Exploiting Web Log Mining for Web Cache Enhancement

Alexandros Nanopoulos, Dimitrios Katsaros, and Yannis Manolopoulos

Department of Informatics, Aristotle University
Thessaloniki 54006, Greece
{alex,dimitris,manolopo}@delab.csd.auth.gr

Abstract. Improving the performance of the Web is a crucial requirement, since its popularity resulted in a large increase in the user perceived latency. In this paper, we describe a Web caching scheme that capitalizes on prefetching. Prefetching refers to the mechanism of deducing forthcoming page accesses of a client, based on access log information. Web log mining methods are exploited to provide effective prediction of Web-user accesses. The proposed scheme achieves a coordination between the two techniques (i.e., caching and prefetching). The prefetched documents are accommodated in a dedicated part of the cache, to avoid the drawback of incorrect replacement of requested documents. The requirements of the Web are taken into account, compared to the existing schemes for buffer management in database and operating systems. Experimental results indicate the superiority of the proposed method compared to the previous ones, in terms of improvement in cache performance.

Keywords: Prediction, Web Log Mining, Web Caching, Prefetching, Association rules.

1 Introduction

The problem of modelling and predicting a user's accesses on a Web-site has attracted a lot of research interest. It has been used [20] to improve the Web performance through caching [2,12] and prefetching [34,22,35,29,39,40], recommend related pages [19,38], improve search engines [11] and personalize browsing in a Web site [39].

Nowadays, the improvement of Web performance is a very significant requirement. Since the Web's popularity resulted in heavy traffic in the Internet, the net effect of this growth was a significant increase in the user perceived latency. Potential sources of latency are the Web servers' heavy load, network congestion, low bandwidth, bandwidth underutilization and propagation delay.

The caching of Web documents at various points in the network (client, proxy, server [2,12]) has been developed to reduce latency. Caching capitalizes on the temporal locality. Effective client and proxy caches reduce the client perceived latency, the server load and the number of travelling packets, thus increase the available bandwidth. Several caching policies have been proposed during the previous years, especially for proxy servers [2,12]. Nevertheless, there exist cases where the benefits reaped due to caching can be limited [28], e.g., when Web resources tend to change very frequently, resources cannot be cached (dynamically generated Web documents), they contain cookies (this issue matters only caching proxies), or when request streams do not exhibit high temporal locality. The negative effects of the first problem can be partially alleviated by employing some, costly though, cache consistency mechanism. The second problem could be addressed by enhancing the cache with some of the respective server's query processing capabilities, so as to perform the necessary processing on data [13]. The third and fourth problems seem that cannot be tackled by caching at all.

Web prefetching is the process of deducing client's future requests for Web documents and getting that documents into the cache, in the background, before an explicit request is made for them. Prefetching capitalizes on the spatial locality present in request streams, that is, correlated references for different documents, and exploits the client's idle time, i.e., the time between successive requests. The main advantages of employing prefetching is that it prevents bandwidth underutilization and hides part of the latency. However, an over-aggressive scheme may cause excessive network traffic. Additionally, without a carefully designed prefetching scheme, several transferred documents may not be used by the client at all, thus they waste bandwidth. We focus on predictive prefetching [18], since other categories, like informed prefetching [36], seem inapplicable due to the client-server paradigm of computing the Web implements and its hypertextual nature (i.e., a user in most cases does not know in advance his/her future document requests).

Web prefetching acts complementary to caching, it can significantly improve cache performance and reduce the user-perceived latency [34]. However, there are cases where a non-effective prefetching algorithm, presenting the aforementioned drawbacks, can impact cache performance [25]. For instance, if the accuracy of the prefetching algorithm is low, then several useful documents in the cache may be evicted by prefetched documents that are not going to be referenced. Therefore, there exists a requirement for both

– accurate prefetching algorithms, and
– caching schemes that will coordinate with prefetching.

In this paper we describe a scheme for: (a) Effective prefetching, which exploits Web log mining, it is not affected by factors like noise (i.e., random document requests) and high-order dependencies among document requests and thus, can significantly improve cache performance. (b) Coordination of caching and prefetching, by storing in the cache the prefetched documents separately from those which have been explicitly requested. The latter approach is based on the one of [25], which dedicates part of the cache to separately accommodate prefetched documents. However the scheme in [25] was designed for database disk buffer management. Therefore, it does not address the significantly different requirements encountered in the case of Web caching. Experimental results indicate that the proposed scheme outperforms existing ones in terms of improvement in cache performance.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes the prefetching algorithm, whereas in Section 4 the caching scheme is presented. Section 5 provides the experimental results and finally, Section 6 contains the conclusions.

2 Related Work

Research on predictive Web prefetching has involved the significant issue of log file processing and the determination of user transactions (sessions) from it [17,14]. However, the most important factor in Web prefetching is the prediction algorithm. For the purpose of prediction, most of the Web prefetching schemes rely on existing algorithms from the context of file systems. This approach neglects issues that arise in the case of Web, and stem from both the contents of Web documents in a site (which induce dependencies to their references) and the site's structure, i.e., the links among documents (which affect user's navigation).

The scheme described in [34] uses a prefetching algorithm proposed in the context of file systems. It constructs a data structure, called the *Dependency Graph* (*DG*), which maintains the pattern of access to different documents stored at the server. As described above, the choice of forthcoming pages can depend, in general, on a number of previously visited pages [20]. *DG* considers only first order dependencies. Thus, if several previous visits have to be considered (i.e., high-order dependencies), *DG* does not take them into account. The work described in [9] uses essentially the approach of dependency graph, but it makes predictions by computing the transitive closure of this graph. This method was tested and did not show significantly better results compared to the simple dependency graph.

The scheme described in [35,22] also uses a prefetching algorithm from the context of file systems [18]. It is based on the notion of an m-order *Prediction-by-Partial-Match* (*PPM*) predictor. An m-order PPM predictor maintains Markov predictors of order j, for all $1 \leq j \leq m$. This scheme is also called *All-*m^{th}*-Order Markov model* [20]. For a complete description of the scheme see [18,35,22]. During a session, although a user may navigate according to a pattern, s/he may also randomly navigate to pages that do not belong to any pattern (and can be modelled as noise). Hence, a session can both contain documents belonging to patterns and others that do not, and these documents are interleaved. However, PPM considers only subsequences of consecutive documents inside sessions, thus it is affected by the existence of noise[1]. Moreover, PPM uses a constant maximum value for the order. However, no method for the determination of this value is provided in [35,22]. A choice of a small maximum may have a similar disadvantage as in the case of *DG*, whereas a choice of a large maximum may lead to unnecessary computational cost, due to maintenance of a large number of rules.

