Available at www.ComputerScienceWeb.com
POWERED BY SCIENCE DIRECT®

ELSEVIER

Information Sciences 150 (2003) 165–176

INFORMATION SCIENCES
AN INTERNATIONAL JOURNAL

www.elsevier.com/locate/ins

Optimizing Web servers using Page rank prefetching for clustered accesses

Victor Safronov *, Manish Parashar

Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, 94 Brett Road, Piscataway, NJ 08854-8058, USA

Received 30 December 2001; received in revised form 17 March 2002; accepted 4 July 2002

Abstract

This paper presents a Page rank-based prefetching technique for accesses to Web page clusters. The approach uses the link structure of a requested page to determine the "most important" linked pages and to identify the page(s) to be prefetched. The underlying premise of our approach is that in the case of cluster accesses, the next pages requested by users of the Web server are typically based on the current and previous pages requested. Furthermore, if the requested pages have a lot of links to some "important" page, that page has a higher probability of being the next one requested. An experimental evaluation of the prefetching mechanism is presented using real server logs. The results show that the Page rank-based scheme does better than random prefetching for clustered accesses, with hit rates of 90% in some cases.
© 2002 Elsevier Science Inc. All rights reserved.

Keywords: Web prefetching; Page rank; Web server; HTTP server; Page clusters

1. Introduction

It is indisputable that the recent explosion of the World Wide Web has transformed not only the disciplines of computer-related sciences and

* Corresponding author. Tel.: +1-732-445-5388; fax: +1-732-445-0593.
E-mail addresses: safronov@ece.rutgers.edu, vsafronov@att.com (V. Safronov), parashar@ece.rutgers.edu (M. Parashar).

0020-0255/02/$ - see front matter © 2002 Elsevier Science Inc. All rights reserved.
PII: S0020-0255(02)00375-4

engineering but also the lifestyle of people and economy of countries. The single most important piece of software that enables any kind of Web activity is the Web server. Since its inception the Web server has always taken a form of a daemon process. It accepts an HTTP request, interprets it and serves a file back. While *CGI* and *Servlets* extend on these capabilities, file serving remains a key function of the Web server. As Web service becomes increasingly popular, network congestion and server overloading have become significant problems. Great efforts are being made to address these problems and improve Web performance.

Web caching is recognized as one of the effective techniques to alleviate the server bottleneck and reduce network traffic, thereby reducing network latency. The basic idea is to cache recent requested pages at the server so that they do not have to be fetched again. Regular caching however, only deals with previously requested files. New files will never be in the cache by definition. Web prefetching, which can be considered as "active" caching, builds on regular Web caching and helps to overcome its inherent limit by attempting to guess what the next requested page will be. For regular HTML file accesses, prefetching techniques try to predict the next set of files/pages that users will request, and uses this information to prefetch the files/pages into the server cache. This greatly speeds up access to those files and the users experience. To be effective however, the prefetching techniques must be able to reasonably predict (with minimum computational overheads) subsequent Web accesses.

In this paper, we present a Web prefetching mechanism for clustered accesses based on *Page rank*. Clustered accesses are access to closely related pages. For example access to the pages of a single company or research group or to pages associated to the chapters of a book. Clustered accesses are very common and accounted for over 70% of the accesses in the server logs that we studied. These included logs from the University of California, Berkeley Computer Science Division (for year 2000) and Rutgers University Center for Advanced Information Processing (for year 2000). Page rank uses link information in a set of pages to determine which pages are most pointed to and, therefore, are most important relative to the set. This approach has been successfully used by the GOOGLE [7] search engine to rank pages (or clusters of pages) that match a query. In the prefetching mechanism presented, we examine requested pages and compute Page rank for the pages pointed by the requested page. We then use this information to determine the page(s) to be prefetched. Note, that the most pointed to page need not have been requested before. Therefore, the approach we describe here is prefetching and not simple caching.

The rest of this paper is organized as follows. Section 2 describes Web prefetching and presents related work. Section 3 introduces Page rank and describes the Page rank-based prefetching approach. This section presents the algorithm, analyzes its complexity. Section 4 presents an experimental evaluation of the approach. Section 5 presents conclusions and future work.

