Using Full Reference History for Efficient Document Replacement in Web Caches

Hyokyung Bahn[1] Sam H. Noh[2] Sang Lyul Min[3] Kern Koh[1]

[1]*Department of Computer Science, Seoul National University, Seoul 151-742, Korea. http://oslab.snu.ac.kr*

[2]*Department of Computer Engineering, Hong-Ik University, Seoul 121-791, Korea. http://www.cs.hongik.ac.kr/~noh*

[3]*Department of Computer Engineering, Seoul National University, Seoul 151-742, Korea. http://archi.snu.ac.kr*

Abstract

With the increase in popularity of the World Wide Web, the research community has recently seen a proliferation of Web caching algorithms. This paper presents a new such algorithm, that is efficient and robust, called Least Unified-Value (LUV). LUV evaluates a Web document based on its cost normalized by the likelihood of it being re-referenced. This results in a normalized assessment of the contribution to the value of a document, leading to a fair replacement policy. LUV can conform to arbitrary cost functions of Web documents, so it can optimize any particular performance measure of interest, such as the hit rate, the byte hit rate, or the delay-savings ratio. Unlike most existing algorithms, LUV exploits complete reference history of documents, in terms of reference frequency and recency, to estimate the likelihood of being re-referenced. Nevertheless, LUV allows for an efficient implementation in both space and time complexities. The space needed to maintain the reference history of a document is only a few bytes and furthermore, the time complexity of the algorithm is $O(\log_2 n)$, where n is the number of documents in the cache. Trace-driven simulations show that the LUV algorithm outperforms existing algorithms for various performance measures for a wide range of cache configurations.

1. Introduction

In an effort to relieve the problem of network congestion and latency on the World Wide Web (WWW), the research community has recently seen a proliferation of Web cache replacement algorithms [1, 2, 3, 5, 6, 8, 9, 10, 14]. This paper presents yet another such algorithm. However, the algorithm that we propose has the following salient features:

- First, it shows the best performance for a wide range of cache configurations in terms of popular performance measures used in evaluating Web caching algorithm, namely, the hit rate, the byte hit rate, and the delay-savings ratio.

- Second, the proposed algorithm makes full use of all of the past activities, in terms of reference frequency and recency, made upon the Web cache in deciding which document to evict. By so doing, it not only accurately distinguishes actively referenced documents and those that are not so, i.e. hot and cold documents, but it also distinguishes those documents that are hot but are getting colder, and those that are cold but are getting hotter. This is the main reason behind the superior performance.

- Third, the implementation of the replacement algorithm is efficient in both space and time complexities. The space needed to maintain the reference history of a document is only a few bytes per document and furthermore, the time complexity of the algorithm is $O(\log_2 n)$, where n is the number of documents in the cache. This is a feature that is not easy to satisfy when the *size* of the document and *cost* of fetching documents from remote sites have to be incorporated into the algorithm, as it is in Web caching [5,6,8].

- Fourth, the replacement algorithm retains the above features irrespective of what the performance measure of interest is. As mentioned above, in Web

caching environments, the typical performance measures of interest are the hit rate, the byte hit rate, and the delay-savings ratio. The proposed algorithm can easily be conformed to execute based on a particular performance measure. This is unlike many previous algorithms that tightly couple the optimization of a particular performance measure into the algorithm itself.

Before describing the algorithm, we describe, in the next section, the measures that have been the focus of interest and optimization in the Web caching realm. We also briefly make comparisons of previously presented algorithms. We focus on the differences and main features of the algorithms rather than describing them individually in detail. In Sections 3 and 4, we describe the proposed algorithm, namely, the Least Unified-Value (LUV) algorithm and its implementation. In Section 5, we describe the results of the simulation experiments, and compare LUV's performance with previously proposed algorithms. Finally, we conclude in Section 6.

2. Performance Measures and Related Works

Performance measures of interest in the Web caching realm can be defined according to the goal of caching. The three popular performance measures used in Web caching, that is, the hit rate, the byte hit rate, and the delay-savings ratio, denoted as HR, BHR, and DSR, respectively, can be described as follows:

$$\text{HR} = \sum h_i / \sum r_i$$

$$\text{BHR} = \sum (s_i \cdot h_i) / \sum (s_i \cdot r_i)$$

$$\text{DSR} = \sum (d_i \cdot h_i) / \sum (d_i \cdot r_i)$$

where

h_i : number of hit references to document *i,*
r_i : total number of references to document *i* (number of hits + number of misses),
s_i : size of document *i,*
d_i : delay time to fetch document *i* from the original server to the cache.