[1] The use of thresholds for statistical significance, e.g., support values [3], does not address the intervening of random accesses within the patterns.

Recently, several algorithms have been proposed for mining patterns from Web logs [14,8,17,37,32]. The Web prefetching strategy proposed in [29] develops a specialized association rule mining algorithm to discover the prefetched documents. It discovers dependencies between pairs of documents (association rules with one item in the head and one item in the body). However, the scheme in [29], similar to *DG*, considers only first order dependencies and, similar to PPM, it considers only consecutive subsequences within sessions. Therefore, it presents the deficiencies of both approaches, as they are described above. The improvement of the efficiency of PPM is examined in [20], based on three pruning criteria. These criteria are used in a post-processing step, on the set of discovered rules, and can be applied to any prefetching scheme, thus they are orthogonal issues to the subject examined in this paper. Finally, two variations of the PPM prefetcher are described in [39,40]. The first one is a subset of the PPM whereas in the second one the selection of prefetching rules to activate is determined by "weights" assigned on them.

Web caching has received significant attention and several new algorithms were proposed, ranging from extensions to traditional policies (like *LRU*, *LFU*, etc.) to *key-based policies* [1] and more sophisticated *function-based* policies, such as *GD-Size* [12], *PSS* [2]. Moreover, significant results regarding optimal on-line and off-line caching policies for the Web were presented in [24].

Regarding the coordination of caching and prefetching, Jeon and Noh [25] presented the *W^2R* algorithm. Motivated by the *2Q* algorithm [26], W^2R divides the available cache space into two partitions, called *Weighing Room* and *Waiting Room*. Prefetched documents initially enter the Waiting Room, before becoming "normal" cached documents in the Weighing Room. However, W^2R was designed for database disk buffer management. It uses the *One Block Lookahead* (*OBL*) prefetching algorithm, which prefetches only one page each time. Moreover, all pages are of the same size. Web caching presents significantly different requirements, since different prefetching algorithms than the simple *OBL* are used (several documents are allowed to be prefetched each time and thus we must prioritize among them), and documents of different sizes have to be accommodated in the cache.

3 Prefetching Algorithm

As described in Section 2, none of the existing prefetching algorithms addresses at the same time both the factors of noise and high-order dependencies, that may exist within transactions (i.e., user sessions). In this section we describe an algorithm that addresses all the aforementioned factors. It uses the history of user accesses, maintained in the Web server's log file, to derive rules. Since the rules, which are appropriate for the prefetching, should be based on the navigation behaviour of the client[2] (expressed as the process of visiting links) we describe a pruning criterion that is based on the site structure. This pruning

[2] These rules are not appropriate for the identification of other types of usage patterns (see also Section 3.2).

can significantly reduce the computational overhead. On the other hand, it is not adequate for other types of applications, such as recommendation systems.

3.1 Prefetching Based on Rules

Association rules [3] consider several orders of dependencies among items, and the maximum order is derived from the data, i.e., it does not have to be an arbitrary constant value as in PPM. For the support counting procedure, a transaction T supports sequences that do not necessarily contain consecutive documents in T. Thus, this procedure can effectively bypass random document visits (i.e., noise) that may exist within transactions. However, the ordering of documents inside a transaction is important for the purpose of prefetching but it is ignored by association rules mining algorithms [3].

Therefore, a different definition of the candidate generation procedure and the *containment* criterion, for the support counting procedure, are required. At the k-th phase, the candidates are derived from the self-join $L_{k-1} \bowtie L_{k-1}$ [3]. To take the ordering of documents into account, the joining is done as follows. Let two access sequences be $S_1 = \langle p_1, \ldots, p_{k-1} \rangle$ and $S_2 = \langle q_1, \ldots, q_{k-1} \rangle$, both in L_{k-1}. If $p_1 = q_1, \ldots, p_{k-2} = q_{k-2}$, then they are combined to form two candidate sequences, which are: $c_1 = \langle p_1, \ldots, p_{k-2}, p_{k-1}, q_{k-1} \rangle$ and $c_2 = \langle p_1, \ldots, p_{k-2}, q_{k-1}, p_{k-1} \rangle$ (i.e., c_1 and c_2 are not considered as identical, as in [3]). For instance, sequences $\langle A, B, C \rangle$ and $\langle A, B, D \rangle$ are joined to produce $\langle A, B, C, D \rangle$ and $\langle A, B, D, C \rangle$. The *containment* criterion is defined as follows:

Definition 1 *If* $T = \langle p_1, \ldots, p_n \rangle$ *is a transaction, an access sequence* $S = \langle p'_1, \ldots, p'_m \rangle$ *is contained by* T *iff:*

– *there exist integers* $1 \leq i_1 < \ldots < i_m \leq n$ *such that* $p'_k = p_{i_k}$*, for all* k*, where* $1 \leq k \leq m$*.* □

A sequence, S, of documents contained in a transaction, T, with respect to Definition 1 is called a *subsequence* of T and the containment is denoted as $S \preceq T$.

Based on a set of rules discovered with the approach described above (along with the corresponding confidence values), prefetching is performed with the procedure depicted in Figure 1. In this algorithm, R denotes the current request stream formed by the user and M is the maximum number of prefetched documents (user parameter). Also, we use an upper limit, called *maxSize*, in the size of each prefetched document, since it is not desired to transfer very large documents to avoid waste of bandwidth in case of an incorrect prediction.

Evidently, the fact that the ordering of documents within transactions is preserved during the discovery of rules, impacts the complexity of candidate generation and support counting procedures. For this reason, we present in the following section a pruning criterion according to the site structure, that reduces the overhead.

Finally, it has to be mentioned that, differently from [3], the approach in [14] takes into account the ordering within access sequences. However, similar to

```
**Procedure** Prefetch(**Array** $R$, **int** $M$, **float** maxSize)
//$h, b$ are sequences of document ids
**begin**
1.  $prefetchSeq = \emptyset$
2.  **foreach** rule $h \Rightarrow b$ such that $h \preceq R$
3.      **foreach** $d \in b$ such that $d$.size < maxSize
4.          $prefetchSeq = prefetchSeq \bigcup d$
5.      **endfor**
6.  **endfor**
7.  sort documents in $prefetchSeq$ in decreasing order of the confidence
    of the corresponding rule and keep the first $M$ ones.
8.  **return** $prefetchSeq$
**end**
```

Fig. 1. Prefetching algorithm.

