Replacement Problem in Web Caching

Seda Çakıroğlu, Erdal Arıkan
Bilkent University, Engineering Faculty,
Electrical and Electronics Engineering Dept.
06533 Ankara Turkey

Abstract

Caching has been recognized as an effective scheme for avoiding service bottleneck and reducing network traffic in World Wide Web. Our work focuses on the replacement problem in Web caching, which arises due to limited storage. We seek the best configuration for a fully-connected network of N caches. The problem is formulated as a discrete optimization problem. A number of low-complexity heuristics are studied to obtain approximate solutions. Performances are tested under fictitious probabilistic request sequences and access logs of real Web traffic. LFD (longest-forward-distance), the classical optimal off-line paging algorithm, is observed not to be optimal. Instead a window scheme should be used. Under an unchanging request pattern, a simple static placement algorithm achieves the maximum hit rates using the arrival probabilities. Otherwise, for quick adaptation to changing requests and for better worst-case performances a randomized algorithm should be chosen. We also give an analysis of Web data to propose best heuristics for its characteristics.

1 Introduction

The World Wide Web can be considered as a large, distributed information system that provides access to shared data objects. As stated in [9], Web traffic has become the fastest growing component among all kinds of Internet traffic. Although the Internet communication capacity increases 60% per year [10], the demand for bandwidth is likely to be more than the supply. The size of the static Web pages increases approximately 15% per month. Exponential growth of Web's size results in latency due to network congestion and server overloading. Web caching has been recognized as one of the effective schemes to solve these problems. Caching is a performance enhancement technique that has been used in computer systems for decades [7]. It is storing frequently-referred data in a *temporary* memory (cache)

where it can be accessed faster. However caching Web content is significantly different than caching in computer systems because Web object sizes and retrieval costs are nonuniform, and not all of the Web objects are cacheable. Also, in Web requests of the clients are met from a number of nodes which may be cooperating. There are many products implementing Web caching. Content Delivery Networks are contemporary examples of using caching in large scales, analysis on which can be found in the studies of Krishnamurty et al. [8]. For a survey on Web caching studies in general see [10].

We deal with the replacement problem in caching, deciding which pages to keep in the memory of limited size. The objective is finding a replacement algorithm to minimize the service cost in a given network of caches. A classification on the replacement algorithms that have been used in Web caching can be found in the study of Aggarwal et al. [4].

Section 2 describes the theoretical model used, and includes the formulation of caching as a discrete optimization problem. In Section 3 simulation results of some heuristic algorithms trying to approximate the optimal solution are given. Performances of different algorithms are presented and compared with each other in terms of hit rates and cost. The purpose is to find the ranking of the algorithms. Trying request sequences with different characteristics demonstrates how the request patterns affect this ranking. Next, a discussion of the results are made along with a characterization of Web data.

2 Model

We work on a fully-connected network of N caches which can be visualized in Figure 1. Each node i has one cache of capacity C_i to store data objects locally.

The data object universe U of interest consists of P pages and is identified by positive integers. Associated with each object $j \in U$ is a positive size w_j . The cost of retrieving a page j to the node of the user is a positive value dependent on the node i the request has arrived at and on the configuration of the caches at the time of arrival. The contents of the cache entries are denoted

Figure 1: Network Topology

by states.

Definition 1 The state of the caching network is represented by a two-dimensional array,

$$s = [s_{nm} ; 1 \leq n \leq N, 1 \leq m \leq C_n]. \quad (1)$$

The entry $s_{nm} = j$ indicates that the page j is at the m^{th} place in the n^{th} cache when the system is in state s . Let S be the universe of all possible states, i.e., all possible permutations of pages in all nodes. Then we denote the state of the system at time t by $s(t)$ or s_t .

We use a discrete-time system in which one and only one request is served in each time interval. The requests are denoted by ordered pairs $r(t)$, $t = 1, \dots, T$ such that,

$$r(t) = (i, j) \quad \text{if a request arrives at node } i \quad \text{for page } j \text{ at time } t. \quad (2)$$

Definition 2 A caching algorithm for a given cache network topology is a function determining the state $s(t)$ of the caches given the previous state $s(t-1)$ and the sequence of requests $\{r(t)\}$ such that,

$$s(t) = f(s(t-1), \{r(t)\}) \quad t = 1, \dots, T \quad (3)$$

The initial state $s(0)$ of the cache and the request sequence $\{r(t)\}$ are the inputs of the caching algorithm and the state sequence $\{s(t)\}_{t=1}^T$ is the output.

