A Randomized Web-Cache Replacement Scheme

Konstantinos Psounis[†], Balaji Prabhakar[‡]
[†]Department of Electrical Engineering
[‡]Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, CA 94305
kpsounis@leland.stanford.edu, balaji@isl.stanford.edu

Abstract—
The problem of document replacement in web caches has received much attention in recent research, and it has been shown that the eviction rule "replace the least recently used document" performs poorly in web caches. Instead, it has been shown that using a combination of several criteria, such as the recentness and frequency of use, the size, and the cost of fetching a document, leads to a sizeable improvement in hit rate and latency reduction. However, in order to implement these novel schemes, one needs to maintain complicated data structures. We propose randomized algorithms for approximating any existing web-cache replacement scheme and thereby avoid the need for any data structures.

At document-replacement times, the randomized algorithm samples N documents from the cache and replaces the least useful document from the sample, where usefulness is determined according to the criteria mentioned above. The next $M < N$ least useful documents are retained for the succeeding iteration. When the next replacement is to be performed, the algorithm obtains $N - M$ new samples from the cache, and replaces the least useful document from the $N - M$ new samples and the M previously retained. Using theory and simulations, we analyze the algorithm and find that it matches the performance of existing document replacement schemes for values of N and M as low as 8 and 2 respectively. Rather surprisingly, we find that retaining a small number of samples from one iteration to the next leads to an exponential improvement in performance as compared to retaining no samples at all.

Keywords— **Web caching, document replacement policies, randomized algorithm.**

I. Introduction

THE enormous popularity of the World Wide Web in recent years has caused a tremendous increase in network traffic due to HTTP requests. Since the majority of web documents are static, caching them at various network points provides a natural way of reducing traffic. At the same time, caching reduces download latency and the load on web servers.

A key component of a cache is its replacement policy, which is a decision rule for evicting a page currently in the cache to make room for a new page. The rule that replaces the least recently used (LRU) page from the cache, is the most popular replacement policy. This is due to a number of reasons: LRU is an optimal online algorithm in the competitve ratio sense [1], it only requires a linked list to be efficiently implemented as opposed to more complicated data structures required for other schemes, and takes advantage of temporal locality in the request sequence [2].

This research is supported in part by a Stanford Graduate Fellowship, and a Terman Fellowship.

[1] LRU is k-competitive and there is no deterministic online algorithm with a competitive ratio smaller than k [7].

[2] A sequence of requests is said to exibit temporal locality if the probability to request an object after k requests, given that it was just requested, is inversly proportional to k.

Suppose that we associate with any replacement scheme a *utility function*, which sorts pages according to their suitability for eviction. For example, the utility function for LRU assigns to each page a value which is the time since the page's last use. The replacement scheme would then replace that page which is most suitable for eviction.

Whereas for processor caches LRU and its variants have worked very well [11], it has recently been found [2] that LRU is not suitable for web caches. This is because some important differences distinguish a web cache from a processor cache: (i) The size of web documents are not the same, and (ii) the cost of fetching different documents varies significantly. These differences do not occur in a processor cache. Thus, a utility function that takes into account not only the popularity of a web document, but also its size and cost of fetching can be expected to perform significantly better. Recent work proposes many new cache replacement schemes that exploit this point (e.g. LRU-Threshold[1], GD-Size[2], GD*[5], LRV[6], SIZE[12], Hybrid[13]).

However, the data structures that are needed for implementing these new utility functions turn out to be complicated. Most of them require a priority queue in order to reduce the time to find a replacement from $O(K)$ to $O(\log K)$, where K is the number of documents in the cache. Further, these data structures need to be *constantly updated* (i.e., even when there is no eviction), although they are solely used for eviction.

This prompts us to consider randomized algorithms which do not need any data structures. For example, the particularly simple Random Replacement (RR) algorithm evicts a document drawn at random from the cache [7]. However, as might be expected, the RR algorithm does not perform very well.

We propose to combine the benefits of both the utility function based schemes and the RR scheme. Thus, consider a scheme which draws N documents from the cache and evicts the least useful document in the sample. The "usefulness" of a document is as determined by the utility function. Although this basic scheme performs better than RR for small values of N, we find a tremendous improvement in performance by refining it as follows: After replacing the least useful of N samples, the identity of the next $M < N$ least useful documents is retained in memory. At the next eviction time, $N - M$ new samples are drawn from the cache and the least useful of these $N - M$ and M previously retained is evicted, and the identity of the M least useful of the remaining is stored in memory, and so on.

Intuitively, the performance of an algorithm that works on a

few randomly drawn samples depends on the quality of the samples. Therefore, by deliberately tilting the distribution of the samples towards the good side, which is precisely what the refinement achieves, one expects an improvement in performance. Rather surprisingly, we find that the performance improvement can be *exponential* for small values of M (e.g. 1, 2 or 3).

The rest of the paper is organized as follows. In Section II we present the randomized algorithm, and in Section III we analyze it. In particular, we find that a small value of M leads to a big improvement in performance compared to when $M = 0$. On the other hand, we find that choosing too high a value of M degrades the performance, since we will have too few fresh samples. Section IV investigates the variation in performance as M increases from 0 to N. Specifically, we prove that the performance of the algorithm is convex in M with most the benefit obtained for small values of M. In Section V we derive a simple approximate closed form formula for the optimal value of M as a function of N. Section VI presents trace driven simulations comparing the randomized scheme with various existing deterministic schemes. We find that even with small values of N and M (e.g. 8 and 2 respectively) the randomized scheme performs very competitively. Section VII concludes the paper.

II. A Description of the Algorithm

The first time a document is to be evicted, N samples are drawn at random from the cache and the least useful of these is evicted. After replacing the least useful document from the sample, the next $M < N$ least useful documents are retained for the next iteration. And when the next replacement is to be performed, the algorithm obtains $N - M$ new samples from the cache, and replaces the least useful document from the $N - M$ new samples and the M previously retained. This procedure is repeated whenever a document needs to be evicted. Figure 1 presents the algorithm in pseudo-code.

```
if (eviction) {
   if (first_iteration) {
      sample(N);
      evict_least_useful;
      keep_least_useful(M);
   } else {
      sample(N-M);
      evict_least_useful;
      keep_least_useful(M);
   }
}
```

Fig. 1. The randomized algorithm.

