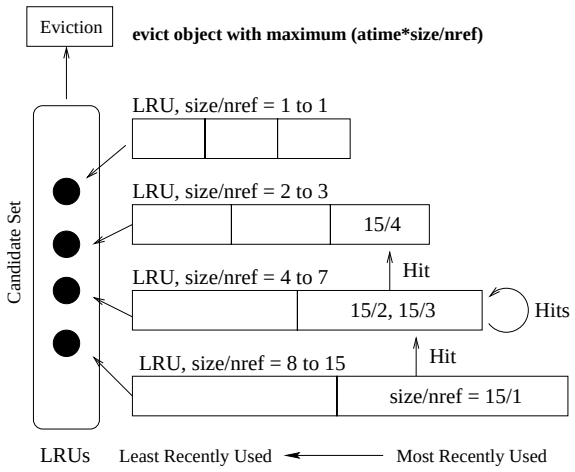

Figure 5. LRU-SP: Larger yet popular objects preserved into LRU queues for smaller objects

access is non-uniform in a great extent, the more frequently an object has been accessed, the more likely it will be reused in the future [3, 7]. Since the total cost being saved is prepositional to access times, it is reasonable to incorporate access times in the cost-to-size ratio model. Given $nref_i$ is the number of accesses since it being cached, then

$$(nref_i/(S_i \cdot \Delta T_{it}))$$

Based on this benefit-to-cost model, the influence of rereferences is utilized (represented by sum): the objects with more references can stay more time before driven out. **Figure 4(a)** and **Figure 4(b)** show the cases corresponding to **Figure 3(a)** and **Figure 3 (b)**: the objects are usually kept in cache in both cases, while the object in **Figure 3 (b)** is evicted from cache.

Implementation of LRU-SP

It is unrealistic to maintain a single priority queue for LRU-SP in terms of the above cost-to-size ratio due to the expensive computation and sorting. Thus, we devise the following approximate implementation scheme for LRU-SP (**Figure 5**): Objects are classified into a limited number of groups according to $\lfloor log_2(S_i/nref_i) \rfloor$, instead of $\lfloor log_2(S_i) \rfloor$; A hit may make the requested object move to new LRU list according to its new value of $\lfloor log_2(S_i/nref_i) \rfloor$; Each group is managed using a LRU policy. The extended cost-to-size model is applied to the eviction candidates from all nonempty groups, purging the object with largest $(\Delta T_{it} \cdot S_i/nref_i)$.

Intuitively, an object with more accesses has been treated as a smaller one $(S_i/nref_i)$, which, to some degree, awards the popular object to become competitive with less popular but small objects. In this scheme, the cache state of an object changes with each hit and with the time elapsed. According to the cache state transition graph in **Figure 6**:

1. An object will stay in the same LRU list unless it accumulates sufficient accesses;

2. The object will be cache in a next LRU list at the most recently used end if it accumulates sufficient accesses before aged out (timeout) ;

3. When an discarded object re-enters the cache, it will start again, in other words, the scheme does not keep history information for evicted objects.

Overhead and Efficiency

According to [2], Size-Adjusted LRU has $N = \lceil \log(M+1) \rceil$ LRU queues where M is the size of cache space. If the cache space is too large, the overhead in maintaining these LRU lists is considerable. However, in practice, the N is reasonable to be no larger than 20, because the objects larger than 1MB ($2^{24}B$) are quite rare. Furthermore, maintaining a LRU queue just requires a tail insertion/head taking and incurs no overhead and choosing the final victim only needs constant times (less than 20) of comparisons. In all, the overhead of this scheme is independent of the scales of object space and cache space, in other words, it is $O(1)$ in any cases. In addition, LRU-SP has no space boundaries for LRU queues, so it is a parameter-free.