HR is the measure used in traditional caching systems such as file caching and database buffer management, and represents the number of hit references over the total number of references. BHR represents the number of bytes saved from retransmission by using the cache over the total amount of bytes referenced. BHR considers the size of the Web document, but does not consider the difference in retrieval costs. Among documents that are of the same size, those that incur higher cost in retrieving the document should be retained in the cache longer than those that incur lower cost. DSR, which considers this matter in terms of retrieval latency, represents the reduced latency by virtue of a cache hit over the total latency incurred when assuming caches are not used [8]. One may also define other new performance measures that reflect the focus of interest one wants to measure. For example, Kelley et al. define VHR (value hit rate) which represents a normalized measure of social welfare [14]. In this case, performance is measured by simply replacing the delay time (d_i) in DSR by *value* in VHR.

Many of the previous algorithms proposed for Web caching have attempted to optimize performance for one particular measure. The LRU, LFU, SIZE [9], HYBRID [10], and LNC-R-W3 [8] are such algorithms (Note the *Performance Measure* column of Table 1). These algorithms have a weakness that as the performance measure of interest changes due to circumstantial changes, they have difficulty in adjusting to these changes. The GD-SIZE [2], LRV [5], sw-LFU [14], and LUV algorithms, on the other hand, are robust to changes in the performance measure of interest. However, for the LRV algorithm, the time complexity varies according to the performance measures. While the algorithm is efficient for the byte hit rate measure, for other measures the complexity of the algorithm is $O(n)$, which makes it impractical as an on-line algorithm**.** For this reason, a recent version of LRV restricts their optimization of the algorithm to only the byte hit rate measure [15]. Another weakness of the LRV is that its replacement decision is based on extensive empirical analysis of trace data. Finally, the MIX algorithm tries to optimize the combination of the three measures, i.e., HR, BHR, and DSR [6].

For caching algorithms to be practical, it is important that the time complexity of the algorithm not be excessive, preferably not higher than $O(\log_2 n)$ where n is the number of documents in the cache [2,4]. Algorithms that do not meet this criterion are the LNC-R-W3 and MIX algorithms (Note the *Complexity* column of Table 1).

Table 1. A summary of Web caching algorithms.

Algorithm	Maintaining Reference History		Performance Measure for which the Algorithm Optimizes	Complexity (n is the number of cached objects)		Advantage/Weakness	
	Recency	Frequency		Time	Space	Advantage	Weakness
LRU	Last reference time	No	Fixed to BHR	$O(1)$	$O(n)$	Simple to implement	Fixed performance measure; Considers partial aspects of reference history
LFU	No	Number of references while in cache	Fixed to BHR	$O(\log_2 n)$	$O(n)$		
SIZE [9]	No	No	Fixed to HR	$O(\log_2 n)$	$O(n)$	Keeps many documents in cache	Fixed performance measure; Does not consider reference history
HYBRID [10]	No	Number of references while in cache	Fixed to DSR	$O(\log_2 n)$	$O(n)$ + per server information	Good estimation of download latency	Per server information overhead; Fixed performance measure
LNC-R-W3 [8]	k-th reference time	Based on k-th reference time	Fixed to DSR	$O(n)$	$O(kn)$ + replaced document's information	Perception of normalized contribution to DSR	Time complexity; Space overhead
LRV [5]	Last reference time	Number of references	Fixed to BHR	$O(1)$	$O(n)$ + replaced document's information + parameter value	Considers trace characteristics	Trace analysis overhead
			Any other measure	$O(n)$			
GD-SIZE [2]	Last reference time	No	HR, BHR, DSR, or any other measure	$O(\log_2 n)$	$O(n)$	No parameter; k-competitive; Any performance measure possible	Does not consider frequency; May cause cache pollution with high cost documents
MIX [6]	Last reference time	Number of references while in cache	HR, BHR, and DSR combined	$O(n)$	$O(n)$	Considers all primary parameters	Time complexity
sw-LFU [14]	Last reference time used only as a tie breaker	Number of references while in cache	HR, BHR, DSR, or any other measure	$O(\log_2 n)$	$O(n)$	Any performance measure possible	Does not consider recency except to break ties
LUV	Time of all past references	Number of references while in cache	HR, BHR, DSR, or any other measure	$O(\log_2 n)$	$O(n)$	Uses complete reference history; Best performance; Any performance measure possible	Parameter tuning

In terms of maintaining previous reference history, most of the algorithms maintain and use the recency of the last reference to the document, with the exception of the LNC-R-W3 algorithm that uses the k-th reference (Note the *Maintaining Reference History* column of Table 1). The LNC-R-W3 algorithm uses a strategy similar to the LRU-k algorithm that was proposed for buffer caching [7]. In contrast, the LUV algorithm, as we will show later, uses the reference recency history of all past references.