An error is said to have occurred if the evicted document *does not* belong to the least useful n^{th} percentile of all the documents in the cache, for some desirable values of n. Thus, the goal of the algorithm we consider is to minimize the probability of error. We shall say that a document is *useless* if it belongs to the least useful n^{th} percentile[3].

[3]Note that samples that are good eviction candidates will be called "useless" samples since they are useless for the cache.

It is interesting to conduct a quick analysis of the algorithm described above in the case where $M = 0$ so as to have a benchmark for comparison. Accordingly, suppose that all the documents are divided into $100/n$ bins according to usefulness and N documents are sampled uniformly and independently from the cache. Then the probability of error equals $(1 - n/100)^N$,[4] which approximately equals $e^{-nN/100}$. By increasing N this probability can be made to approach 0 exponentially fast. (For example, when $n = 8$ and $N = 30$, the probability of error is approximately 0.08. By increasing N to 60, the probability of error can be made as low as 0.0067.)

But it is possible to do much better without doubling N! That is, even with $N = 30$, by choosing $M = 9$, the probability of error can be brought down to 2.4×10^{-6}. In the next few sections we obtain models to further understand the effect of M on performance.

We end this section with the following remark. Whereas it is possible for a document whose id is retained in memory to be accessed between iterations, making it a "recently used document", we find that in practice the odds of this happening are negligibly small[5]. Hence, in all our analysis, we shall assume that documents which are retained in memory are not accessed between iterations.

III. The Model and Preliminary Analysis

In this section we derive and solve a model that describes the behavior of the algorithm precisely. We are interested in computing the probability of error, which is the probability that none of the N documents in the sample is useless for the cache, for any given N and n and for all $M (0 \leq M < N)$.

We proceed by introducing some helpful notation. Of the M samples retained at the end of the $(m-1)^{th}$ iteration, let Y_{m-1} $(0 \leq Y_{m-1} \leq M)$ be the number of useless documents. At the beginning of the m^{th} iteration, the algorithm chooses $N - M$ fresh samples. Let A_m, $0 \leq A_m \leq N - M$ be the number of useless documents coming from the $N - M$ fresh samples. In the m^{th} iteration, the algorithm replaces one document out of the total $Y_{m-1} + A_m$ available (so long as $Y_{m-1} + A_m > 0$) and retains M documents for the next iteration. Note that it is possible for the algorithm to discard some useless documents because of the memory limit of M that we have imposed.

Define $X_m = \min(M + 1, Y_{m-1} + A_m)$ to be precisely the number of useless documents in the sample just prior to the m^{th} document replacement, that the algorithm would ever replace at eviction times. If $X_m = 0$, then the algorithm commits an error at the m^{th} eviction. It is easy to see that X_m is a Markov chain and satisfies the recursion

$$X_m = \min(M + 1, X_{m-1} - 1_{(X_{m-1}>0)} + A_m),$$

and that A_m is binomially distributed with parameters $N - M$ and $n/100$. For a fixed N and n, let $p_k(M) = P(A_m = k)$, $k = 0, \ldots, N - M$, denote the probability that k useless documents for the cache, and thus good eviction candidates, are acquired during a sampling. When it is clear from the context we will

[4]Although the algorithm samples without replacement, the values of N are so small compared to the overall size of the cache that $(1 - n/100)^N$ almost exactly equals the probability of error.

[5]Trace driven simulations in Section VI support our observation.

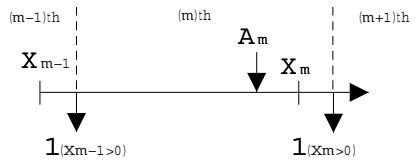

Fig. 2. Sequence of events per iteration. Note that eviction takes place prior to resampling.

abbreviate $p_k(M)$ to p_k. Figure 2 is a schematic of the above Markov chain.

Let T_M denote the transition matrix of the chain X_m for a given value of M. The form of the matrix depends on whether M is smaller or larger than $N/2$. Since we are interested in small values of M, we shall suppose that $M \leq N/2$ [6]. It is immediate that T_M is irreducible and has the general form

$$T_M = \begin{pmatrix} p_0 & p_1 & p_2 & \dots & p_M & 1-\sum_{i=0}^{M} p_i \\ p_0 & p_1 & p_2 & \dots & p_M & 1-\sum_{i=0}^{M} p_i \\ 0 & p_0 & p_1 & \dots & p_{M-1} & 1-\sum_{i=0}^{M-1} p_i \\ 0 & 0 & p_0 & \dots & p_{M-2} & 1-\sum_{i=0}^{M-2} p_i \\ \vdots & & & \ddots & & \vdots \\ 0 & 0 & 0 & \dots & p_0 & 1-p_0 \end{pmatrix}$$

As may be inferred from the transition matrix, the Markov chain models a system with one deterministic server, binomial arrivals, and a finite queue size equal to M (the system's overall size is $M+1$). An interesting feature of the system is that as M increases, the average arrival rate, $E(A_m) = (N-M)n/100$, decreases linearly and the maximum queue size increases linearly.

Let $\pi = (\pi_0, \dots, \pi_{M+1})$ denote the stationary distribution of the chain X_m. Clearly π_0 is the probability of error as defined above. Let $A = (a_{ij})$ be an $(M+2) \times (M+2)$ matrix, with $a_{ij} = 1$ for all i, j. Let $a = (a_i)$ be a $1 \times (M+2)$ matrix with $a_i = 1$ for all i. Since T_M is irreducible, $I - T_M + A$ is invertible [8] and

$$\pi = a \cdot (I - T_M + A)^{-1}. \quad (1)$$

Figure 3 shows a collection of plots of π_0 versus M for different values of N and n. The minimum value of π_0 is written on top of each figure. We note that given N and n there are values of $M > 0$ for which the error probability is very small compared to its value at $M = 0$. We also observe that there is no need for N to be a lot bigger than the number of bins $100/n$ for the probability of error to be as close to zero as desired, since even for $N = 2 \cdot 100/n$ the minimum probability of error is extremely small. Finally, we notice that for small values of M there is a huge reduction in the error probability and that the minimum is achieved for a small M. As M increases further the performance deteriorates linearly.

The exponential improvement for small M can be intuitively explained as follows. For concreteness, suppose that $M = 1$

[6] Figure 3 suggests that the M at which the probability of error is minimized is less than $N/2$.