2. Web-page prefetching—overview and related work

Web prefetching builds on Web caching to improve the file access time at Web servers. The memory hierarchy made possible by caches helps to improve HTML page access time by significantly lowering average memory/disk access time. However, cache misses can reduce the effectiveness of the cache and increase this average time. Prefetching attempts to transfer data to the cache before it is asked for thus lowering the cache misses even further. Prefetching techniques can only be useful if they can predict accesses with reasonable accuracy and if they do not represent a significant computational load at the server. Note that prefetching files that will not be requested not only wastes useful space in the cache but also results in wasted bandwidth and computational resources.

2.1. Related work

Existing prefetching approaches are client-side, proxy-based or server-side. In the client-side approach, the client determines pages to be prefetched and request them from the server. Client-side prefetching is presented by Jiang and Kleinrock in [4]. A key drawback of this approach is that it typically requires modifications to the client browser code or use of a plug-in, which may be impractical. Furthermore, it may double the required bandwidth actually resulting in deteriorated performance. For example, in the worst-case, the prefetcher will repeatedly request files that the user never wants to see. Therefore, the number of requests to the server will double without any benefit to the user. Finally, maintaining cache coherency in client-side prefetching approaches is expensive.

The proxy-based prefetching approach uses an intermediate cache between the server and a client [9]. This proxy can request files to be prefetched from the server, or the server can push some files to the proxy. Both of these schemes increase the required bandwidth. Furthermore, like client-side schemes, maintaining cache coherency in proxy-based schemes is expensive.

One advantage of client and proxy side prefetching is that they separate the HTTP server part from the caching part thus allowing greater geographic and IP proximity to the client. For example, placing a proxy cache next to or inside of an organization's subnet means that the data user requests will have far fewer IP hops. These schemes are also better suited for user-pattern tracking algorithms. In particular, the client-side mechanism is dedicated to a particular user and spends all its time trying to follow what the user might want. Another advantage is that requests from multiple servers can be cached.

In server-side approaches, the entire prefetching mechanism resides on the Web server itself. These approaches avoid the problems mentioned above. There is no increase in the bandwidth, as no files that have not been requested will be sent to the client. Furthermore, maintaining cache coherency in this case is almost trivial. Proxy-based caches and client-side prefetching mechanisms require additional messaging and protocols between the cache and the HTTP server for cache coherency. This overhead can become expensive in terms of wasted bandwidth. There is no complicated protocol and no extra messaging outside the server in case of sever-side schemes. As the file system in this case is either local or mounted, all the messaging is within the server and does not require external bandwidth. Furthermore, the OS file system guarantees access to the latest copy of a file, and provides excellent and easy to use mechanisms to check file attributes such as creation and modification times and dates, to assist in maintaining cache coherency. Another distinction with the client-related schemes is that client-side prefetching makes decisions on which files to prefetch based on the particular user's preferences, whereas in the server-side prefetching, decisions are based on the document popularity and more than one client can benefit from it.

A server-side prefetching approach based on analyzing server logs and predicting user actions on the server side is presented by Su et al. in [5]. Tracking users on a server, however, is quickly becoming impractical due to the widespread use of Web proxies. The proxy either presents one IP address to the server for a large group of users, or it cycles through some set of IP addresses according to its load-balancing scheme. Both cases render a single user identity moot.

The work presented by Zukerman et al. in [6], uses AI-related techniques to predict user requests. They implement a learning algorithm such as some variation of Markov chains and use some previous log accesses in order to train it. This approach also relies on tracking user patterns. Furthermore, it does not handle newly introduced pages, or old pages that have changed substantially. This approach also requires a rather long sequence of clicks from a user to learn his/her access patterns.

The Page rank-based prefetching technique presented in this paper is a server-side approach and uses the information about the link structure of the pages and the current and past user accesses to drive prefetching. The approach is effective for access to Web page clusters, is computationally efficient and scalable, and can immediately sense and react to changes in the link structure of Web pages. Furthermore, the underlying algorithm uses relatively simple matrix operations and is easily parallelizable, making it suitable for cluster server environments.

3. Page rank-based Web prefetching

3.1. Background

Serving files to a requesting client had been implemented long before the advent of the Web. Applications such as file servers and networked file systems are well known. However, it has been recognized, that serving Web requests presents a unique set of challenges. General Web files are text files containing HTML [8] syntax, and tend to be relatively small in size. A key feature of HTML is the ability to embed links to other files. Attempts have been made to try to utilize this special structure of HTML files for various purposes, particularly searching. One application based entirely on the link structure is the Page rank technique utilized by the GOOGLE [7] search engine.