As for the reference frequency history, some algorithms simply use the frequency count, while others combine this information with the recency history that is maintained. The GD-SIZE algorithm, however, does not consider any frequency information. Again, note that the LUV algorithm uses the frequency count as well as the recency history of all prior references to a document.

Another aspect that may be considered in classifying replacement algorithms is the way in which reference history is maintained. Basically, there are two ways to maintain the reference history of documents. One is in-cache-history and the other is perfect-history. The in-cache-history method retains the reference history of only those objects that are in the cache. Hence, reference information is lost once the object is evicted. On the other hand, the perfect-history method retains the reference history of an object even after its eviction, allowing this information to be used when it returns to the cache. This method may offer considerable information about the reference behaviors, resulting in better prediction of future references than the in-cache-history method. However, it incurs more space and time overhead for maintaining the information of the evicted objects. To resolve this problem, perfect-history algorithms may choose to retain approximations of the reference history.

Breslau et al. show that perfect-history LFU outperforms in-cache-history LFU in terms of HR and BHR [13]. Recency history, i.e. reference time information, can also be maintained as either in-cache-history or perfect-history. In fact, most caching algorithms may be implemented using both in-cache-history and perfect-history methods. However, this paper basically deals with in-cache-history algorithms with the exception of some algorithms that inherently use perfect-history such as the LNC-R-W3 and LRV algorithms. Other algorithms shown in Table 1 are in-cache-history algorithms unless explicitly stated.

Other features of various Web caching algorithms, including advantages and weaknesses, are summarized in Table 1.

3. The LUV (Least Unified-Value) Algorithm

In this section, we describe the Least Unified-Value (LUV) replacement algorithm. LUV evaluates a Web document based on its retrieval cost normalized by the likelihood of it being re-referenced. This results in a normalized assessment of the contribution to the value of a document, leading to a fair replacement policy. Like most existing algorithms, LUV associates a value *Value*(*i*) to each document *i* in the cache and, when needed, replaces the document that has the smallest *Value*. *Value*(*i*) in LUV is defined as

$$Value(i) = Weight(i) \cdot H(i).$$

Weight(*i*) denotes the retrieval cost of a document per unit size and is defined as

$$Weight(i) = c_i / s_i$$

where c_i and s_i denotes the cost and the size of document *i,* respectively. c_i can be defined differently according to the performance measure of interest, and for the HR, BHR, and DSR measures, the c_i value used would normally be 1, the document size, and the download latency, respectively. For other measures such as social welfare as referred to by Kelly et al. [14], one can define c_i appropriately for the performance measure of interest.

H(*i*) represents the likelihood of re-reference, that is, the worth of the document based on observations of past behavior. For the LUV algorithm, this is based on the recency and frequency history of all of the past references on document *i*. This notion is taken directly from the LRFU replacement policy [4]. Each reference to a document *i* in the past contributes to *H*(*i*) and a reference's contribution is determined by a weighing function *F*(*x*) where *x* is the time span from the reference in the past to the current time. For example, assume that document *i* was referenced at t_1, t_2, and t_3. Then, *H*(*i*) at current time, t_c, is computed by

$$H(i) = F(\delta_1) + F(\delta_2) + F(\delta_3)$$

where $\delta_1 = t_c - t_1$, $\delta_2 = t_c - t_2$, and $\delta_3 = t_c - t_3$.

A formal definition of *H*(*i*) is, then, given as

$$H(i) = \sum_{k=1}^{n} F(t_c - t_k)$$

where t_c is the current time, *n* is the number of references made to document *i* since it has been brought into the cache, and t_k is the time of the *k*-th reference. *F*(*x*) would generally be a decreasing function as more weight should be given to more recent references. In this paper, we use the function that was chosen in the original LRFU presentation [4], that is,

$$F(x)=(1/2)^{\lambda x} \quad (0 \le \lambda \le 1).$$

```
if document i is already in the cache
{   Update the Value of i;
    Adjust the position of i in the heap appropriately;
}
else
{   Fetch document i from the original site;
    Give an initial Value to i;
    while(Size of i is larger than free space)
        Remove the root and reorganize the heap;
    Insert document i into the proper place in the heap;
}
```

Figure 1. Operation upon request for a document on the Web.