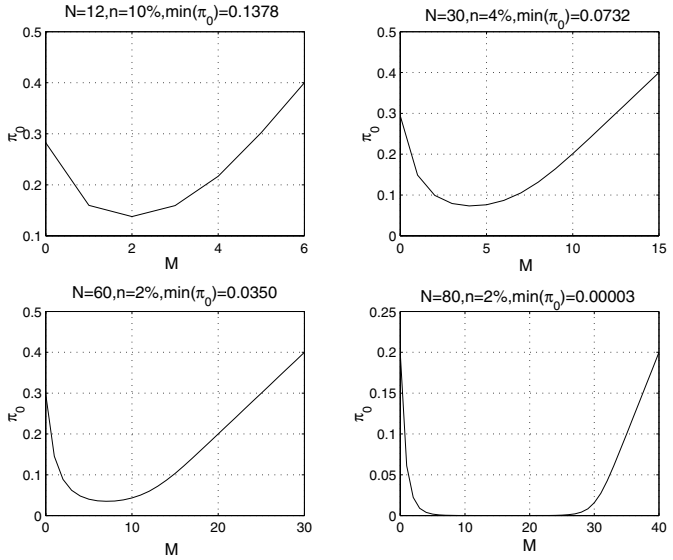

Fig. 3. Probability of error (π_0=probability not a useless document for the cache is replaced) versus number of documents retained (M).

and that the Markov chain X_m has been running from time $-\infty$ onwards (hence it is in equilibrium at any time $m \geq 0$). The relationship $\{X_m = 0\} \subset \{A_m = 0; A_{m-1} \leq 1\}$ immediately gives that $P(X_m = 0) \leq P(A_m = 0)[P(A_{m-1} = 0) + P(A_{m-1} = 1)]$. Supposing that $N \geq 2 \cdot 100/n$, $P(A_m = 0) \approx e^{-3}$ and $P(A_m = 1) \approx 3e^{-3}$. Therefore $P(X_m = 0) \leq 4e^{-6}$. Compare this number with the case $M = 0$, where $P(X_m = 0) = P(A_m = 0) \approx e^{-3}$, and the claimed exponential improvement is apparent.

IV. A Closer Look at the Performance Curves

From Figure 3 it is evident that for the specific values of N and n used in the plots, as M increases from small to large values, the error probability decreases exponentially, flattens out, and then increases linearly. In Figure 4, we plot the error probability for $N = 30$ and various small values of n, to investigate the behavior of π_0 when the value of the sample deteriorates. We observe that in all these curves the error probability is a convex function of M, almost always possessing the features mentioned above.

It is therefore interesting to investigate if this convexity holds for any value of N and n. Before launching into proofs, we briefly give an insight into why the error probability is convex in M.

Fix the values of N and n and let A and B be two instantiations of the scheme proposed, with memory sizes of M and $M+1$ respectively. Let λ_A (respectively, λ_B) be the average arrival rate of useless documents from resampling in system A (respectively, system B). Since $\lambda_A = (N-M)n/100 > (N-M-1)n/100 = \lambda_B$, system A gets more useless documents from resampling than system B on the average. However, the queue size of system A, which equals M, is smaller than that of system B's by one place. Hence there will be times at which system A will be full and drop samples, while system B will be able to accommodate an extra sample. When M increases

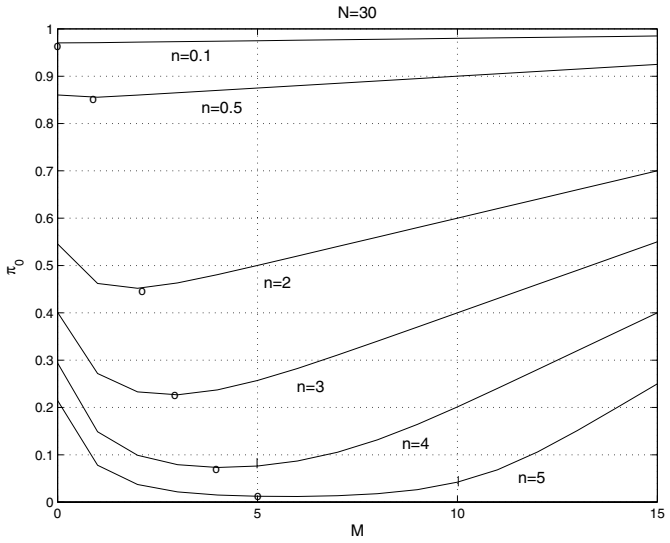

Fig. 4. Convexity of π_0 as a function of M.

from 0 to 1, the positive effect of an increase in the queue size offsets the negative effect of a decrease in the arrival rate. As M increases further, it is less likely for overflows to occur and the dominating phenomenon is the decrease in arrival rate. This trade-off between high arrival rate and high queue size causes π_0 to be a convex function of M, and thus there is an optimal value of M at which π_0 is minimized.

To establish convexity directly, it would help greatly if $\pi_0(M)$ could have been expressed as a function of the elements of T_M in closed form. Unfortunately, this is not the case and we must use an indirect method, which seems interesting in its own right. Our method consists of relating $\pi_0(M)$ to the quantity $D(t, M, \lambda(M))$, which is the number of overflows in the time interval $[0, t]$ from a buffer of size M with average arrival rate $\lambda(M)$. Let

$$\overline{D}(M) \stackrel{\triangle}{=} \lim_{t\to\infty} D(t, M, \lambda(M))/t.$$

Theorem 1: The probability of error is convex in M.

Sketch of proof: Let $A^M[0, t]$ be the number of arrivals in $[0, t]$. Then the probability the system is full as observed by arrivals, or equivalent the probability of drops, equals

$$P_{drop} = \lim_{t\to\infty} \frac{D(t, M, \lambda(M))}{A^M[0, t]} = \frac{\overline{D}(M)}{\lambda(M)}.$$

Lemma 3 below implies that $\overline{D}(M)$ is convex in M.