PPM algorithm [35], it considers only subsequences with consecutive accesses within transactions. The work described in [4,30] examines the problem of mining sequential patterns which consider ordering, as well. However, the algorithms in [4,30] do not take into account that user navigation is performed in a site which has a structure determined by its linkage. Therefore, they do not address the problem of dramatic increase in the number of candidates. Moreover, the work in [30] seeks patterns in a single large sequence of events using a sliding window over this sequence and thus it does not consider user sessions as is our case. The scheme proposed in [23] uses a scheme that is called mining of path fragments. It is based on discovering patterns containing regular expressions with the $*$ wild-card between accesses of a sequence. Although the use of wild-cards presents differences in a semantic level (it may distinguishes the sequences that explicitly do not contain consecutive accesses), for the purpose of Web-prefetching, the use of Definition 1 assures the addressing of noise within transactions without the need for wild-cards. Moreover, in the case of path fragments, the candidate-trie [3] should store, additionally to ordinary candidates, the ones containing wild-cards. Consequently, a significant space and time overhead (since the wild-cards may appear in a number of combinations that grows rapidly with the size of candidates). However, [23] does not present any method for the support counting phase to address the above issues, and no experimental results are provided to examine its performance.

3.2 Pruning Criterion

For the purposes of prefetching, we focus on the paradigm of *traversal patterns* [14], which will be used for prediction, not on *usage patterns*. Based on the assumption that navigation is performed by following the hypertext links, the traversal patterns have to reflect the way navigation is performed guided by the site structure. Thus, we can apply pruning according to the structure of the site. This is based on the approach in [32,33]. The work in [23] also mentions the possibility of taking into account the links between site documents but, differently from [32,33], it focuses only on the second phase of the algorithm. The pruning based on the structure of the site results in a significant reduction in the number of candidates. Without pruning, due to the consideration of ordering a large number of candidates would have been generated.

For instance, let three documents A, B and C for which the following linkage exists: $A \rightarrow B$ and $A \rightarrow C$. Assume a user-traversal $T = \langle A, B, A, C \rangle$, and that the second request for A is due to the 'back-button', which will be probably serviced by the client-cache. Thus, the server encounters the traversal $T' = \langle A, B, C \rangle$. T' may constitute a useful pattern for the description of site-usage, since it indicates a correlation between the usage of documents A, B and C. However, the objective of prefetching is to predict the forthcoming visits within the user's traversal. Based on the assumption of navigation through the hypertext links, let the corresponding traversal patterns will $\langle A, B \rangle$ and $\langle A, C \rangle$. Having visited document A, the prefetching of B and C can take place based on these patterns. Thus, along the lines of [14][3], we do not focus on usage patterns that indicate general correlation between document usage, as the correlation between the usage of A, B and C. In contrast, we are interested in the identifying the traversal patterns, like $\langle A, B \rangle$ and $\langle A, C \rangle$, in order to predict the forthcoming visits that will be done at A by following the corresponding links. Evidently, from the aspect of traversal patterns, the correlation between B and C exists only due to their linkage with A, differently than the usage pattern of T' that is based on the paradigm of association rules among basket data [3]. Thus, at A only the two aforementioned patterns will be required in order to prefetch B and C.

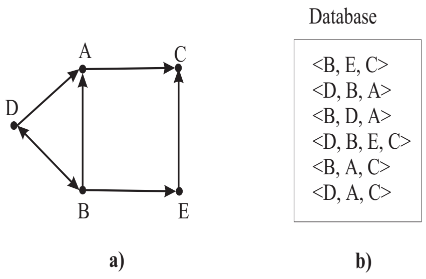
According to the proposed pruning criterion, an access sequence, and thus a candidate, has to correspond to a path in this graph. The candidate generation procedure and the apriori-pruning criterion [3] have to be modified appropriately, and the corresponding procedure is depicted in Figure 2 (for the generation of candidates at phase $k + 1$). L_k denotes the set of large sequences (i.e., frequent) at phase k and G the graph, which corresponds to the site structure.

Example 1 Let a site with the structure depicted in Figure 3a and a collection of transactions depicted in Figure 3b. Candidate $\langle B, E, C \rangle$ corresponds to a path in the graph. On the other hand, candidate $\langle B, C, E \rangle$ does not, thus it can be pruned. The reason of pruning the later candidate is that no user transaction will contain $\langle B, C, E \rangle$, since there are no links to produce such an access sequence that will contain (according to Definition 1) this candidate and increase its support. For instance, candidate $\langle B, E, C \rangle$ will have support equal to two (contained in first and fourth transaction), whereas candidate $\langle B, C, E \rangle$ is not supported by any transaction. The same applies for candidates of length equal to two. Among candidates $\langle A, B \rangle$ and $\langle B, A \rangle$, the former can be pruned. Evidently, $\langle B, A \rangle$ is contained in three transactions, whereas $\langle A, B \rangle$ in none. Containment is tested

[3] Please also notice that similar to [14] we consider maximum forward traversals, which discard backward movements.

```
**Procedure** GenCandidates($L_k$, $G$)
**begin**
1.  $candidate\text{-}trie = \emptyset$
2.  **foreach** $L = \langle \ell_1, \ldots, \ell_k \rangle$, $L \in L_k$
3.      $N^+(\ell_k) = \{v | \exists \text{ arc } \ell_k \rightarrow v \in G\}$
4.      **foreach** $v \in N^+(\ell_k)$
5.          /* apply the modified apriori-pruning */
6.          **if** $v \notin L$ **and** $L' = \langle \ell_2, \ldots, \ell_k, v \rangle \in L_k$
7.              $C = \langle \ell_1, \ldots, \ell_k, v \rangle$
8.              **if** $(\forall S \preceq C \Rightarrow S \in L_k$, where $|S| = k$ and $S \neq L')$
9.                  insert $C$ in the *candidate-trie*
10.             **endif**
11.         **endif**
12.     **endfor**
13. **endfor**
14. **return** *candidate-trie*
**end**
```

Fig. 2. Candidate generation algorithm.