There are *off-line algorithms*, which are of the form $s(t) = f(s(t-1), \{r(t')\}_{t'=1}^T)$. If knowledge on the requests up to time t suffice, i.e., if the algorithm is of the form $s(t) = f(s(t-1), \{r(t')\}_{t'=1}^t)$ then it is called an *on-line algorithm*. Some of the algorithms studied in this study make decisions based only on the current request and current state such that, $s(t) = f(s(t-1), r(t))$. Therefore we call such algorithms *markovian algorithms*. There are also *randomized algorithms* for

which the state transition function is of the form $s(t) = f(s(t-1), r(t), X(t))$ where $X(t)$ is a random parameter.

Let $b_{ij}(s)$ denote the unit cost defined for retrieving one object j to node i when the system is in state s . An example of the cost model that will be used here is the following:

$$b_{ij}(s) = \begin{cases} 0 & \text{if } j = s_{im} \text{ for some } 1 \leq m \leq C_i \\ 1 & \text{if } j = s_{nm} \text{ for some } n \neq i \text{ and } 1 \leq m \leq C_n \\ 5 & \text{otherwise} \end{cases} \quad (4)$$

Let $B(s, r, s')$ be the cost of serving the request $r(t)$ when the system is in state $s(t-1) = s$ and changing the state to $s(t) = s'$, where $s(t)$ is determined by the caching algorithm employed. This 1-step cost is of the form

$$\begin{aligned} B(s(t-1), r(t), s(t)) &= B(s(t-1), r(t), f(s(t-1), \{r(t)\})) \\ &= B^f(s(t-1), r(t)) \end{aligned} \quad (5)$$

This cost can be decomposed as

$$\begin{aligned} B(s, r, s') &= b_r(s) + B_e(s, r, s') \\ &= b_r(s) + B_e^f(s, r) \end{aligned} \quad (6)$$

where $b_r(s)$ is the cost of serving r when in state s as defined in Eq. 4 and $B_e(s, r, s')$ is the extra cost apart from the retrieval cost of r , associated with changing state from s to s' . While $b_r(s)$ is independent of the policy f , $B_e(s, r, s')$ depends on f through s' . We assume $B^f(s, r)$ s are given as a part of the problem specification. The cost of serving a request sequence is defined next.

Definition 3 Given a request sequence $\{r(t)\}_{t=1}^T$ and an initial state $s(0) = s_0$, we define the *point-wise cost* of a caching policy f as the total cost of the state sequence formed by serving the requests.

$$B_T^f(s_0, \{r(t)\}) = \sum_{t=1}^T B^f(s_{t-1}, r_t) \quad (7)$$

We wish to find caching algorithms for which the cost given in Eq. 7 is as small as possible, so we define the point-wise optimality as

$$B_T(s_0, \{r(t)\}) = \min_f \left\{ \sum_{t=1}^T B^f(s_{t-1}, r_t) \right\}, \quad (8)$$

By definition, achieving this optimality criterion requires to know the whole sequence of requests. Therefore, it allows f to be an off-line algorithm. However in Web caching practice it is not possible to know the future requests, markovian algorithms should be preferred instead. Following, average-case cost will be defined which is more appropriate for comparing such algorithms.

For the rest of this work we assume $\{r(t)\} = r(1), r(2), \dots$ is an independent identically distributed (IID) sequence of requests. The probability of receiving the request

$r(t) = (i, j)$ is given by $p_r(i, j)$. The *average cost* of a time-invariant markovian algorithm under an IID request sequence is now given by

$$\begin{aligned}\overline{B_T^f(s_0)} &= \mathbf{E} \left[\sum_{t=1}^T B(s_{t-1}, r_t, s_t) \right] \\ &= \sum_{r_1, r_2, \dots, r_T} p_r(r_1) p_r(r_2) \dots p_r(r_T) \\ &\quad [B(s_0, r_1, s_1) + B(s_1, r_2, s_2) + \dots + B(s_{T-1}, r_T, s_T)] \\ &= \sum_{r_1} p_r(r_1) \left[B(s_0, r_1, s_1) + \overline{B_{T-1}^f(f(s_0, r_1))} \right] \quad (9)\end{aligned}$$

where $s_t = f(s_{t-1}, r_t)$ and $r_t = r(t)$ and $\mathbf{E}$ is expectation over request. The summation over r_i , $1 \leq i \leq T$, runs through all possible values of $r(i)$.