When λ is equal to 0, $H(i)$ simply counts the number of previous references and LUV is reduced to the sw-LFU [14], which is basically a weighted LFU. As λ increases, LUV gives more weight to more recent references. When λ is equal to 1, the original LRFU is reduced to the LRU algorithm, which considers only the last reference time. Then, at first glance, LUV may again be thought of as a weighted LRU similar to GD-SIZE [2]. However, this is not the case, though LUV indeed gives more weight to more recent references. Moreover, there is an important difference between LUV with λ=1 and GD-SIZE. Both of the algorithms increase the *Value* of a document when it is referenced, and decrease it as time progresses. This aging mechanism reflects the recency of past references in caching environments where non-uniform costs may be associated with the objects. While LUV applies the same decremental-*rates* to all of the documents in the cache, GD-SIZE applies the same decremental-*values*. For example, let the *Value* of documents *A* and *B* be 1000 and 10, respectively, at time *t*, and no reference is made to these documents up to time *t*+1. Then, when the same decremental-value of 1 is applied, as in GD-SIZE, the *Value* of *A* and *B* becomes 999 and 9, respectively, at time *t*+1. When the same decremental-rate of 0.1 is applied, as in LUV, the *Value* of *A* and *B* becomes 900 and 9, respectively. Though both methods are identical in philosophy, the same decremental-value method has a weakness that it may incur cache pollution when the cost of documents has large variations as it may be difficult to age documents with large costs in such environments.

4. Implementation of the LUV Algorithm

According to the description of the LUV algorithm given in Section 3, computing the *Value* of each document requires all of the past reference times. At first thought, this may seem impractical as space and time overhead for attaining and maintaining this information may seem infeasible. Moreover, *Value* of each document changes over time, and this necessitates recomputing the *Value* of all documents in the cache as time progresses. Here, we show that this is not the case, and that an efficient implementation is possible. Similar arguments were presented for the LRFU algorithm [4]. In this paper, we show that the original argument can be extended to the Web caching realm, where the *Weight* factor is incorporated into the document value calculations.

Property 1. Let the *Value* of document i at the n-th reference time t and the $(n+1)$-th reference time t' be $Value_t(i)$ and $Value_{t'}(i)$, respectively. Then $Value_{t'}(i)$ is derived from $Value_t(i)$ as follows:

$$Value_{t'}(i) = Value_t(i) \cdot F(\delta) + Weight(i)$$

where $\delta = t' - t$.

Proof. Let δ_k denote the time interval between the k-th reference time and the n-th reference time where $1 \le k \le n$. Then,

$$\begin{aligned}
&Value_{t'}(i) \\
&= Weight(i)\, H_{t'}(i) \\
&= Weight(i)\{F(\delta_1+\delta)+ \cdots +F(\delta_n+\delta) + F(0)\} \\
&= Weight(i)\{(1/2)^{\lambda(\delta_1+\delta)}+ \cdots +(1/2)^{\lambda(\delta_n+\delta)} + (1/2)^0\} \\
&= Weight(i)\{(1/2)^{\lambda\delta_1}+ \cdots +(1/2)^{\lambda\delta_n}\}(1/2)^{\lambda\delta} + Weight(i) \\
&= Weight(i)\{F(\delta_1)+ \cdots +F(\delta_n)\} F(\delta) + Weight(i) \\
&= Weight(i)\, H_t(i)\, F(\delta) + Weight(i) \\
&= Value_t(i)\, F(\delta) + Weight(i)
\end{aligned}$$

□

Property 1 states that the *Value* at the time of the $(n+1)$-th reference can be computed directly from the time of the n-th reference and the *Value* and *Weight* at that time. Property 1 implies that the space complexity of LUV is constant per document, thereby resolving the space complexity problem.

Table 2. Characteristics of traces used in the experiments.

Trace	Trace gathering period	Total Requests	Total Unique Requests	Total Mbytes	Total Unique Mbytes
DEC [11]	09/01/1996 – 09/22/1996	609951	306685	6415.64	3826.34
NLANR [12]	07/18/1999 – 07/31/1999	2688258	1405635	23051.80	14290.83

Lemma 1. Let the *Value* of document i at time t and t' ($t' > t$) be $Value_t(i)$ and $Value_{t'}(i)$, respectively. If there have been no references to document i between the time interval t and t', $Value_{t'}(i)$ is derived from $Value_t(i)$ as follows:

$$Value_{t'}(i) = Value_t(i) \cdot F(\delta)$$

where $\delta = t' - t$.