The Page rank technique [1,2] provides a ranking of Web pages based on the premise that pages pointed to the most must be the most important ones. In this technique, the importance of a page is defined recursively, that is, a page is important if important pages link to it. To calculate the actual rank of the page a stochastic matrix is constructed as follows:

1. Each page i corresponds to row i and column i of the matrix.
2. If page j has n successors (links), then the ijth entry is $1/n$ if page i is one of those n successors of page j, 0 otherwise.

In our prefetching scheme, we only consider links to other pages on the same server. After the matrix has been populated the actual calculation is performed. This essentially consists of a principal eigenvector calculation [3]. Some additional modifications are required in order to avoid a few Web graph quirks. Web pages that have no outward links or those that only link to themselves have to be specially dealt with. One solution to these problems is to "tax" each page some fraction of its current importance instead of applying the matrix directly. The taxed importance is distributed equally among all pages. The overall algorithm is present below.

3.2. Page rank algorithm

To illustrate the Page rank algorithm used for prefetching, consider the Web page graph shown in Fig. 1.

Then the equation to be solved to determine the Page rank is as follows.

$$\begin{bmatrix} n \\ m \\ a \end{bmatrix} = \begin{bmatrix} 1/2 & 0 & 1/2 \\ 0 & 1 & 1/2 \\ 1/2 & 0 & 0 \end{bmatrix} \begin{bmatrix} n \\ m \\ a \end{bmatrix} + \begin{bmatrix} 0.2 \\ 0.2 \\ 0.2 \end{bmatrix}$$

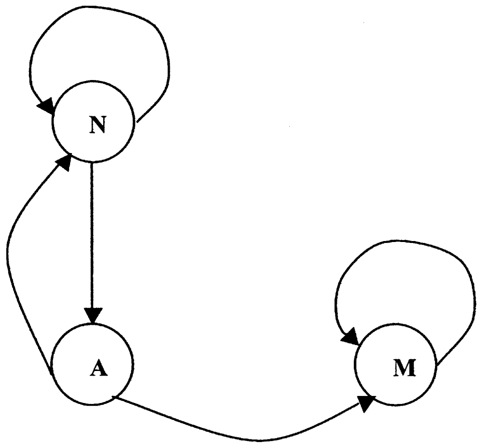

Fig. 1. Example graph of a cluster.

The solution of this equation is computed iteratively, by comparing the *norm* of the current resulting vector with that of the previous one until the difference is less than some delta. That is, if M is the matrix and R is the $[n, m, a]$ vector, the following algorithm is executed.

```
DO
  $R$previous = $R$;
  $R = M \times R + 0.2 \times [1]$;
WHILE $|R\text{previous}|_1 - |R|_1 > \Delta$;
Multiply each page' s rank by the number of requests;
```

The solution of this equation is $n = 7/11$; $m = 21/11$; $a = 5/11$—i.e. M is the most important page.

3.3. Page rank-based prefetching

The Page rank-based prefetching approach uses the link structure of a requested page to determine the "most important" linked pages and to identify the page(s) to be prefetched. The underlying premise of the approach is that the next pages requested by users of the Web server are typically based on the current and previous pages requested. Furthermore, if the requested pages have a lot of links to some "important" page, that page has a higher probability of being the next one requested. The relative importance of pages is calculated using the Page rank method as described above. The important pages identified are then prefetched into the cache to speed up users' access to them.

For each page requested, the Page rank algorithm performs the following operations. First the URL is scanned to see if it belongs to a cluster. If it does as soon as the contents of that page are retrieved they are used to populate or update that cluster's matrix. As soon as the matrix update operation is complete, the Page rank calculations are performed to determine the most important pages among those requested or pointed to in the cluster. A configurable number of these pages are then prefetched into the cache. It is also important to note that if the matrix and/or cache cannot hold all the pages, Page rank is used as a replacement mechanism, i.e. those pages with the lowest rank get replaced with new ones.

While GOOGLE uses the Page rank technique for Web searching, we use it for prefetching i.e., it is not used as a "spider" scouting the whole of the Web. We apply the ranking calculations described above only to pages on a single server. Furthermore, we only apply it for pages that are part of a defined cluster. Finally, prefetching calculations are real time by nature. As soon as new cluster access is processed the ranking calculations are performed to determine how the graph of requested pages has changed and which new pages need to be prefetched as a result of those changes. In other words, instead of building a static graph of the Web as in the original application, we build a dynamic graph of user accessed pages in a particular cluster on the server and use Page rank to determine which pages will be asked for next.