Proceeding, equating effective arrival and departure rates we obtain

$$\begin{aligned} \lambda(M) \cdot (1 - P_{drop}(M)) &= (1 - P_{empty}(M)), \\ \text{or } P_{empty}(M) &= 1 - \lambda(M) + \overline{D}(M). \end{aligned} \quad (2)$$

Since $\lambda(M) = (N - M)\frac{n}{100}$ is linear in M, and $\overline{D}(M)$ is convex in M, Equation (2) implies $P_{error} = P_{empty}(M)$ is convex in M. ∎

To complete the proof it remains to show that $\overline{D}(M)$ is convex in M. The proof of the convexity of $\overline{D}(M)$ is carried out

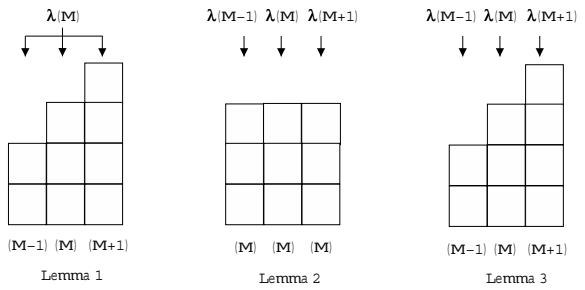

Fig. 5. The cases relevant for Lemma 1, Lemma 2, and Lemma 3 respectively.

in Lemmas 1, 2, and 3 below. In the following we abbreviate $D(t, M, \lambda(M))$ to $D(t, M)$ when the arrival process does not depend on M, and to $D(t, \lambda(M))$ when the buffer size is constant, regardless of the value of M. Lemma 1 shows that $D(t, M)$ is convex in M for all $t > 0$. Lemma 2 shows the convexity of $\lim_{t\to\infty} D(t, \lambda(M))/t$. Finally, Lemma 3 shows the convexity of $\overline{D}(M) = \lim_{t\to\infty} D(t, M, \lambda(M))/t$. Figure 5 schematically describes the cases that each lemma deals with.

Due to limitations of space, we only give a sketch of the somewhat combinatorially involved proofs of these results. The full proofs can be found in [10].

Lemma 1: $D(t, M)$ is a convex function of M, for each $t > 0$.

Sketch of proof: To prove convexity it suffices to show that the second order derivative of the number of drops is non-negative; i.e., that $(D(t, M-1) - D(t, M)) - (D(t, M) - D(t, M+1)) \geq 0$. This can be done by comparing the number of drops $D(t, M-1)$, $D(t, M)$, and $D(t, M+1)$ from systems with buffer sizes $M-1$, M, and $M+1$ respectively, under identical arrival processes, as shown in figure 5. Essentially, the comparison entails considering the situations for buffer occupancies in the three systems that lead to drops. ∎

Let $D(\lambda(M)) \stackrel{\triangle}{=} \lim_{t\to\infty} D(t, \lambda(M))/t$.

Lemma 2: $D(\lambda(M))$ is a convex function of M when $\lambda(M) = (N - M)\frac{n}{100}$.

Sketch of proof: We need to show $D(\lambda(M-1)) - 2D(\lambda(M)) + D(\lambda(M+1)) \geq 0$ by considering three systems with same buffer sizes and binomially distributed arrival processes with average rates $\lambda(M-1)$, $\lambda(M)$ and $\lambda(M+1)$, as shown in figure 5. Thus, there will be common arrivals and exclusive arrivals as categorized below:

(a) An arrival occurs at all three systems.

(b) An arrival occurs only at the system with buffer size $M-1$.

(c) An arrival occurs at the two systems with buffer sizes $M-1$ and M and there is no arrival at the system with buffer size $M+1$.

Due to the arrival rates being as in the hypothesis of the lemma, category (b) and (c) arrivals are identically distributed. Using this and combinatorial arguments one can then show that $D(\lambda(M))$ is convex. ∎

Lemma 3: $\overline{D}(M)$ is a convex function of M when $\lambda(M) = (N - M)\frac{n}{100}$.

TABLE I

OPTIMUM VALUES OF π_0 AND M FOR VARIOUS N AND n.

N	$min(\pi_0)$			M^*		
	n=10	n=20		n=10	n=20	
8	0.3643	0.0593		1	2	
10	0.2450	0.0110		1	3	
12	0.1378	0.0011		2	4	
	n=5	n=10		n=5	n=10	
20	0.1946	0.0013		2	5	
	n=4	n=8		n=4	n=8	
30	0.0732	2.4454^{-6}		4	9	
	n=3	n=6	n=9	n=3	n=6	n=9
40	0.0558	8.0595^{-8}	4.6629^{-15}	5	12	16
	n=2	n=4	n=6	n=2	n=4	n=6
50	0.1354	1.8678^{-6}	9.5368^{-14}	4	13	18
60	0.0350	8.3933^{-11}	-	7	19	-
70	0.0025	-	-	11	-	-
80	3.1553^{-5}	-	-	16	-	-

TABLE II

COMPARISON OF OPTIMUM VALUES OF M FOR VARIOUS N AND n, CALCULATED FROM MG APPROXIMATION (M^o) AND FROM MC (M^*).

N	M^o, M^*					
	n=10		n=20			
8	0	1	1.3	2		
10	0	1	2.6	3		
12	0.5	2	3.9	4		
	n=5		n=10			
20	0	2	5.5	5		
	n=4		n=8			
30	2.2	4	10.3	9		
	n=3		n=6		n=9	
40	3.0	5	13.8	12	18.7	16
	n=2		n=4		n=6	
50	0	4	14.3	13	20.8	18
60	4.8	7	20.9	19	-	-
70	10.4	11	-	-	-	-
80	16.4	16	-	-	-	-

Sketch of proof: We consider three systems of buffer sizes $M-1$, M and $M+1$, whose arrival processes are Binomially distributed with rates $\lambda(M-1)$, $\lambda(M)$ and $\lambda(M+1)$, as shown in figure 5. This is a combination of Lemma 1 and 2. ■

V. ON THE OPTIMAL VALUE OF M

The objective of this section is to derive an approximate closed form expression for the optimal value of M for a given N and n.

Let $M^* = \arg\min\{\pi_0(M)\}$ be the optimal value of M. As remarked earlier, even though the form of the transition matrix, T_M, allows one to write down an expression for $\pi_0(M)$, there is no closed form solution from which one might calculate M^*. Thus, we numerically solve Equation (1), compute $\pi_0(M)$ for all $M \leq N/2$, and read off M^* for various values of N and n, as done in Table I. This table is to be read as follows: For example, suppose N=30 and $n = 4\%$, the minimum value of π_0 is 0.0732 and it is achieved at $M^* = 4$.

Even though exact closed form solutions from which one might calculate M^* are hard if not impossible to obtain, we can derive an approximate close form solution using elementary martingale theory [4]. Recall that X_m is the number of useless documents in the sample and that $X_{m+1} = ((X_m - 1 + A_m) \wedge (M+1)) \vee 0$[7]. The boundaries at 0 and $M+1$ complicate the analysis of this Markov Chain (MC). The idea is to work with a MC that has no boundaries, consider its respective exponential martingale, and then use the *Optional Stopping Time Theorem* [4] to take into account the boundaries at 0 and $(M+1)$. Due to limitations of space we skip the derivation and only present the final result. The full derivation can be found in [10].