Proof. Suppose document i had been referenced n times before t, and let δ_k denote the time interval between the k-th reference time and t. Then,

$$\begin{aligned}
&Value_{t'}(i) \\
&= Weight(i)\, H_{t'}(i) \\
&= Weight(i)\{F(\delta_1+\delta)+ \cdots +F(\delta_n+\delta)\} \\
&= Weight(i)\{(1/2)^{\lambda(\delta_1+\delta)}+ \cdots +(1/2)^{\lambda(\delta_n+\delta)}\} \\
&= Weight(i)\{(1/2)^{\lambda\delta_1}+ \cdots +(1/2)^{\lambda\delta_n}\}(1/2)^{\lambda\delta} \\
&= Weight(i)\{F(\delta_1)+ \cdots +F(\delta_n)\}F(\delta) \\
&= Weight(i)\, H_t(i)\, F(\delta) \\
&= Value_t(i)\, F(\delta)
\end{aligned}$$ □

Lemma 1 states that the *Value* of document i can be computed directly from a recently computed *Value* and the time this *Value* was computed.

Property 2. If $Value_t(a) > Value_t(b)$ and neither a nor b have been referenced after t, then $Value_{t'}(a) > Value_{t'}(b)$ holds for any t' ($t' > t$).

Proof. Let $\delta = t' - t$. Then, from Lemma 1,

$$Value_{t'}(a) = Value_t(a)\, F(\delta) > Value_t(b)\, F(\delta) = Value_{t'}(b)$$
$$(\because F(\delta) > 0)$$ □

Property 2 shows that the relative ordering of the documents in the cache does not change if they are not re-referenced. This property, along with Property 1, allows LUV to be implemented by a heap structure, where the time complexity of operations on this data structure is $O(\log_2 n)$. Figure 1 shows the algorithm that is invoked upon a request for a document.

5. Experimental Results

In this section, we discuss the results from trace-driven simulations performed to assess the effectiveness of the LUV algorithm. We used two public traces: Digital Equipment Corporation Web proxy traces (DEC) and access logs of proxy caches at the National Lab for Applied Network Research (NLANR). DEC traces used in our simulations are local-client traces that are references made by clients within Digital's Palo Alto campus [11]. NLANR provides ten sanitized proxy cache access logs: *bo*1, *bo*2, *lj*, *pa*, *pb*, *rtp*, *sd*, *sj*, *sv*, and *uc*. Among these *sj* was used in our experiments [12]. Table 2 shows the characteristics of the traces. In the experiments, we filtered out some requests such as UDP requests, *cgi_bin* requests, and requests whose size is larger than the cache size used in the simulation.

The performance of LUV is compared with those of the LRU, LFU, SIZE, HYBRID, LRV, GD-SIZE, LNC-R-W3, MIX, and sw-LFU in terms of HR, BHR, and DSR. Before discussing the results themselves, we point out for clarity, some of the peculiarities of the presented results and algorithms.

- In case of the LRV algorithm, we analyzed each trace a priori and used the complete parameter value information to reflect the characteristics of the target traces. Note that this would not be possible in real life.
- For the GD-SIZE algorithm, it has been reported that using the *cost* (c_i) value of 1 instead of the document download latency results in a higher DSR [2]. It is not clear why this is so, but we think this is due to the decrements used in GD-SIZE as described previously. As GD-SIZE applies the same decrements to all of the documents in the cache, it is slow in aging documents that have high latencies possibly

leading to cache pollution. To deal with this situation, we experimented with both cost values, that is, 1 and the download latency, and selected the best results.

- LNC-R-W3 and LRV are inherently perfect-history algorithms [5,8]. For LFU, both in-cache-history and perfect-history methods are used. All other algorithms are in-cache-history algorithms.
- For GD-SIZE, sw-LFU, and LFU (both in-cache-history LFU and perfect-history LFU), the LRU order is used as a tie breaker. That is, when there is more than one document with the same *Value*, we select the victim by the LRU order.
- For the LUV algorithm, the λ value was tuned to reflect the characteristics of the given traces.

Figures 2 and 3 show the HR, BHR, and DSR of each algorithm as a function of the cache size for the DEC and NLANR traces, respectively. In the figures, *Infinite* represents the performance when the cache size is equal to *Total Unique Mbytes* in Table 2, which is essentially equivalent to having an infinite size cache. The *x*-axis, which is the cache size, is the size relative to the *Infinite* cache size. We simulated a wide range of cache sizes, from 0.05% to 30% of the *Infinite* cache size, to see the relation between the various cache sizes and the performance of each algorithm. Small cache size results, such as when the size is 0.05%, may be applicable to main memory caching, while results for large sizes may be appropriately used for disk caching. Note that the scale of the *x*-axis is not uniform. This arrangement is intended to show clearly the relative performances of the algorithms. Note that for clarity, we show the results in two groups of algorithms. The first (left-hand side) shows LUV, LNC-R-W3, GD-SIZE, LRV, and the two types of LFU, i.e. in-cache-history LFU (denoted by I-LFU) and perfect-history LFU (denoted by P-LFU), while the second (right-hand side) shows LUV, LRU, HYBRID, SIZE, MIX, and sw-LFU. From the results, the following observations can be made.