Dataset	Total Requests	Total Bytes	Unique Bytes	HR_{max}	BHR_{max}
NLANR	1,848,319	21.0 GB	15.9 GB	0.228	0.245
DEC	4,985,128	45.1 GB	30.1 GB	0.476	0.332

Table 1. Profiles of trace datasets

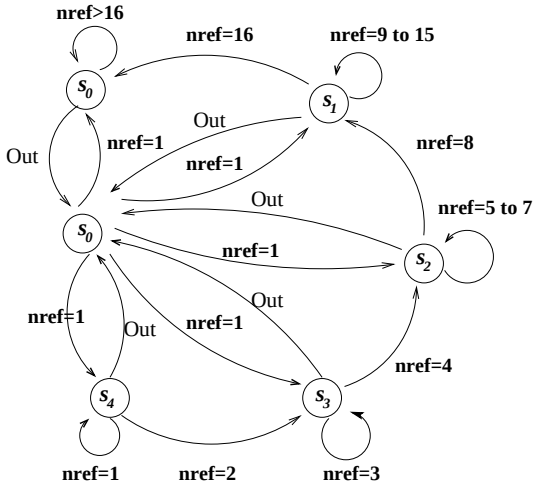

Figure 6. CSTG of Segmented LRU: cache states change with the increment of access frequency, popular objects alive longer

5. Performance Evaluation

5.1. Data Collections

We have two datasets to be used in driving our caching simulator: NLANR and DEC as summarized in **Table 1**. The dataset NLANR in **Table 1** is a one-week top level caching proxy traces publicly available [1]. This dataset contains 1,848,319 requests with total 21.0 GB Web data, where unique data is 15.9 GB with a maximum hit rate 0.228 and byte hit rate 0.245. The DEC dataset contains 5.0M requests with total 45.1 GB web data, where unique data is 30.1 GB with a maximum hit rate 0.476 and byte hit rate 0.332.

5.2. Simulation Results

We have carried out simulations on both trace datasets of NLANR and DEC. In addition to evaluation of LRU-SP and its predecessors Size-Adjusted LRU and Segmented LRU, we also compare them to *Least Relative Value (LRV)* [7], a well-known replacement algorithm, which also takes object sizes, recency and frequency of reference into account.

Outperforming Segmented LRU LRU-SP achieves much higher hit rates than Segmented LRU under two datasets, (**Figure 7** and **Figure 9**). Meanwhile, LRU-SP performs almost as well as Segmented LRU in terms of byte hit rates. This is because Segmented LRU cares nothing about object sizes, thus quite a number of smaller objects are displaced by bigger objects. Whereas LRU-SP balances well between object sizes and popularity, so it preserves a large number of smaller and popular objects in cache, which guarantees high hit rate but no harm to byte hit rates.

Outperforming Size-Adjusted LRU LRU-SP significantly improves Size-Adjusted LRU in byte hit rate under dataset NLANR (**Figure 8** and **Figure 10**) without loss of hit rate. The result is not so good for dataset DEC, because the access pattern of DEC biases towards so many small objects that the strategy of awarding bigger yet popular objects became not so effective. LRU-SP performing better than Size-Adjusted LRU demonstrates that LRU-SP has really retained bigger objects that are popular enough to make up the loss of cache space.

Outperforming Least Relative Value LRU-SP also outperforms LRV in most cases, especially in terms of hit rate. The reason is the pure LRV is unrealistic to implement. The simplified implementation of LRV given by its designers only differentiates object sizes in its first queue (for objects with only one access). While in LRU-SP, objects with different sizes are distributed among several LRU queues. So LRU-SP makes better advantage of information about sizes than LRV. Consequently, LRU-SP can achieve higher hit rates than LRV.

6. Conclusion and Future Work

High performance, low overhead and adaptability to access patterns are desirable properties to Web caching. Based on the analysis of two LRU extensions, in this paper, we have proposed a new cache replacement algorithm, namely, LRU-SP. LRU-SP incorporates key factors to Web caching in a consistent way, which integrates the major advantages of its predecessors such as low overhead, adaptability, and improves them by significantly reducing "early evictions" and "cache pollution". Trace-driven simulations show LRU-SP outperforms its predecessors.