<B, E, C>
<D, B, A>
<B, D, A>
<D, B, E, C>
<B, A, C>
<D, A, C>

Fig. 3. *a:* An example of site structure. *b:* A database of access sequences.

with respect to Definition 1. Thus, candidate $\langle B, A \rangle$ is contained in the third transaction, i.e., $\langle B, D, A \rangle$, although documents B and A are not consecutive in the transaction. $\dashv$

With the procedure described in Figure 2, candidate generation is performed by extending candidates according to their outgoing edges in the graph. Consequently, the ordering is preserved and only paths in the graph are considered. Additionally, the apriori-pruning criterion of [3] is modified, since, for a given candidate, only its subsequences have to be tested and not any arbitrary subset of documents (step 8), as in [3]. Candidates are stored in a trie structure. Each transaction that is read from the database is decomposed into the paths it contains and each one of them is examined against the trie, thus updating the support of the corresponding candidates. Procedure GenCandidates aims at improving the efficiency of candidate generation. For this reason, step 3 determines the set N^+ of nodes that are linked by the last element of the candidate.

This way, in step 6 the trie is probed only by the corresponding L' sequences. In contrast, if N^+ is not determined (because one may consider that the arc $l_k \rightarrow v$ will be examined during the probing of the trie for the L' sequences that are members of L_k), the formation of L' sequences would first require the consideration of any possible element to be appended at the end of L and then the probing of the trie for all possible L' (since no combination can be discarded without having first to look-up in the trie). Evidently, the cost for the separate determination of N^+ pays off in the former case due to the much smaller number of trie look-ups.

Although several heuristics have been proposed for the reduction of the number of candidates for the Apriori algorithm, they involve basket data. The pruning with respect to the site structure is required for the particular problem, due to the large increase in the number of candidates that is the result of preserving ordering. The pruning criterion does not have the objective of addressing the noise that may occur within user transactions. This is addressed by the containment criterion given in Definition 1. In contrast, it opts for a reduction in the number of candidates, since the latter significantly affects the overall execution time required for the candidate generation and support counting procedures [3]. Hence their efficiency is improved by this pruning criterion. The effectiveness of pruning is verified by experimental results in Section 5. More details can be found in [32,33], whereas a further examination of the generalization of the described prefetching algorithm compared to existing ones, can be found in [31]. Finally, it worths mentioning that the proposed pruning criterion is used in combination with the support-pruning criterion. However, the modified-apriori criterion is applied (steps 6–9 of the GenCandidates procedure), which examines only the subpaths of a sequence and not any arbitrary subsequence.

4 Caching Policy

Based on the idea in [25], we describe a caching scheme *PECache*, which divides the cache into a *Weighing Room* (with *LRU* as the replacement policy) and a *Waiting Room* (using the *FIFO* policy). This partitioning of the cache space aims at isolating the effect of document mispredictions or the effect of aggressive prefetching. It achieves this by dedicating part of the cache space to exploit the temporal locality of the request stream (on-demand requests) and the rest of the cache space is dedicated to exploit the spatial locality (prefetch requests). The relative size of the partitions should reflect the “amount” and type of the locality of the request stream.

The caching procedure *PECache* (Prefetch Enhanced Cache), given in Figure 4, has as input the requested document (d) and the current request stream of the user (R).

The *PECache* procedure uses the prefetching algorithm (step 3) that may return several documents, whereas W^{2}R uses the *OBL* prefetching algorithm, which always prefetches one document. Therefore, differently from the W^{2}R algorithm, the set of prefetched documents are inserted in the *FIFO* structure

```
Procedure PECache(Array R, Document d)
begin
1.  $R = R \bigcup d$
2.  if not (d in Weighing Room or d in Waiting Room)
2.      put d at head of the LRU list of the Weighing Room
3.      prefetchSeq = Prefetch(R, M, maxSize)
4.      foreach p in prefetchSeq
5.          append p at the end of Waiting Room queue
6.      endfor
7.  else if d in Waiting Room
8.      remove d from Waiting Room
9.      put d at head of the LRU list of the Weighing Room
10. else if d in Weighing Room
11.     put d at head of the LRU list of the Weighing Room
12. endif
end
```

Fig. 4. The caching procedure.

of the Waiting Room according to the corresponding confidence values (this is performed at step 5 of the *PECache* procedure). It is assumed that at steps 4–5 of the *PECache* procedure the prefetched documents enter the *FIFO* structure in the exact order they were requested, i.e., the caching mechanism resolves the issues of identifying the documents that belong to the same *prefetchSeq* and sorting them according to the requested order. Moreover, differently from W^2R, the *PECache* procedure does not perform prefetching in the case the requested document *d*, is contained in the Waiting Room (step 7). Otherwise, this would result in excessive network traffic and bandwidth consumption (notice that W^2R is designed for buffer management in a DBMS).

It should be mentioned that the replacement policy used in the Weighing Room can be selected independently. For instance, the *2Q* algorithm can be used, as described in [25] or some other policy more appropriate for the Web, as in [6]. For simplicity and based on [25], we assume in the following that the *LRU* policy is used. *FIFO* seems the most appropriate policy for the Waiting Room, since it gives

4.1 The Cooperative Caching and Prefetching Mechanism

The caching scheme is applied at the client-side. Prefetching can be either client or server initiated. However, Web servers are in better position in making predictions about future references, since they log a significant[4] part of requests by all Internet clients for the resources they own. Moreover, the pruning criterion described in Section 3.2 can be easily applied in this case, since the site structure is available to the Web server.

[4] They only miss the requests satisfied by browser or proxy caches.

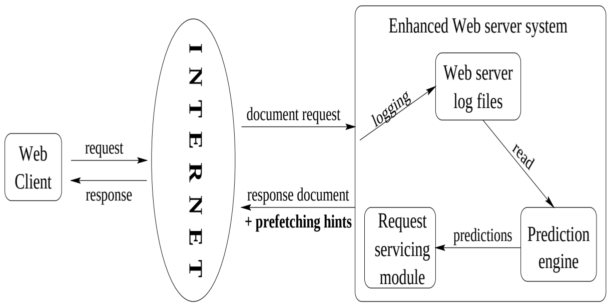


Fig. 5. Proposed architecture of a prediction-enabled Web server.

The prediction engine can be implemented by exchange of messages between the server and clients, having the server piggybacking information about the predicted resources onto regular response messages, avoiding establishment of any new TCP connections [16]. Such a mechanism has been implemented in [16,21] and seems the most appropriate, since it requires relatively few enhancements to the current request-response protocol and no changes to HTTP 1.1 protocol.