Since any random page on the server does not necessarily link to other pages on the same server we define the concept of Web page clusters. Clusters are groups of pages that are tightly interlinked. Those are the areas of the server where Page rank excels. Each cluster has it's own Page rank calculation. As soon as the front end determines that a page belongs to a cluster it is routed for its cluster's calculation. We heuristically define any Web directory with 200 or more files under it as a candidate cluster. We find the node closest to the root having this property but exclude the root itself. The justification is that there is a great chance that those files are related and are interlinked and their hierarchies should be sufficiently wide and deep.

3.4. Complexity of the Page rank prefetching algorithm

The prefetching mechanism has to be invoked for each access at the server. Consequently, it is imperative that the underlying algorithm be efficient. A complexity analysis of the algorithm is presented in this section. The main part of the Page rank algorithm consists of populating the matrix and then calculating its principal eigenvector. These are two consecutive operations:

1. Matrix population (simplified)
   - For each new page find all the pages it links to, also find all the pages that link to it.

- Keep an array of pages of length $\boldsymbol{n}$ to streamline matrix population.
- Find all pages that the new page links to. This requires a full array scan. For each array element, all the links on the new page need to be checked. Our observations show that it is rare for a page to have more than 20 links to pages on the same server. Consequently, if $\boldsymbol{n}$ is the maximum number of links on a page, then the worst case performance is $\mathbf{O}(\boldsymbol{n}^2)$.
- Find all the pages that link to the new page. This again requires a full array scan consisting of a scanning of the links on the current page and comparing them to the link to the new page. Making the same assumption as above, we have a worst case performance of $\mathbf{O}(\boldsymbol{n}^2)$.
- The two operations above are consecutive and can be combined into one with the same $\mathbf{O}(\boldsymbol{n}^2)$ complexity. Furthermore, ordering the array would not change the worst-case performance.
- Recalculate the matrix values. This as an $\mathbf{O}(\boldsymbol{n}^2)$ complexity as well.

2. Matrix multiplication
   - Iterative matrix-vector multiplication and addition. This typically converges in less than 20 iterations.
   - The cost of multiplying a $\boldsymbol{n} \times \boldsymbol{m}$ matrix by a $\boldsymbol{m} \times \boldsymbol{p}$ matrix is $\mathbf{O}(\boldsymbol{nmp})$. We have $\boldsymbol{n} \times \boldsymbol{n}$ by $\boldsymbol{n} \times \mathbf{1}$ therefore our multiplication algorithm's cost is $\mathbf{O}(\boldsymbol{n}^2)$.

As a result we have the overall complexity of the Page rank prefetching algorithm as $\mathbf{O}(\boldsymbol{n}^2)$.

3.5. Implementation overview

We have implemented a prototype server with Page rank prefetching. The server was built on a cluster and performed all the basic functions required, but did not include any optimizations. It additionally maintained running statistics (i.e. hit rate). The architecture of the server is shown in Fig. 2. The main components of the server are the router (R), and HTTP handler (C) and prefetcher (P) pairs. Each component was implemented as a separate process. The P–C pairs were identical and were implemented on separate nodes of the cluster. The router ran on a dedicated machine. A key motivation for implementing the server on cluster was to exploit the inherent parallelism in the Page rank prefetching algorithm and maintain server scalability. Page rank computations for different clusters can be performed in parallel. Furthermore, the associated matrix computations can also be parallelized.

The router was simple and efficient. It accepted an incoming HTTP request, determined which cluster it belonged to, and handed it off to a P–C pair for Page rank computations. The HTTP handler performed the functions of a regular HTTP server with caching and custom prefetching. The prefetcher implemented the Page rank prefetching algorithm and decided which files needed to be prefetched. When the HTTP handler sent a reply page (either