Given the request statistics $\{p_r\}$, the objective is to find an algorithm f which minimizes the T -step cost for the initial state $s(0) = s_0$ such that,

$$\overline{B_T(s_0)} = \min_f \{ \overline{B_T^f(s_0)} \}. \quad (10)$$

The solution of the minimization problem given in Eq. 10 can be found for a sequence of length T via stochastic dynamic programming. However, finding the best action at each step for each of the states requires too much computation. The number of possible states is exponential in P , N and C , which causes the computational complexity of the dynamic programming solution to be exponential in P , N , C , and T .

On the other hand we know that, the optimal policy should definitely decide not to change the state once the system reaches the optimal state. It should have $B_e = 0$ for minimizing T -step costs for all T . Let $s^* = \text{argmin}\{\overline{b(s)}\}$, then the optimal policy, f^* is such that,

$$f^*(s^*, r) = s^* \quad \text{for all } r, \quad (11)$$

i.e., s^* is a fixed-point of f^* . Recalling that Eq. 9 can be written as

$$\begin{aligned}\overline{B_T^f(s_0)} &= \sum_{r_1} p_r(r_1) [b_{r_1}(s_0) + B_e(s_0, r_1, s_1)] + \\ &\quad \sum_{r_1} p_r(r_1) \overline{B_{T-1}^f(f(s_0, r_1))} \\ &= \overline{b(s_0)} + \overline{B_e^f(s_0)} + \sum_{r_1} p_r(r_1) \overline{B_{T-1}^f(f(s_0, r_1))} \quad (12)\end{aligned}$$

$B_T^{f^*}(s^*)$ can be calculated as follows:

$$\overline{B_T^{f^*}(s^*)} = \overline{b(s^*)} + \sum_{r_1} p_r(r_1) \overline{B_{T-1}^{f^*}(f^*(s^*, r_1))} \quad (13)$$

where $\overline{B_e^{f^*}(s^*)} = 0$ and $\overline{B_1^{f^*}(s^*)} = \overline{b(s^*)}$. Then the above equation becomes

$$\overline{B_T^{f^*}(s^*)} = T \overline{b(s^*)}. \quad (14)$$

Theorem 1

$$\overline{B_T^f(s)} \geq \overline{B_T^{f^*}(s^*)} = T \overline{b(s^*)} \quad \text{for all } f, s \text{ and } T \geq 1. \quad (15)$$

Proof. For $T = 1$,

$$\overline{B_1^f(s)} = \overline{b(s)} \geq \overline{b(s^*)} = \overline{B_1^{f^*}(s^*)} \quad (16)$$

because $\overline{B_e^{f^*}(s^*)} = 0$. To use induction, assume

$$\overline{B_{T-1}^f(s)} \geq \overline{B_{T-1}^{f^*}(s^*)}. \quad (17)$$

Then from Eq. 12,

$$\begin{aligned}\overline{B_T^f(s)} &= \overline{b(s)} + \overline{B_e^f(s)} + \sum_{r_1} p_r(r_1) \overline{B_{T-1}^f(f(s, r_1))} \\ &\geq \overline{b(s^*)} + \sum_{r_1} p_r(r_1) \overline{B_{T-1}^{f^*}(f^*(s, r_1))} \\ &= T \overline{b(s^*)} \\ &= \overline{B_T^{f^*}(s^*)}. \quad (18)\end{aligned}$$

Under optimum policy, the system will change state until it reaches the optimum state and from then on it will stay there. If the request sequence is sufficiently long, the number of iterations the system needs until it converges to the optimum state may be ignored. Then the objective becomes just finding the state with minimum expected cost $\overline{b(s)}$. Due to the big state space, finding this is not a trivial matter. Instead some heuristics may be used to find an approximately optimum state and that state can be employed as the static configuration of the caches. The heuristic algorithms that arise from this concept are given under the name *static algorithms*.

Since the dynamic programming solution is computationally expensive, some approximations which minimize 1 or 2-step average costs are tried. They make their decisions with the constraint that $B_e^f(s, r) = 0$ for all r and s . In other words, no new page is inserted into the caches apart from the requested object and it can only be copied to the node the request has arrived. This reduces the number of states the current state can change into, thus the number of computations and comparisons.