The approximate closed form solution for the optimal value of memory is quite simple and is given by

$$M^o = N - \sqrt{(N+1)100/n}. \quad (3)$$

[7]The symbols $\wedge$ and $\vee$ denote the minimum and maximum operations, respectively.

In Table II we compare the results for the optimal M obtained by: (i) Equation (3), denoted by M^o, and (ii) by the MC model, denoted by M^*. This table is to be read as follows: For example, suppose $N = 40$ and $n = 6$, the optimal M equals (i) $M^o = 13.8$, and (ii) $M^* = 12$. The approximation is quite accurate over a large range of values of N and n for reasonable operating conditions. In particular, it is only when the number of fresh samples $(N - M)$ is less than the number of bins $(100/n)$, for example for $n = 2$, $N = 50$, and $M > 0$, that M^o is not very close to M^*.

VI. TRACE DRIVEN SIMULATIONS

In this section we conduct web-trace driven simulations to evaluate the performance of our algorithm under real traffic. In particular, we approximate deterministic cache replacement schemes using our randomized algorithm, and compare the performance of the deterministic schemes with the performance of the randomized algorithm. Recall that any cache replacement algorithm is characterized by a utility function, and that each item in the cache is characterized by its sorting value, assigned by the respective utility function. The main issues we wish to understand by conducting the simulations are:

- How good is the performance of the randomized algorithm according to realistic metrics like hit rate and latency? It is important to understand this because we have analyzed performance using the frequency of eviction from designated percentile bins as a metric. This metric has a strong positive correlation with realistic metrics but doesn't directly determine them.
- Our analysis in the previous sections assumes that documents retained in memory are not accessed between iterations. Clearly, in practice, this assumption can only hold with a high probability at best. We show that this is indeed the case and determine the probability that a sample retained in memory is accessed between iterations.
- How long do the best eviction candidates stay in the cache? If this time is very long (on average), then the randomized scheme

would waste space on "dead" items that can only be removed by a cache flush.

Of the three items listed above, the first is clearly the most important and the other two are of lesser interest. Accordingly, the bulk of the section is devoted to the first question and the other two are addressed towards the end.

A. Deterministic Replacement Algorithms

Using our randomization technique, we shall approximate the following two deterministic algorithms: LRU and GD-Hyb. GD-Hyb is a combination of the GD-Size [2] and the Hybrid [13] algorithms. LRU is chosen because it is the standard cache replacement algorithm. GD-Hyb is chosen to represent the class of new algorithms that base their document replacement policy not only on recentness of use, but also on the size of a document, the cost to fetch it from the server, and its frequency of use. We briefly describe the details of the deterministic algorithms mentioned above.

1. *LRU.* The utility function assigns to each document the most recent time that the document was accessed.
2. *Hybrid [13].* The utility function f assigns eviction values to documents according to the formula

$$f = (L + \frac{W_1}{B})F^{W_2}/S,$$

where L is an estimate of the latency for connecting with the corresponding server, B is an estimate of the bandwidth between the proxy cache and the corresponding server, F is the number of times the document has been requested since it entered the cache (frequency of use), S is the size of the document, and W_1, W_2 are weights[8]. Hybrid evicts the document with the smallest value of f.
3. *GD-Size [2].* Whenever there is a request for a document, the utility function f adds the reciprocal of the document's size to the currently minimum eviction value among all the documents in the cache, and assigns the result to the document. Thus, the eviction value for document i is given by

$$f_i = \min(f_j : j \text{ in cache}) + \frac{1}{S_i},$$

Note that the quantity $\min(f_j : j \text{ in cache})$ is increasing in time and it is used to take into account the recentness of a document. Indeed, since whenever a document is accessed its eviction value is increased by the currently minimum eviction value, the most recently used documents tent to have larger eviction values. GD-Size evicts the document with the smallest value of f.
4. *GD-Hyb* uses the utility function of Hybrid in place of the quantity $1/S$ in the utility function of GD-Size. Thus, its utility function is as follows:

$$\begin{aligned} f &= \min(f) + f', \text{ where} \\ f' &= (L + \frac{W_1}{B})F^{W_2}/S. \end{aligned}$$

We shall refer to the randomized versions of LRU and GD-Hyb as RLRU and RGD-Hyb respectively. Note that the RGD-Hyb algorithm uses the $\min(f)$ among the samples, and not the global $\min(f)$ among all documents in the cache.

[8] In the simulations we use the same weights as in [13].

So far we have described the utility functions of some deterministic replacement algorithms. Next, we comment on the implementation requirements of those schemes. Recall that the randomized algorithm requires no data structures to be implemented, irrespectively of which deterministic scheme it approximates.

LRU can be implemented with a linked list that maintains the order in which the cached documents were accessed so far. This is due to the "monotonicity" property of its utility function; whenever a document is accessed, it is the most recently used. Thus, it should be inserted at the bottom of the list and the least recently used document always resides at the top of the list. However, most algorithms, including those that have the best performance, lack the monotonicity property and they require to search all documents to find which to evict. To reduce computation overhead, they must use a priority queue to drop the search cost to $O(logK)$, where K is the number of documents in the cache. In particular, Hybrid, GD-Size, and GD-Hyb must use a priority queue.

The authors in [6] propose an algorithm called LRV (Lowest Relative Value). This algorithm uses a utility function that is based on statistical parameters collected by the server. By separating the cached documents into different queues according to the number of times they are accessed, or their relative size, and by taking into account within a queue only time locality, the algorithm maintains the monotonicity property of LRU *within* a queue. LRV evicts the best among the documents residing at the head of these queues. Thus, the scheme can be implemented with a constant number of linked lists, and finds an eviction candidate in constant time. However, its performance is inferior to algorithms like GD-Size [2]. Also, the cost of maintaining all these linked lists is still high.

The best cache replacement algorithm is in essence the one with the best utility function. In this paper we don't seek for the best utility function. Instead, we propose a low cost, high performance, robust algorithm that treats all the different utility functions in a unified way. We show that the randomized version of any scheme, regardless of the utility function it uses, can perform as well as the non-random scheme, without the need to maintain any data structures.