[1] ftp://ircache.nlanr.net/Traces/

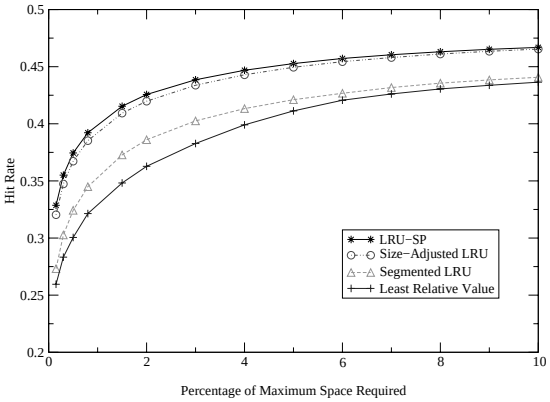

Figure 7. Hit rates under DEC dataset

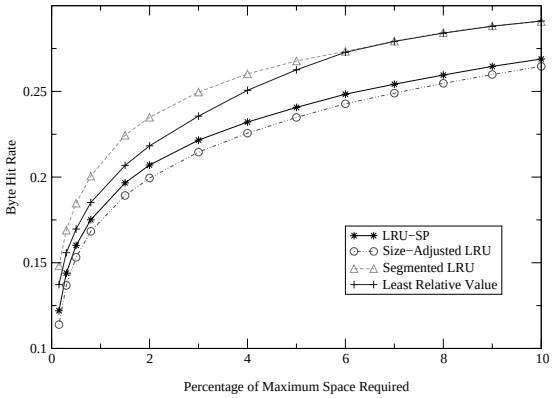

Figure 8. Byte hit rates under DEC dataset

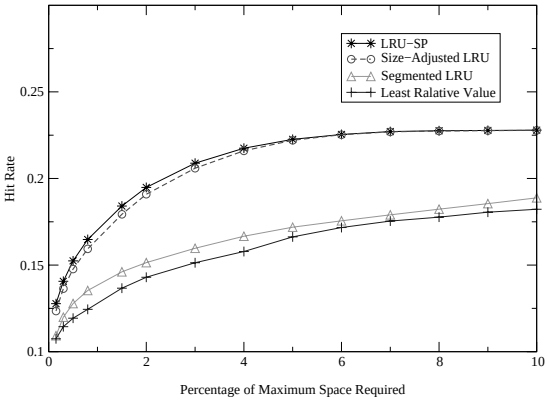

Figure 9. Hit rates under NLANR dataset

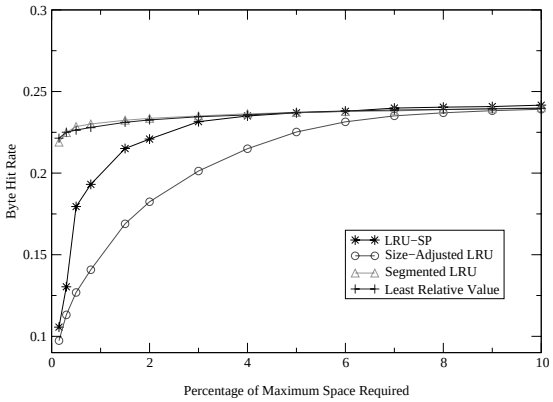

Figure 10. Byte hit rates under NLANR dataset

Meanwhile, we also found several directions need further work. One is to introduce precise access frequency. Currently we only use access times since an object entering the cache. It is important to use k-size window as in LRU-K to calculate real dynamic frequency. Another work is to explore the possibility of introducing more application-level information as well as new cache performance metrics.

References

[1] M. Abrams, C. R. Standridge, G. Abdulla, S. Wililams, and E. A. Fox. Caching Proxies: Limitations and Potentials. In *Proceedings of the Fourth International WWW Conference*, 1995.

[2] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the World Wide Web. *IEEE transactions on knowledge and data engineering*, 11(1), 1999.

[3] P. Cao and S. Irani. Cost-Aware WWW Proxy Caching Algorithms. In *Proceedings of the 1997 USENIX Symposium on Internet Technology and Systems*, pages 193–206, December 1997.

[4] S. Jin and A. Bestavros. Popularity-Aware GreedyDual-Size Web Caching Algorithms. Technical Report TR-99/09, Computer Science Department, Boston University, 1999.

[5] R. Karedla, J. S. Love, and B. G. Wheery. Caching Strategies to Improve Disk System Performance. *IEEE Computer*, 27(3):38–46, March 1994.

[6] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K Page Replacement Algorithm for Database Disk Buffering. In *Proceedings of ACM SIGMOD International Conference on Management of Data*, pages 297–306, New York, 1993.

[7] L. Rizzo and L. Vicisano. Replacement Policies for a Proxy Cache. Technical report rn/98/13, University College London, Department of Computer Science, Gower Street, London WC1E 6BT, UK, 1998. http://www.iet.unipi.it/ luigi/ caching.ps.gz.

[8] J. T. Robinson and M. V. Devarakonda. Data Cache Management Using Frequency-based Replacement. *Performance Evaluation Review*, 18(1):134–142, May 1990.