3 Simulation Results

In the first set of experiments, the object universe includes 200 pages. We use $w_j = 1$ for all j , and $C_i = 10$ for all i . Different request sequences are generated, all according to some geometric distribution where each page is made popular at at least one of the nodes. The arrival probabilities are set according to a Zipf-like distribution, which is claimed to be the real world traffic's

Figure 2: Total hits under Bilkent Web Server trace

characteristic by Breslau et al. [5]. They showed that the i^{th} most popular document receives requests proportional to $1/i^\alpha$ for some α value in between 0.6 and 0.8. Thus we chose $\alpha = 0.6, 0.8, 1$, where nodes have different popularity rankings. We also generated a sequence with $\alpha = 0.8$ where the pages' arrival probabilities are the same at each node.

In the second set of experiments the same topology is considered, but actual Web traces retrieved from Bilkent University Web Server access logs in October, 2001 [1], and from the proxy traces of a Turkish ISP [2] in August, 2001 are used. The node at which the request arrives is decided according to the source IP addresses available in the traces.

We use commonly known heuristics such as FIFO (First In First Out), LRU (Least Recently Used), LFU (Least Frequently Used), and RAND (Random Removal), and their variations. SWAP is similar to LRU but it evaluates the popularity of the pages more gradually. MARK is a combination of LRU and RAND. It is derived from the heuristic *Marking Algorithm* presented in [6] for k-server paging problem. It chooses a page to remove randomly among the not recently requested pages. On-line variations of LFU, which use some additional and global-in-time counters for improvements are also tried. LFU-G holds counters of all requests, starting from the beginning. LFU-D algorithms allow duplication of objects among nodes by keeping track of page retrievals from neighboring nodes. LFU-DF uses frequency information of such retrievals while LFU-DT uses the time of occurrence. All of them use two-parted cache structure, one for recently cached pages, and other for pages with longer popularity. We also test dynamic programming approximations DP-1 and DP-2, and static algorithms SLS, STS, and SDS. SLS decides

for a static configuration of the caches by sorting the arrival probabilities at each node to maximize hits received locally. STS sorts the arrival probabilities of page requests coming to any of the nodes in order to achieve high total hits. SDS implements a double sort to make use of the former two algorithms. The drawback of these heuristics is that, their run-time complexities are higher than that of the markovian algorithms. Under the real world data, we also test the performances of the off-line algorithm LFD and its variants. LFD-A counts the future requests to pages without a distinction between the nodes of arrival. LFD-L considers only local requests. LFD-S tries to increase local hit rates by allowing duplication among nodes. In the heuristics named WIN, instead of the time when the next request for a page arrives, the number of references in a window of future requests are used to determine the value of pages. In WIN-A, the nodes from which the requests arrive does not make difference. WIN-L favors hits from the local nodes, and WIN-S makes local copies of pages that are retrieved from neighboring nodes. LFD-A and WIN-A are centralized algorithms, on the other hand LFD-L and WIN-L are distributed.

We compare local hit rates achieved by serving the requests directly from local nodes. Total hit rates achieved as a result of requests arriving at any of the nodes, and costs of serving the requests are also compared. Tables 1 to 3 summarize the results obtained in the 1st set of experiments, Table 4 summarize the 2nd set of experiments. The Figures 2, and 3 present the total hits achieved under real Web traffic, where subplots *a* show markovian, *b* on-line, *d* off-line algorithms, and *c* display static algorithms and dynamic programming approximations. The maximum hits that can be achieved are also given as asymptotic limits.

Figure 3: Total hits under a Web proxy trace

Figure 4: Characteristics of Web data

Algorithms	Zipf			
	$\alpha = 1$	$\alpha = 0.8$	$\alpha = 0.8$ all same	$\alpha = 0.6$
RAND	0.5243	0.4275	0.0694	0.3491
FIFO	0.5204	0.4263	0.0684	0.3518
LRU	0.5590	0.4476	0.0746	0.3589
SWAP	0.5355	0.4349	0.0709	0.3534
MARK	0.4930	0.3927	0.0758	0.3095
LFU	0.5799	0.4823	0.0791	0.3742
LFU-G	0.6589	0.5624	0.0805	0.4539
LFU-D	0.5973	0.4823	0.2326	0.3630
LFU-DF	0.6189	0.5268	0.2676	0.4276
LFU-DT	0.5906	0.4745	0.2418	0.3674
SLS	0.6805	0.5865	0.3570	0.4831
STS	0.6042	0.4613	0.0854	0.3260
SDS	0.6330	0.4978	0.0854	0.3593
DP1	0.6740	0.5803	0.3113	0.4770
DP2	0.6307	0.5052	0.0937	0.3820