B. Web Traces

The traces we use are taken from Virginia University, Boston University, and National Laboratory for Applied Network Research (NLANR). In particular:

- The *Virginia* [12] trace consists of every URL request appearing on the Computer Science Department backbone of Virginia University with a client inside the department, naming any server in the world. The trace was taken for a 37 day period in September and October 1995 representing around 54000 requests. There are no latency data on that trace thus it can not be used to evaluate RGD-Hyb.
- *Boston* [3] traces consist of two sets. They record all HTTP requests originating from 32 workstations. The first was collected in January 1995 and consists of around 18000 requests. The second was collected in February 1995 and consists of around 110000 requests. Both contain latency data.

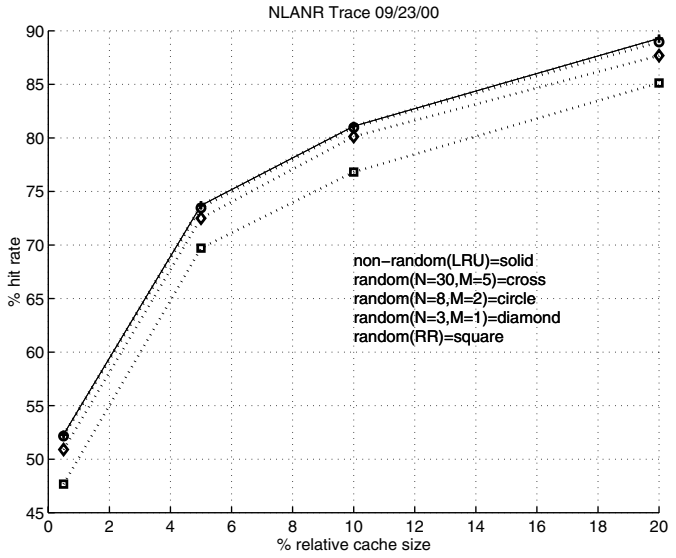

Fig. 6. Hit rate comparison between LRU and RLRU.

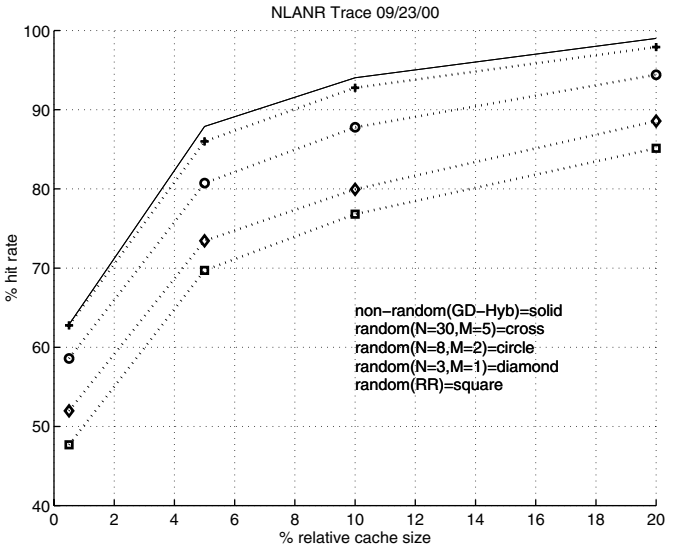

Fig. 7. Hit rate comparison between GD-Hyb and RGD-Hyb.

• The *NLANR* [14] traces consist of seven daily sets, with around 300000 requests each. The daily traces were recorded from the 22nd to 28th of September 2000[9]. All of them contain latency data.

We only simulate requests with a known reply size.

C. Results

The performance criteria used are three:
(i) the hit rate (HR), which is the fraction of client-requested URLs returned by the proxy cache,
(ii) the byte hit rate (BHR), which is the fraction of client requested bytes returned by the proxy cache, and
(iii) the latency reduction (LR), which is the reduction of the waiting time of the user from the time the request is made till the time the document is fetched to the terminal (download latency), over the sum of all download latencies.

For each trace, HR, BHR, and LR are calculated for a cache of infinite size. Then, they are calculated for a cache of size $0.5\%, 5\%, 10\%$, and 20% of the maximum size required to avoid any evictions. This size is around 500MB, 900MB, and 2GB for Virginia, Boston, and each daily NLANR trace respectively. All the traces give similar results. Since the NLANR traces consist of more requests, are more recent, and contain latency data, we only present simulation results from those traces.

Figure 6 and 7 present the ratio of HR of various schemes over the HR achieved by an infinite cache. The former figure compares LRU to RLRU, and the later GD-Hyb to RGD-Hyb. In Figure 6, RLRU nearly matches LRU for N and M as small as 8 and 2 respectively. In Figure 7, RGD-Hyb requires 30 samples and a memory of 5 to closely approximate GD-Hyb. The performance of GD-Hyb is superior to LRU. Indeed, GD-Hyb achieves around 100% of the infinite cache performance while LRU achieves below 90%. Note that RR's performance is 15% worse than GD-Hyb's.

[9]NLANR traces consist of daily traces from many sites; the traces we used are from the PA site.

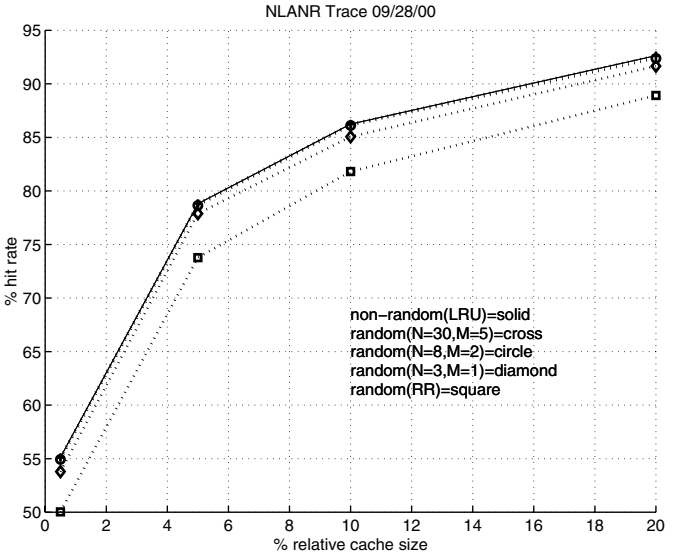

Fig. 8. Hit rate comparison between LRU and RLRU.