- For most cases, the LUV algorithm performs the best irrespective of the cache size for all performance measures, while the other algorithms give and take amongst each other at particular cache sizes and measures.
- Algorithms not considering the recency history perform worse than recency-based algorithms when the cache size is small. The performance of the SIZE, HYBRID, and LFU-based policies is consistently inferior and the gap is quite wide for most of the cache sizes that were studied. The only range that they come close is when the cache is large. Note that these algorithms do not consider the recency of past references at all, which is primarily considered in other algorithms compared in our experiments.
- We find that generally, frequency-based algorithms are inferior to recency-based replacement algorithms. However, as implied by the superior performance of the LUV algorithm, a good combination of recency and frequency can lead to even better performance.
- MIX shows good performance for all performance measures, though the performance gap between MIX and LUV is somewhat wider for smaller cache sizes. Recall, however, that the MIX algorithm requires $O(n)$ time complexity.
- Though perfect-history LFU shows better performance than in-cache-history LFU for all cases, there is no significant advantage of other perfect-history algorithms, that is, LNC-R-W3 and LRV over in-cache-history algorithms. In fact, even with perfect-history, none of these algorithms performed better than the LUV algorithm. Moreover, LNC-R-W3 requires $O(n)$ time complexity and LRV requires trace analysis overhead.

6. Conclusion

In this paper, we presented a Web cache replacement algorithm called LUV. LUV evaluates a Web document based on its retrieval cost normalized by the likelihood of it being re-referenced. This results in a normalized assessment of the contribution to the value of a document, leading to a fair replacement policy. Unlike most previous algorithms, LUV estimates the re-reference likelihood of a document by its full reference history during cache residency. Despite this, we showed that LUV allows for an efficient implementation in both space and time complexities. The space needed to maintain the reference history of a document is only a few bytes and furthermore, the time complexity of the algorithm is $O(\log_2 n)$, where n is the number of documents in the cache. Through trace-driven simulations we showed that the LUV algorithm performs better

than existing algorithms for various performance measures for a wide range of cache configurations.

One direction for future research is to model the distribution of Web requests seen by Web proxy caches considering both recency and frequency. Breslau et al. show that an independent request stream following a Zipf-like distribution is sufficient to model the requests seen at Web proxies [13]. However, our experimental results show that temporal locality is also important. Hence, we are attempting to model Web requests considering both the recency and frequency information to simultaneously reflect the reference popularity and the temporal locality. This will allow us to fix a priori the control parameter λ, which currently has to be determined through off-line experiments, according to the corresponding model.

Also, in this paper, we only considered the in-cache-history LUV algorithm, which requires low overhead in both time and space. However, because only a few bytes of information is required for each object, perfect-history LUV may improve performance even more without incurring much system overhead. We are currently conducting experiments to validate this conjecture.

Acknowledgment

We would like to thank NLANR and DEC for making their proxy traces available. Without their generosity this paper would not exist. Many thanks to the anonymous reviewers for their very helpful comments.

References

[1] M. Abrams, C. Standridge, G. Abdulla, S. Williams, and E. Fox, "Caching proxies: Limitations and potentials," *In Proceedings of the 4th International WWW Conference*, pp. 119-133, 1995.

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

[3] S. Glassman, "A caching relay for the World Wide Web," *Computer Networks and ISDN system*, vol. 27, no. 2, pp. 165-173, 1994.

[4] D. Lee, J. Choi, J. Kim, S. H. Noh, S. L. Min, Y. Cho, and C. Kim, "On the Existence of a Spectrum of Policies that Subsumes the LRU and LFU Policies," *In Proceedings of the 1999 ACM SIGMETRICS Conference*, pp.134-143, 1999.

[5] P. Lorenzetti, L. Rizzo, and L. Vicisano, "Replacement Policies for a Proxy Cache," http://www.iet.unipi.it/~luigi/caching.ps.gzeunis, 1996.

[6] N. Niclausse, Z. Liu, and P. Nain, "A new and efficient caching policy for the world wide web," *In Workshop on Internet Server Performance (WISP'98)*, 1998.

[7] E. O'Neil, P. O'Neil, and G. Weikum, "The LRU-*k* Page Replacement Algorithm for Database Disk Buffering," *In Proceedings of the 1993 ACM SIGMOD Conference*, pp. 297-306, 1993.

[8] P. Scheuermann, J. Shim, and R. Vingralek, "A Case for Delay-Conscious Caching of Web Documents," *In Proceedings of the Sixth International WWW Conference*, 1997.