Table 1: Local hit rates

4 Discussion

It is observed that performances strongly depend on the request sequence characteristic. If the network traffic can be categorized, then it is possible to find the most suitable algorithm. In the 1st set of experiments, it is possible to maximize local or total hit rates by employing static algorithms. Such probabilistic distributions also allow frequency dependent algorithms like LFU to outperform others. As α decreases, i.e., as the arrival probabilities become closer, costs increase, hit rates de-

crease, and duplication algorithms perform worse. Comparing the results of the two set of experiments we observe that under the request sequence generated with α equal to 0.8, each node having the same request pattern, the results are similar to that of the Web server access logs. The static algorithms, and LFU variants achieve good result under these sequences, which shows that the probabilities of the pages stay unchanging in time. Low local hit rates are increased by duplication.

When α equals to 0.6, the generated request sequence resembles the proxy logs most. Under the proxy traces, off-line algorithms achieve high hits, WIN algorithms better than LFD variants. On-line algorithms are disappointing, the big object space and the lack of distinction between pages avoids them to have high hit rates. Moreover, their implementations are more complex, which discourages their use. However, the markovian algorithms achieve the most promising results. The best one turns out to be MARK as a result of its centralized and randomized nature. Dynamic programming approximations also have good performances, however their run-time is much higher.

In Figure 4, the number of references and the ranking of the documents are plotted on a log-log scale for the server and proxy traces used. The α values obtained are 1.1319 for the Web server, and 0.6498 for the proxy. It is no surprise to have small α in the proxy traces, where the object space is large, and the demand characteristics have more variations. Caching is effective for popular documents, but not on the heavy tail which comprises a significant factor. Web's effectiveness increases only logarithmically as increased cache sizes. But in the Web server access logs, number of page references are observed to have steeper dependence on page rankings, the objects within a Web server are more

Algorithms	Zipf			
	$\alpha = 1$	$\alpha = 0.8$	$\alpha = 0.8$ all same	$\alpha = 0.6$
RAND	0.6147	0.5600	0.6946	0.5375
FIFO	0.6094	0.5548	0.6925	0.5373
LRU	0.6445	0.5788	0.7349	0.5514
SWAP	0.6220	0.5646	0.7154	0.5421
MARK	0.6392	0.5765	0.7327	0.5494
LFU	0.6684	0.6266	0.7615	0.5790
LFU-G	0.6873	0.6080	0.7898	0.5129
LFU-D	0.6811	0.6092	0.6655	0.5541
LFU-DF	0.6766	0.6088	0.5555	0.5573
LFU-DT	0.6514	0.5722	0.5865	0.5110
SLS	0.6805	0.5865	0.3570	0.4831
STS	0.7287	0.6757	0.8170	0.6638
SDS	0.7297	0.6749	0.8170	0.6621
DP1	0.6764	0.5838	0.6499	0.4806
DP2	0.7255	0.6726	0.8219	0.6561

Table 2: Total hit rates

Algorithms	Zipf			
	$\alpha = 1$	$\alpha = 0.8$	$\alpha = 0.8$ all same	$\alpha = 0.6$
RAND	2.0170	2.3324	2.1521	2.5008
FIFO	2.0421	2.3544	2.1615	2.4991
LRU	1.8628	2.2373	1.9857	2.4354
SWAP	1.9765	2.3066	2.0674	2.4783
MARK	1.9502	2.3112	1.9933	2.4928
LFU	1.7463	2.0114	1.8749	2.3099
LFU-G	1.5920	2.0343	1.7601	2.4946
LFU-D	1.6782	2.0810	2.1053	2.4205
LFU-DF	1.6745	2.0379	2.5105	2.3433
LFU-DT	1.8039	2.2366	2.4121	2.5885
SLS	1.5973	2.0674	3.2152	2.5846
STS	1.4810	1.8358	1.6467	2.0189
SDS	1.4482	1.8017	1.6467	1.9921
DP1	1.6266	2.0847	2.0892	2.6004
DP2	1.4674	1.8046	1.6188	1.9937

Table 3: Costs per request

densely concentrated on the popular ones.

5 Conclusion

We formulated the caching problem as a discrete optimization problem, whose solution is exponential in N , C , and P . For the model used here the optimal solution, even though exists, can not be found except for small scale problems. Instead, some heuristics are tried to approximate it. For real world data, our heuristics achieved good results according to the remark made by Abrams et al. [3] that no caching system can achieve more than 40 – 50% hit ratio. We observed that the traffic on a proxy server and a Web server can be successively simulated by Zipf-like distribution. It is shown that LFD is not optimal. Rather, algorithms implementing a sliding window scheme should be used since the requests may arrive in bursts.