Therefore, we assume that there is a system implementing a server-based predictive prefetcher, which applies the Prefetch procedure of Section 3.1. The server piggybacks its predictions to the clients only as hints (in this case, the *prefetchSeq* in Prefetch procedure comprises these hints, i.e., *ID* numbers of the documents "to be prefetched"). The client receives these hints and discards all those which correspond to documents found in its cache. Then, in a second stage, prefetching takes place requesting the predicted documents. The caching of the requested documents (on-demand and prefetched) is performed with the *PECache* procedure.

Figure 5 illustrates how such an enhanced Web server could cooperate with a prefetch engine to disseminate hints every time a client requests a document of the server.

5 Performance Results

This section presents the experimental results. The cache performance is examined against the factors of high-order dependencies, amount of noise and cache size. The performance measure is the hit ratio achieved by the cache. We examine the performance of the proposed caching policy, presented in Section 4, in coordination with the prefetching algorithm, presented in Section 3. The proposed method is denoted as *PEC* (Prefetch Enhanced Cache). For the purposes of comparison, we also examine the cache performance in the case of using the *DG*, *PPM*, and LBOT[5] prefetching algorithms in coordination with the caching policy of Section 4, so as to clearly identify the advantages of the proposed prefetching algorithm.

[5] In the experiments, the algorithm proposed in [29] will be referenced as LBOT.

Additionally, we examine the performance of the plain *LRU* caching policy (i.e., when no prefetching is performed and only one cache partition is used), so as to identify the advantages of the proposed caching policy. In the case where one cache partition is used, its size is equal to the sum of sizes of the two partitions (i.e., Weighing and Waiting rooms) of the proposed caching policy. We separately examine the impact of the proposed pruning criterion (Section 3.2) on the reduction of the number of candidates during the generation of prefetching rules for the *PEC* method.

5.1 Generation of Synthetic Workloads

In order to evaluate the performance of the algorithms over a large range of data characteristics, we generated synthetic workloads. Each workload is a set of transactions. Our data generator implements a model for the documents and the linkage of the Web site, as well as a model for user transactions.

We choose so that all site documents have links to other documents, that is, they correspond to HTML documents. The fanout of each node, that is, the number of its outgoing links to other nodes of the same site, is a random variable uniformly distributed in the interval [1..*NFanout*], where *NFanout* is a parameter for the model. The target nodes of these links are uniformly selected from the site nodes. If some nodes have no incoming links after the termination of the procedure, then they are linked to the node with the greatest fanout. With respect to document sizes, following the model proposed in [7], we set the maximum size equal to 133KB and assign sizes drawn from a lognormal distribution[6] with mean value equal to 9.357KB and variance equal to 1.318KB.

In simulating user transactions, we generated a pool of P paths ("pattern paths", in the sequel). Each path is a sequence of linked in the site and pairwise distinct Web server documents, and will be used as "seeds" for generating the transactions. Each of these paths is comprised of 4 nodes (documents), simulating the minimum length of a transaction (clearly, for very small transactions i.e., of length two or three, the effect of prefetching is not significant, since user's interaction in this case does not allow for effective prediction of forthcoming requests). The paths are created in groups. Each group comprises a tree. The paths are actually the full length paths found in these trees. The fanout of the internal tree nodes is controlled by the parameter bf. Varying this parameter we are able to control the 'interweaving' of the paths. The nodes of these trees are selected using either the 80-20 fractal law or from the nodes that were used in the trees created so far. The percentage of these nodes is controlled by the parameter *order*, which determines the percentage of node dependencies that are non-first order dependencies. For example, 60% order means that 60% of the dependencies are non-first order dependencies. Thus, varying this parameter, we can control the order of the dependencies between the nodes in the path. The use of the fractal law results in some nodes to be selected more frequently than

[6] Without loss of generality, we assume that HTML files are small files. Thus, according to [7] their sizes follow a lognormal distribution.

Table 1. The parameters for the generator.

N	Number of site nodes
$NFanout$	Max num of nodes' links
T	Number of transactions
P	Number of pattern paths
bf	Branching factor of the trees
$order$	Order of the dependencies
$noiseMean$	Mean value of the noise
$noiseVar$	Variance of the noise
$corProb$	Prob. excluding a node

others. This fact reflects the different popularity of the site documents, creating the so-called "hot" documents.

In order to create the transactions, we first associate a weight with each path in the pool. This weight corresponds to the probability that this path will be picked as the "seed" for a transaction. This weight is picked from an exponential distribution with unit mean, and is then normalized so that the sum of the weights for all the paths equals 1. A transaction is created as follows. First, we pick a path, say $\langle A, B, C, x\rangle$, tossing a P-sided weighted coin, where the weight for a side is the probability of picking the associated path. Then, starting from node A we try to find a path leading to node B or with probability *corProb* to node C, whose length is determined by a random variable, following a lognormal distribution, whose mean and variance are parameters of the model. This procedure is repeated for every node of the initial path except from those that, with probability *corProb*, were excluded from the path. The mean and variance of the lognormal distribution determine the "noise" inserted in each transaction. Low values for mean and variance leave the transaction practically unchanged with respect to its pattern path, whereas larger values increase its length with respect to the pattern path. Table 1 summarizes the parameters of the generator.

5.2 Results on Cache Performance

In order to carry out the experiments we generated a number of workloads. Each workload consisted of T=100,000 transactions. From these, 30,000 transactions were used to train the algorithms and the rest to evaluate their performance. The number of documents of the site for all workloads was fixed to N=1000 and the maximum fanout to *NFanout*=100, so as to simulate a dense site. The branching factor was set to *bf*=4 to simulate relatively low correlation between the paths. The number of paths of the pool for all workloads was fixed to P=1000. With several experiments, not shown in this report, it was found that varying the values of the parameters P and N does not affect the relative performance of the considered algorithms. For all the experiments presented here, the *order* of the *PPM* algorithm was set equal to 5, so as to capture both low and higher order dependencies. The default value for the mean transaction size was set to 10. Throughout the experiments, the range of cache size was selected to be in the range of few hundred KB, to simulate the fact that not all, but only a small part of the Web client's cache is "dedicated" to the documents of a particular Web server.