[9] S. Williams, M. Abrams, C. Standridge, G. Abdulla, and E. Fox, "Removal policies in network caches for world-wide web documents," *In Proceedings of the ACM SIGCOMM '96*, pp. 293-305, 1996.

[10] R. Wooster and M. Abrams, "Proxy caching that estimates page load delays," *In Proceedings of the Sixth International WWW Conference*, 1997.

[11]ftp://ftp.digital.com/pub/DEC/traces/proxy/locallistv1.2.html

[12] ftp://ircache.nlanr.net/Traces/sj_sanitized-access_990718.gz ~ sj_sanitized-access_990731.gz

[13] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web Caching and Zipf-like Distributions: Evidence and Implications," *In Proceedings of Infocom '99*, 1999.

[14] T. Kelly, Y. Chan, S. Jamin, and J. MacKie-Mason, "Biased Replacement Policies for Web Caches: Differential Quality-of-Service and Aggregate User Value," *In Proceedings of the Fourth International Web Caching Workshop*, 1999.

[15] L. Rizzo and L. Vicisano, "Replacement Policies for a Proxy Cache," *Technical Report UCL-CS RN/98/13*, 1998 (available at http://www.iet.unipi.it/~luigi).

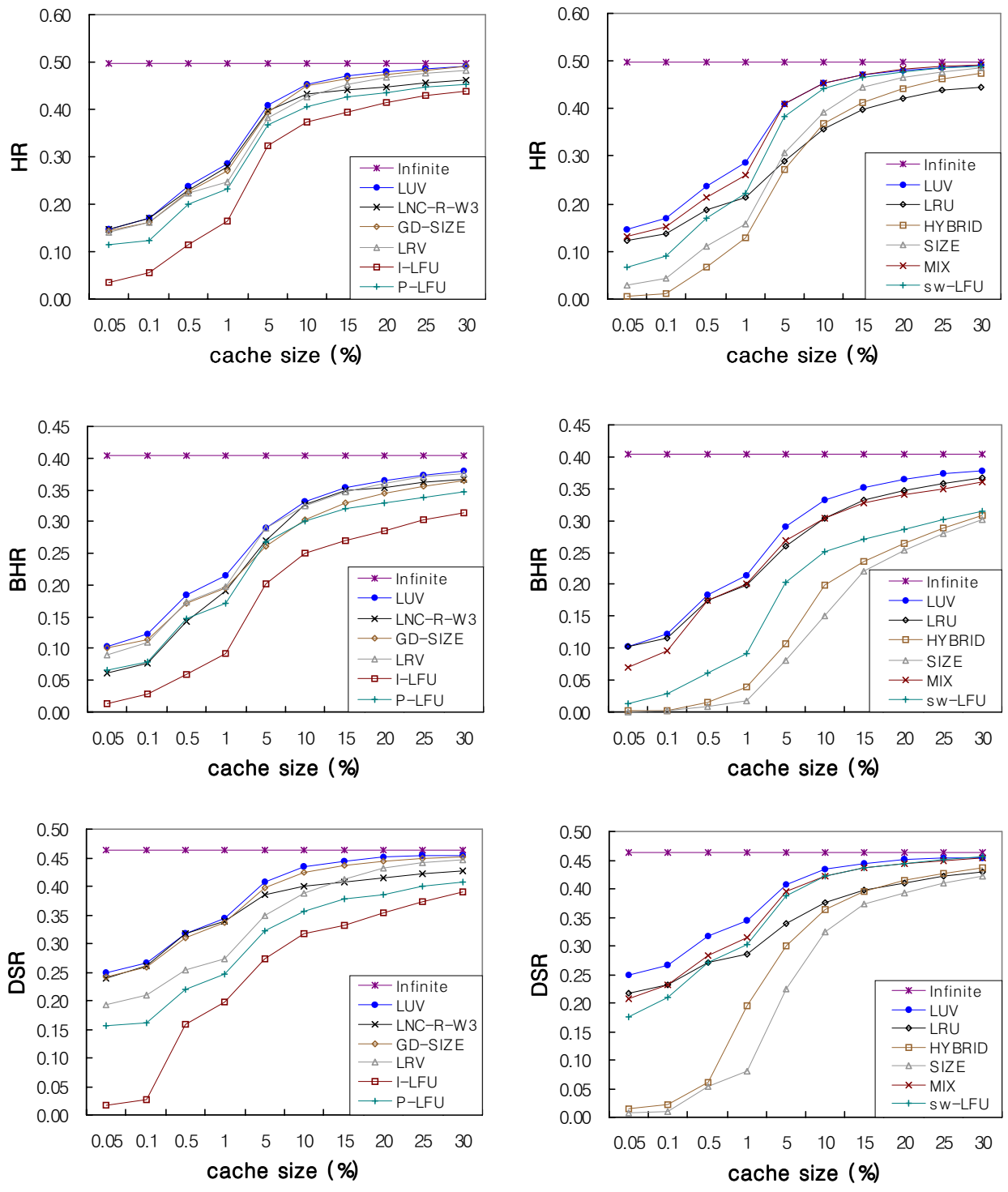

Figure 2. Comparison of LUV with other algorithms using DEC trace.