Similar results are obtained from all traces. As a second example, Figure 8 and 9 plot the HR achieved by LRU,LRLU and GD-Hyb, RGD-Hyb respectively, using another daily NLANR trace. Again RLRU nearly matches LRU for N and M as small as 8 and 2 respectively, and GD-Hyb requires 30 samples and a memory of 5 to closely approximate GD-Hyb.

Figure 10 and 11 present the ratio of BHR of various schemes over the BHR achieved by an infinite cache. The former figure compares LRU to RLRU, and the later GD-Hyb to RGD-Hyb. The randomized algorithm works well in respect to BHR, requiring N and M to be as low as 3 and 1.

Note that RGD-Hyb performs better than GD-Hyb for small cache sizes and more importantly, LRU performs as good as GD-Hyb. Actually, for some of the traces LRU performed slightly better than GD-Hyb. This somewhat unexpected result is caused because GD-Hyb makes relatively poor choices

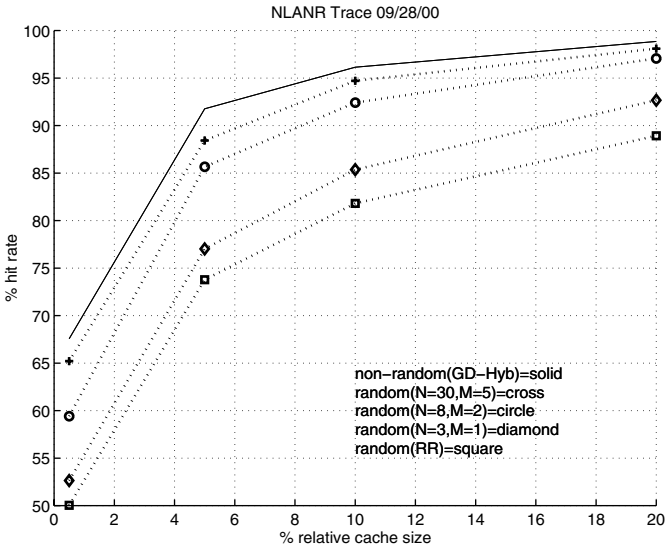

Fig. 9. Hit rate comparison between GD-Hyb and RGD-Hyb.

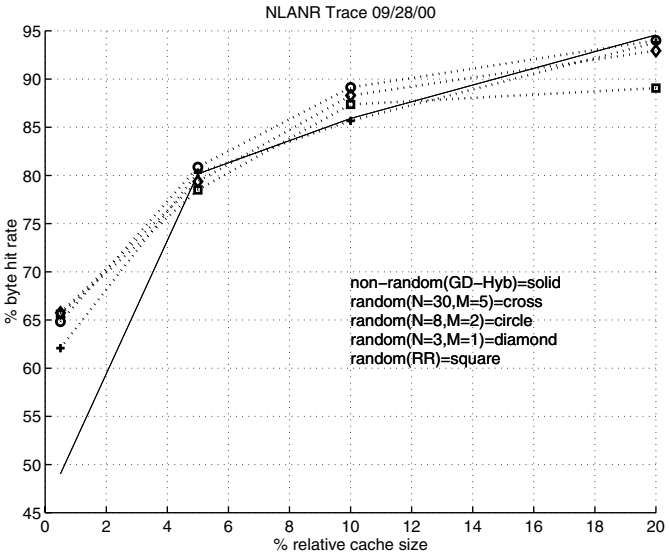

Fig. 11. Byte Hit rate comparison between GD-Hyb and RGD-Hyb.

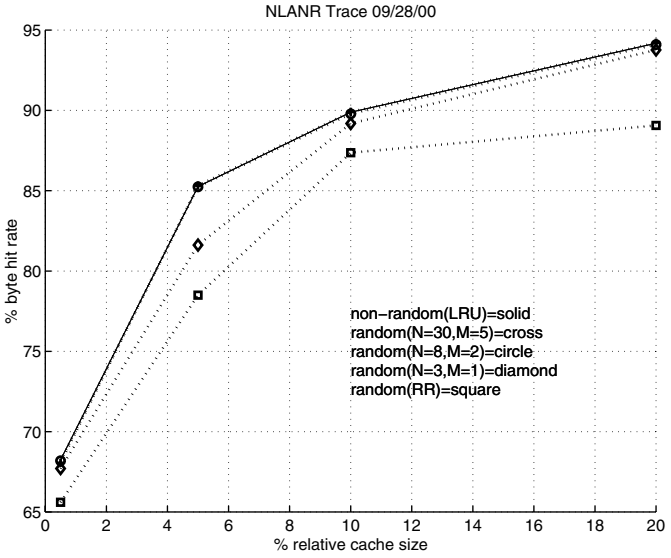

Fig. 10. Byte Hit rate comparison between LRU and RLRU.

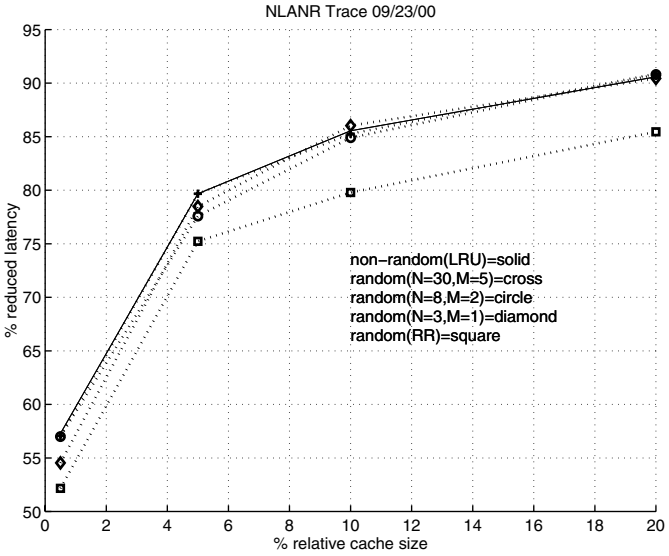

Fig. 12. Latency reduction comparison between LRU and RLRU.

in terms of BHR by design, since it has a strong bias against large size documents even when these documents are popular. This suboptimal performance of GD-Hyb is inherited from SIZE [12] and Hybrid [13] and could be removed by fine-tuning. All the three schemes trade in HR for BHR[10].

Figure 12 and 13 present the ratio of LR of various schemes over the LR achieved by an infinite cache. The former figure compares LRU to RLRU, and the later GD-Hyb to RGD-Hyb. In Figure 12, RLRU nearly matches LRU for N and M as small as 3 and 1 respectively. In Figure 13, it suffices for N and M to be equal to 8 and 2 respectively for RGD-Hyb to perform very well.