The confidence threshold was tuned separately for each algorithm, so as to derive the same network traffic overhead, which is defined to be the number of documents that the client gets when prefetching is used divided by the one when prefetching is not used. The examined network traffic was 150%. This is also the value of the average network byte overhead. This means that for each byte the user requested, the prefetchers fetched another 0.5 byte that the user never requested. This is a relatively conservative approach considering that existing techniques and implementations incur a much larger overhead (250% overhead by [27], 412% by [15]).

First, we evaluated the impact of varying order on the hit ratio (Notice that the order of dependencies varies with the type of the site [10]). The mean noise value was set to 1.0. The total cache size was set to 150 KB and 50 KB of this total size were dedicated to the Waiting Room (this does not apply for the case of plain *LRU*). The results of this set of experiments are reported in Figure 6a. As illustrated, *PEC* is not affected by increasing order. This is in accordance to what expected from the discussion in Section 3. *PEC* clearly outperforms all other methods, achieving the highest hit ratio in all cases. *PPM*, as expected, is not heavily affected by increasing order and presents the second best performance. Focusing on *DG*, the impact of order in this case is noticeable. The hit ratio when *DG* is used as the prefetching algorithm, is reduced significantly with increasing order. This verifies the discussion on *DG* given in Section 2. In the case of LBOT, the hit ratio is also reduced with increasing order, however the reduction is much smaller than that of *DG*. Nevertheless, *DG* outperforms LBOT for smaller order values, whereas they present comparable performance for larger ones. Finally, the plain *LRU* caching policy presents the worst performance, in all cases. This verifies the advantages of exploiting prefetching and using the cooperative caching mechanism presented in Section 4.

Next, we assessed the impact of noise on the hit ratio. The order value was set to 0.5. The results of this set of experiments are reported in Figure 6b. As shown, the performance of all methods reduces with increasing noise, since noise alters transactions and reduces the number of discovered patterns. However, *PEC* clearly is much less affected by increasing noise, and it achieves the highest hit ratio in all cases. This verifies the discussion in Section 3. In contrast, *PPM* is significantly affected by noise. As depicted, its performance downgrades with increasing noise, and although it outperforms *DG* and LBOT for low mean noise values, it performs much worse for larger ones. This is in accordance with what expected from the description of Section 2. As in the previous experiment, *LRU* presents the worst performance in all cases.

We measured the impact of cache size. We kept the Waiting Room size equal to 50 KB and varied the total cache size (which includes the size of the Waiting

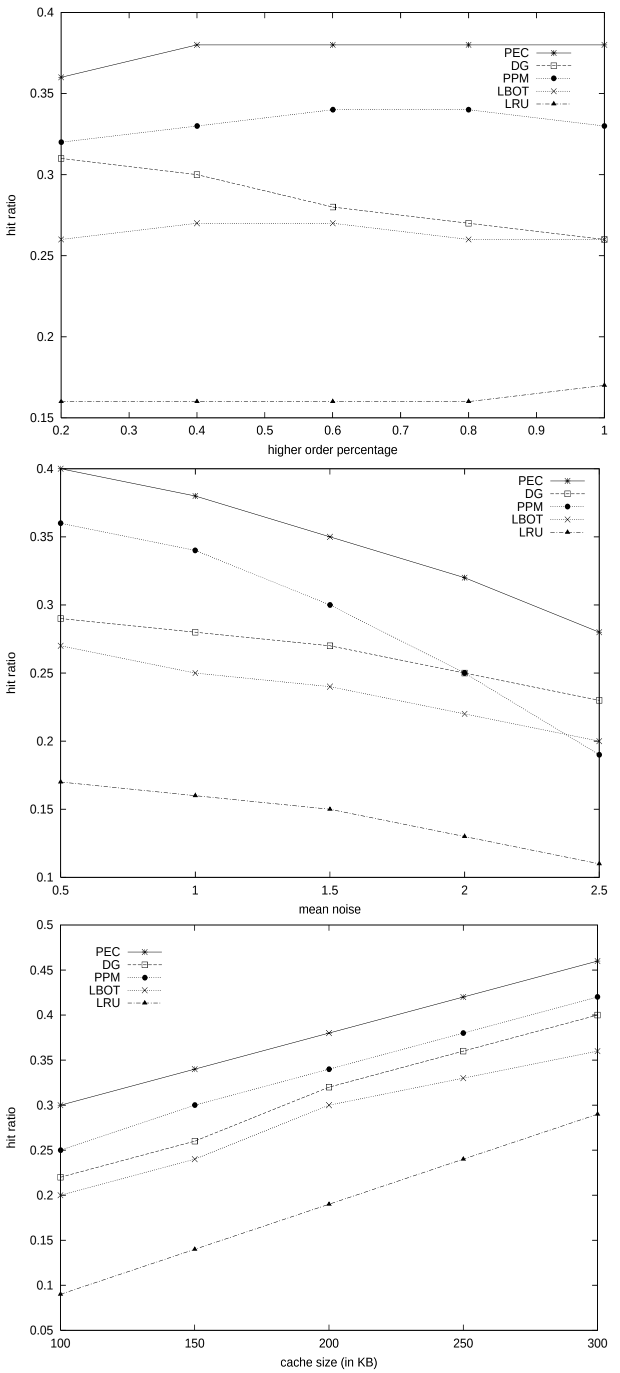


Fig. 6. Results on hit ratio w.r.t.: (a) order, (b) noise, and (c) cache size (from top to bottom, respectively).

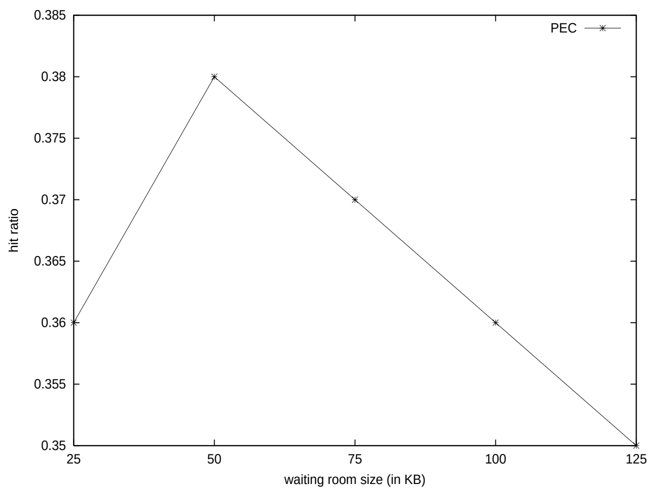


Fig. 7. Hit ratio w.r.t. Waiting Room size.