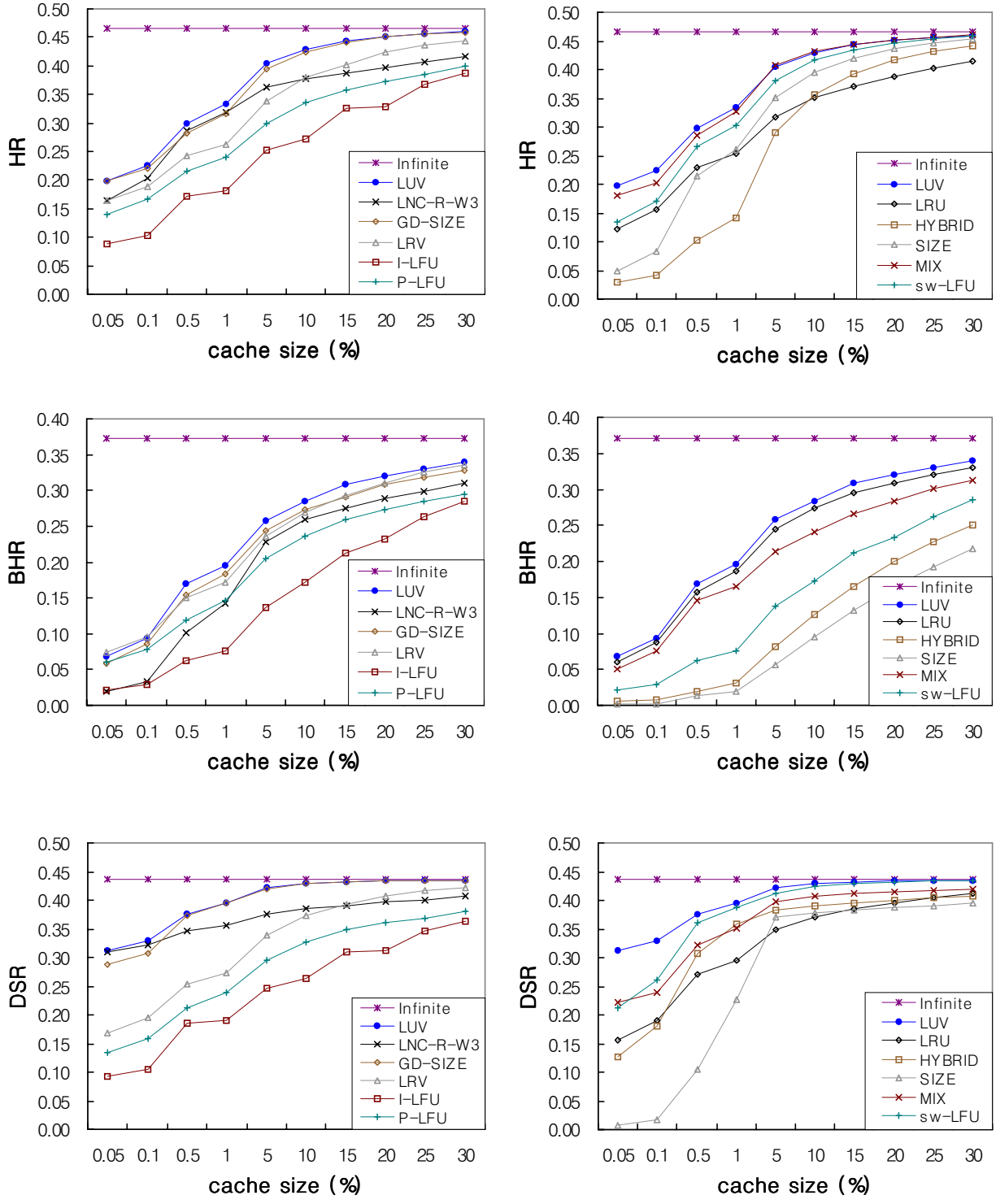

Figure 3. Comparison of LUV with other algorithms using NLANR trace.