The resulting design tips can be summarized as follows. If the request sequence has a probabilistic character, it is good to make decisions based on frequency of request arrivals. If this distribution is stationary, employing global counters may improve performance. However, these increase convergence times. Under such requests, it may be good to use a 2-parted cache structure or optional caching. These avoid oscillations in system's state due to possible temporal variances in input sequence. If the request arrivals are changing in time as in the case of proxy traces, it is better to make decisions based on the time of references. In real world, the traffic characteristics may be changing too fast or even randomly. Employing small memory heuristics, and algorithms that have some level of randomness helps. MARK turns out

Algorithms	Local Hit		Total Hit		Cost	
	Server	Proxy	Server	Proxy	Server	Proxy
RAND	0.0505	0.1452	0.4520	0.2354	3.1414	3.9131
FIFO	0.0476	0.1506	0.4540	0.2403	3.1363	3.8883
LRU	0.0526	0.1503	0.4949	0.2420	2.9679	3.8817
SWAP	0.0469	0.1649	0.4473	0.2412	3.1637	3.8705
MARK	0.0523	0.0911	0.5073	0.3181	2.9184	3.6367
LFU	0.0614	0.0833	0.5780	0.2046	2.6267	4.0981
LFU-G	0.0653	0.0921	0.6099	0.2226	2.4950	4.0175
LFU-D	0.1886	0.1645	0.5260	0.2034	2.7075	4.0220
LFU-DF	0.0756	0.0431	0.4917	0.1014	2.9578	4.5514
LFU-DT	0.0547	0.0027	0.5304	0.0499	2.8235	4.7977
SLS	0.2965	0.1843	0.3886	0.1993	3.1491	4.0186
STS	0.0835	0.1028	0.6273	0.2366	2.4071	3.9508
SDS	0.0993	0.1596	0.6241	0.2319	2.4041	3.9127
DP1	0.2435	0.2325	0.5309	0.2662	2.6331	3.7026
LFD-A	0.0880	0.0434	0.5946	0.1238	2.5336	4.4614
LFD-L	0.1042	0.0373	0.5693	0.0946	2.6185	4.5844
LFD-S	0.1883	0.0413	0.5185	0.1005	2.7377	4.5568
WIN-A	0.1135	0.1641	0.6230	0.3184	2.3944	3.5623
WIN-L	0.1168	0.1687	0.6222	0.3186	2.3942	3.5567
WIN-S	0.1473	0.2390	0.6124	0.3173	2.4031	3.4919

Table 4: Real data

to be the best heuristic. If the objective is maximizing the local hits, then duplication may be used. On the other hand, a centralized algorithm has better performances if the total hit rate is concerned.

Having such results, next topic to be studied is the cache placement problem, deciding for the places of a limited number of caches for a given request distribution in a mesh network to minimize traffic or retrieval times.

References

- [1] Bilkent university web server access logs. *Contact Can Ugur Ayfer from Bilkent Computer Center.*
- [2] Proxy trace of a turkish isp. *Personal communication.*
- [3] M. Abrams, C. R. Stranridge, G. Abdulla, S. Williams, and E. A. Fox. Caching proxies: Limitations and potentials. *Proceedings of 4th International World-Wide Web Conference*, December 1995.
- [4] C. Aggarwal, J. L. Wolf, and P. S. Yu. Caching on the www. *IEEE Transactions on Knowledge and Data Engineering*, 11, 1999.
- [5] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker. Web caching and zipf-like distributions: Evidence and implications. *Proceedings IEEE Infocom*, March 1999.
- [6] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young. Competitive paging algorithms. *Technical Report CMU-CS-88-196*, Carnegie-Mellon University, 1988.
- [7] S. H. Khayat and J. R. C. Jr. Optimal solution of off-line and on-line generalized caching. *Washington University*, 1997.
- [8] B. Krishnamurthy and J. Rexford. *Web Protocols and Practice: HTTP1.1, Networking Protocols, Caching and Traffic Measurement*. Addison Wesley Longman, Reading, 2001.
- [9] A. L. Montgomery and C. Faloutsos. Identifying web browsing trends and patterns. *Computer Magazine, IEEE*, July 2001.
- [10] J. Wang. A survey of web caching schemes for the internet. *ACM Computer Communication Review*, October 1999.