Room, besides the case of plain *LRU* caching policy). The mean noise value was set to 1.5 and the order was set to 0.5. Figure 6c illustrates the results for all methods. As depicted, the hit ratio increases linearly with increasing cache size. *PEC* presents the best performance in all cases, whereas *PPM* the second best. As in the previous cases, plain *LRU* presents the worst hit ratio among all methods.

To examine the impact of the size of the Waiting Room, we conducted a separate experiment. The total cache size was set to 200 KB and we varied the Waiting Room size. The results for the hit ratio of the *PEC* method are given in Figure 7 (We focus on *PEC* since this experiment concerns the tuning of the Waiting Room size and the performance of the remaining methods is analogous). For very small and large Waiting Room sizes, the hit ratio has lower values, compared to the case where it takes the maximum value (at the examined case, this value was 50 KB). This can be explained with the help of the notion of temporal and spatial locality [5]. When the size of the Waiting Room is very small, then only a relatively small percentage of the prefetched documents remain enough time in the cache, so as to get referenced and contribute to the hit ratio. Thus, the benefits of prefetching –even in the case of very accurate prefetching – are very limited. On the other hand, when the size of the Waiting Room is relatively large (compared to the total cache size), then the Waiting Room "steals" useful space from the Weighing Room and the cache is not capable of exploiting the temporal locality of the request stream. The tradeoff associated with the relative sizes of the Weighing and Waiting Rooms is obvious. For every request stream, depending on the "amount" of temporal and spatial locality (and of course, on the document sizes), there is an optimal value for the relative sizes of the two partitions. In our case, this value equals 50 KB.

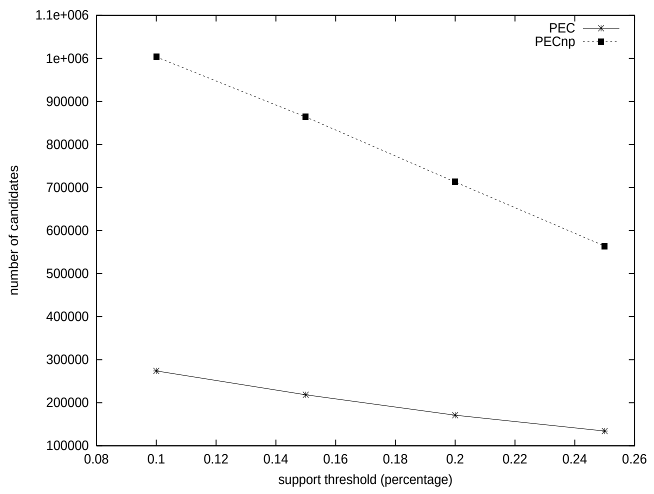


Fig. 8. Number of candidates w.r.t. support threshold.

5.3 Evaluation of Pruning Criterion

Finally, we examined the effectiveness of the proposed pruning criterion. We used a synthetic dataset with the same characteristics as the ones used in the experiments of Section 5.2. This experiment compares the *PEC* algorithm with a version that does not use pruning with respect to the site structure and is denoted as PEC_{np} (no pruning). It has to be noticed that the SP algorithm [4] generates candidates without taking into account the site structure. Moreover, SP also considers the ordering between items (for instance, in the second phase of SP, which is the bottleneck due to the very large number of candidates, let two large items A and B for which only the link $A \rightarrow B$ exists; however, SP generates both $\langle A, B\rangle$ and $\langle B, A\rangle$ as candidates whereas the latter can be pruned). Thus, for the same support threshold, the number of candidates produced by PEC_{np} corresponds to the one produced by SP.

Figure 8 illustrates the number of candidates for these methods with respect to the support threshold (given as a percentage). As it is depicted, *PEC* with pruning criterion significantly outperforms PEC_{np}, both for low and high support thresholds.

The number of candidates significantly impacts the performance of this type of algorithms. This is in accordance with related work on association rule mining [3]. Therefore, the efficiency of *PEC* is improved by the proposed pruning. Detailed experimental results on execution time for the case of pruning according to the graph structure can be found in [33].

6 Conclusions

We considered the problem of enhancing Web caching with the technique of predictive Web prefetching, that is, of deriving users' future requests for Web documents based on their previous requests. We also examined the problem of the coordination between Web caching and prefetching.

We proposed a new algorithm called *PEC*, which focuses both on attaining accurate prefetching and using cooperative caching so as to effectively accommodate the prefetched documents with the normal cached ones (i.e., the ones cached after an explicit user request). For the former factor, we described a prefetching algorithm, which exploits Web log mining techniques. To address the problem of large computational overhead for the rule generation phase, we described a pruning criterion that is based on the site structure. For the latter factor, based on the approach of [25], we presented an algorithm which uses a small part of the cache so as to separately store the prefetched documents. We addressed the new requirements due to the particularities of the Web, compared to the case of DBMS buffer management, as examined in [25].

Experimental results illustrated the superiority of *PEC*. In contrast to existing methods, *PEC* is not affected by factors like high-order dependencies among document references, or the existence of noise within user transactions. Also, experimental results showed the effectiveness of the pruning criterion.

Future work includes:

– The examination of other caching policies within the framework of *PEC*.
– The development of dynamic methods for the tuning of the Waiting Room size.

References

1. M. Abrams, C.R. Standridge, G. Abdulla, E.A. Fox, and S. Williams. Removal policies in network caches for World-Wide Web documents. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (ACM SIGCOMM'96)*, pages 293–305, 1996.
2. C. Aggarwal, J. Wolf, and P.S. Yu. Caching on the World Wide Web. *IEEE Transactions on Knowledge and Data Engineering*, 11(1):95–107, 1999.
3. R. Agrawal and R. Srikant. Fast algorithms for mining association rules. In *Proceedings of the 20th Conference on Very Large Data Bases (VLDB'94)*, pages 487–499, 1994.
4. R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the IEEE Conference on Data Engineering (ICDE'95)*, pages 3–14, 1995.
5. V. Almeida, A. Bestavros, M. Crovella, and A. de Oliveira. Characterizing reference locality in the WWW. In *Proceedings of the IEEE Conference on Parallel and Distributed Information Systems (IEEE PDIS'96)*, pages 92–103, 1996.
6. M. Arlitt, L. Cherkasova, J. Dilley, R. Friedrich, and T. Jin. Evaluating content management techniques for Web proxy caches. *ACM SIGMETRICS Performance Evaluation Review*, 27(4):3–11, 2000.
7. P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems, (ACM SIGMETRICS'98)*, pages 151–160, 1998.