[10]Recently, an algorithm called GreedyDual* has been proposed [5], that achieves superior HR *and* BHR when compared to other web cache replacement policies.

From the figures above, it is evident that the randomized versions of the schemes can perform competitively with very small number of samples and memory. One would expect to require more samples and memory to get such good performance. However, since all the online cache replacement schemes rely on heuristics to predict future requests, it is not necessary to exactly mimic their behavior in order to achieve high performance. Instead, it usually suffices to evict a document that is within a reasonable distance from the least useful document.

There are two more issues to be addressed. First, we wish to estimate the probability that documents retained in memory are accessed between iterations. This event very much depends on the request patterns and is hard to analyze exactly. Instead, we use the simulations to estimate the probability of occurring. Thus, we change the eviction value of a document retained in

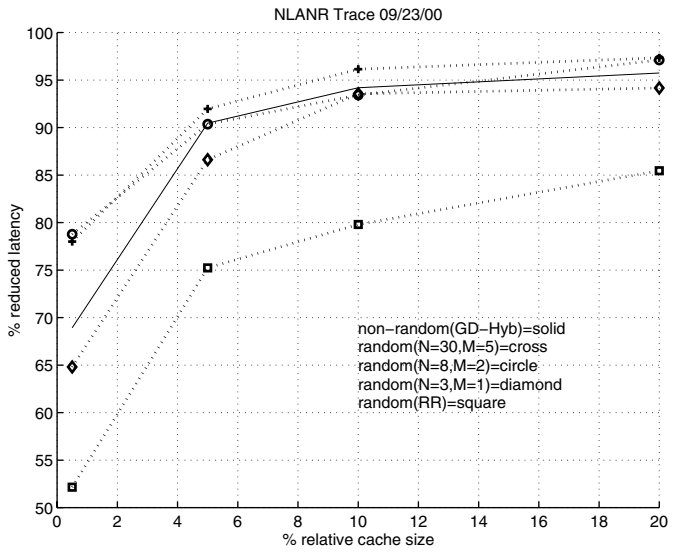

Fig. 13. Latency reduction comparison between GD-Hyb and RGD-Hyb.

memory whenever it is accessed between iterations, which deteriorates its value as an eviction candidate. Also, we don't obtain a new, potentially better, sample. Despite the above, the performance is not degraded. The reason is that our policy for retaining samples in memory deliberately chooses the best eviction candidates. Therefore, the probability that they are accessed is very small. In particular, it is less than 10^{-3} in our simulations.

Second, we wish to verify that the randomized versions of the schemes do not produce dead documents. Due to the sampling procedure, the number of sampling times that a document is not chosen follows a geometric distribution with parameter roughly equal to N over the total number of documents in the cache. This is around 1/100 in our simulations. Hence, the probability that the best ones are never chosen is zero, and the best ones are chosen once every 100 sampling times or so.

VII. Conclusions

In this work we have introduced a randomized algorithm for approximating any existing web-cache replacement scheme. We find that carrying a small amount of information regarding good samples from one iteration to the next, leads to a dramatic improvement in performance. By a judicious choice of parameters (the total number of samples, N, and the number of good samples, M, retained from one iteration to the next) we find that any replacement scheme can be approximated as closely as desired. Trace-driven simulations show that $N = 8$ and $M = 2$ suffice in practice.

High performance deterministic algorithms require a priority queue or multiple linked lists in order to reduce seek time. Further, most of these algorithms spend $O(\log K)$ time whenever there is an *access* to a document to keep the data structure updated. From an implementation point of view this is much more complex than avoiding the use of any data structure and just randomly sampling 6 documents and remembering 2 ($N = 8,\ M = 2$) whenever a document is to be *evicted*.

In general, our scheme can be used efficiently whenever there is a large population of objects from which the "best" is to be chosen according to some utility function.

VIII. Acknowledgments

We thank Dawson Engler for early conversations regarding cache replacement schemes, and A.J. Ganesh for fruitful discussions regarding the martingale argument.

References

[1] M. Abrams, C.R. Standbridge, G. Abdulla, S. Williams and E.A. Fox, "Caching Proxies: Limitations and Potentials", *WWW-4*, Boston, December, 1995.

[2] P. Cao and S. Irani, "Cost-Aware WWW Proxy Caching Algorithms", In proceedings of the *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, Dec. 1997.

[3] C.R. Cunba, A. Bestavros, M.E. Crovella, "Characteristics of WWW Client-based Traces", BU-CS-96-010, Boston University.

[4] R. Durrett, *Probability: Theory and Examples*, Duxbury Press, 2nd edition, 1996.

[5] S. Jin and A. Bestavros, "GreedyDual* Web Caching Algorithm: Exploiting the Two Sources of Temporal Locality in Web Request Streams", In Proceedings of the *5th International Web Caching and Content Delivery Workshop*, Lisbon, Portugal, May 2000.

[6] L. Rizzo and L. Vicisano, "Replacement Policies for a Proxy Cache", *IEEE/ACM Transactions On Networking*, Vol. 8, No. 2, April 2000.

[7] R. Motwani and P. Raghavan, *Randomized Algorithms*, Cambridge University Press, 1995.

[8] J. Norris, *Markov Chains*, Cambridge University Press, 1997.

[9] K. Psounis, B. Prabhakar, D. Engler, "A Randomized Cache Replacement Scheme Approximating LRU", *34th Annual Conference on Information Sciences and Systems*, March 15-17, Princeton University.

[10] K. Psounis, B. Prabhakar, "A Randomized Web-Cache Replacement Scheme", CSL Technical Report No: CSL-TR-00-805, Revision 1, December 2000, Stanford University.

[11] A. Silberschatz and P. Galvin, *Operating System Concepts*, Fifth Edition, Addison Wesley Longman, 1997.

[12] S. Williams, M. Abrams, C.R. Standbridge, G. Abdulla and E.A. Fox, "Removal Policies in Network Caches for World-Wide Web Documents", In Proceedings of the *ACM Sigcomm96*, August, 1996, Stanford University.

[13] R. Wooster and M. Abrams, "Proxy Caching that Estimates Edge Load Delays", In the *6th International World Wide Web Conference*, April 7-11, 1997, Santa Clara, CA.

[14] NLANR Cache Access Logs, ftp://ircache.nlanr.net/Traces/.