8. B. Berendt and M. Spiliopoulou. Analysis of navigation behavior in Web sites integrating multiple information systems. *The VLDB Journal*, 9(1):56–75, 2000.
9. A. Bestavros. Speculative data dissemination and service to reduce server load, network traffic and service time. In *Proceedings of the IEEE Conference on Data Engineering (ICDE'96)*, pages 180–189, 1996.
10. J. Borges and M. Levene. Data mining of user navigation patterns. In *Proceedings of the Workshop on Web Usage Analysis and User Profiling (WEBKDD'99)*, pages 92–111, 1999.
11. S. Brin and L. Page. The anatomy of large-scale hypertextual Web search engine. In *Proceedings of the World Wide Web Conference (WWW'98)*, pages 107–117, 1998.
12. P. Cao and S. Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings USENIX Symposium on Internet Technology and Systems (USITS'97)*, pages 193–206, 1997.
13. P. Cao, J. Zhang, and K. Beach. Active Cache: Caching dynamic contents on the Web. In *Proceedings of the IFIP Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98)*, pages 373–388, 1998.
14. M.S. Chen, J.S. Park, and P.S. Yu. Efficient data mining for path traversal patterns. *IEEE Transactions on Knowledge and Data Engineering*, 10(2):209–221, 1998.
15. K. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the INET Conference*, 1997.
16. E. Cohen, B. Krishnamurthy, and J. Rexford. Improving end-to-end performance of the Web using server volumes and proxy filters. In *Proceedings of the ACM Conference on Applications, Technologies, Architectures and Protocols for Computer Communication (ACM SIGCOMM'98)*, pages 241–253, 1998.
17. R. Cooley, B. Mobasher, and J. Srivastava. Data preparation for mining World Wide Web browsing patterns. *Knowledge and Information Systems*, 1(1):5–32, 1999.
18. K.M. Curewitz, P. Krishnan, and J.S. Vitter. Practical prefetching via data compression. In *Proceedings of the ACM Conference on Management of Data (ACM SIGMOD'93)*, pages 257–266, 1993.
19. J. Dean and M. Henzinger. Finding related pages in the World Wide Web. In *Proceedings of the World Wide Web Conference (WWW'99)*, pages 1467–1479, 1999.
20. M. Deshpande and G. Karypis. Selective Markov models for predicting Web page accesses. In *Proceedings of the SIAM Conference on Data Mining (SDM'01)*, 2001.
21. D. Duchamp. Prefetching hyperlinks. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'99)*, 1999.
22. L. Fan, P. Cao, W. Lin, and Q. Jacobson. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings of the ACM Conference on Measurement and Modeling of Computer Systems (ACM SIGMETRICS'99)*, pages 178–187, 1999.
23. W. Gaul and L. Schmidt-Thieme. Mining Web navigation path fragments. In *Proceedings of the Workshop on Web Usage Analysis and User Profiling (WEBKDD'00)*, 2000.
24. S. Hosseini-Khayat. On optimal replacement of nonuniform cache objects. *IEEE Transactions on Computers*, 49(8):769–778, 2000.
25. H.S. Jeon and S.H. Noh. A database disk buffer management algorithm based on prefetching. In *Proceedings of the ACM Conference in Information and Knowledge Management (ACM CIKM'98)*, pages 167–174, 1998.

26. T. Johnson and D. Shasha. 2Q: A low overhead high performance buffer management replacement algorithm. In *Proceedings of the 20th Conference on Very Large Data Bases (VLDB'94)*, pages 439–450, 1994.
27. R. Klemm. WebCompanion: A friendly client-side Web prefetching agent. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):577–594, 1999.
28. T. Kroeger, D.E. Long, and J. Mogul. Exploring the bounds of Web latency reduction from caching and prefetching. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'97)*, pages 13–22, 1997.
29. B. Lan, S. Bressan, B.S. Ooi, and Y. Tay. Making Web servers pushier. In *Proceedings of the Workshop on Web Usage Analysis and User Profiling (WEBKDD'99)*, 1999.
30. H. Mannila, H. Toivonen, and A.I. Verkamo. Discovery of frequent episodes in event sequences. *Data Mining and Knowledge Discovery*, 1(3):259–289, 1997.
31. A. Nanopoulos, D. Katsaros, and Y. Manolopoulos. A data mining algorithm for generalized Web prefetching. *IEEE Transactions on Knowledge and Data Engineering*, 2002. to appear.
32. A. Nanopoulos and Y. Manolopoulos. Finding generalized path patterns for Web log data mining. In *Proceedings of the East-European Conference on Advances in Databases and Information Systems (ADBIS-DASFA'00)*, pages 215–228, 2000.
33. A. Nanopoulos and Y. Manolopoulos. Mining patterns from graph traversals. *Data and Knowledge Engineering (DKE)*, 37(3):243–266, 2001.
34. V. Padmanabhan and J. Mogul. Using predictive prefetching to improve World Wide Web latency. *ACM SIGCOMM Computer Communications Review*, 26(3), 1996.
35. T. Palpanas and A. Mendelzon. Web prefetching using partial match prediction. In *Proceedings of the 4th Web Caching Workshop*, 1999.
36. H. Patterson, G. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the ACM Symposium on Operating Systems Principles (ACM SOSP'95)*, pages 79–95, 1995.
37. J. Pei, J. Han, B. Mortazavi-Asl, and H. Zhu. Mining access patterns efficiently from Web logs. In *Proceedings of the Pacific-Asia Conference on Knowledge Discovery and Data Mining (PAKDD'00)*, 2000.
38. P. Pirolli, H. Pitkow, and R. Rao. Silk from a sow's ear: Extracting usable structures from the Web. In *Proceedings of the ACM Conference on Human Factors and Computing Systems (ACM CHI '96)*, pages 118–125, 1996.
39. J. Pitkow and P. Pirolli. Mining longest repeating subsequences to predict World Wide Web surfing. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'99)*, 1999.
40. R. Sarukkai. Link prediction and path analysis using Markov chains. *Computer Networks*, 33(1–6):377–386, 2000.