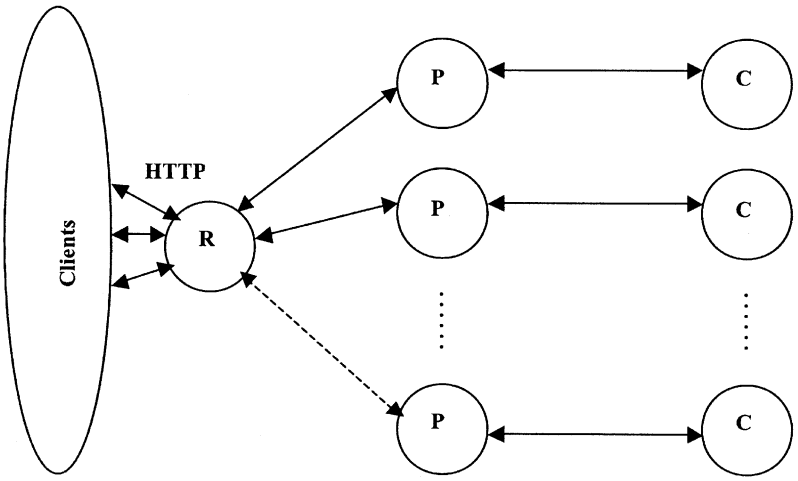

Fig. 2. Server architecture.

from its cache or from the disk) back, the prefetcher extracted the list of "*href*" links to local pages from this page, and computed the Page rank. It then sent a list of pages to be prefetched to the HTTP handler. The client, in the meantime, received the page it requested. Both the router and the prefetcher are multi-threaded for further efficiency.

The Page rank prefetcher calculates the pages to be prefetched on the fly allowing the server to respond very quickly to any change in access pattern popularity. The server prefetches pages that are not yet accessed and registers changes in the page's contents as soon as the page is accessed again. In other words, the router maintains a running rank of pages on the server based on the pages accessed so far.

The distributed server achieves almost perfect scalability as processing for each cluster is performed independently. The overall runtime in this case is the maximum of the computation times for the cluster plus some synchronization overheads. A single server would have processed the request sequentially resulting in an overall runtime equal to the sum of the computation times for each cluster. Running with 12 machines in a cluster reduced the overall running time by a factor of 8.

Our experiment showed that a matrix size $\boldsymbol{n} = 200$ resulted in the most appropriate balance. Matrices of size less than 10 produce results that were fast but were not useful for prefetching. On the other hand, running with a matrix size of 1000 took an unacceptably long time on an 850 MHz PIII with 256 MB RAM running Windows 2000. A matrix size of 200 gave good prefetching predictions and had a reasonable computational cost. Similarly, we empirically found that the most appropriate fraction of pages in the cache that should be prefetched is 0.25. Values that were too high wasted cache space while values that are too low wasted computational effort.

4. Experimental evaluation

We used server logs from the University of California, Berkeley Computer Science Division (for year 2000) (www.cs.berkeley.edu) and Rutgers University Center for Advanced Information Processing (for year 2000) (www.caip.rutgers.edu) to experimentally evaluate the Page rank-based perfecting mechanism. In particular, we chose September 2000 log as a representative one for our experiment. The experiment consisted of identifying the access clusters in the logs and extracting requests to these clusters. The accesses were then used to drive the evaluation, which consisted of measuring the hit rate for accesses at server with the Page rank-based prefetching scheme versus a random prefetching scheme.

We define hit rate as follows. Let H be the number of user requests that were found in cache at the time of the request. Let M be the number of user requests that were not found in the prefetch cache. Then the total number of requests is $H + M$ and the hit rate is defined as

$$\text{Hit rate} = \frac{H}{H+M} \times 100\%$$

Using our heuristic, we found 28 clusters on the Berkeley server, constituting about 70% of all the files on the server. So these clusters are quite common. We extracted requests for each cluster and used them to evaluate our prefetching scheme. The results are as follows. Hit rates per cluster range from 0% to 95%. In all, 61% of all the clusters gave hit rates greater than 30% (i.e. greater than random). Requests to those clusters constitute about 15% of all the requests in the log.

Using the log from the CAIP server for November 2000 we found the following. There are 12 clusters as defined by our heuristic. Files in those clusters constitute 49% of all the files on the server. Requests to those files constitute 39% of all the server requests.

Fig. 3 shows the cluster accesses for the CAIP log. As can be seen from the chart the hit rate varies from 20% all the way to 95% with only one cluster having the hit rate less than 30%. One half of all the clusters have hit rate greater than 70% and one quarter reach or exceed 90%. This again shows that cluster pages are common, that they account for a substantial number of requests, and that the Page rank scheme does very well prefetching these type accesses.

It should be noted that the heuristic we employed is a temporary solution to finding the clusters. It should be relatively straightforward to develop a spider

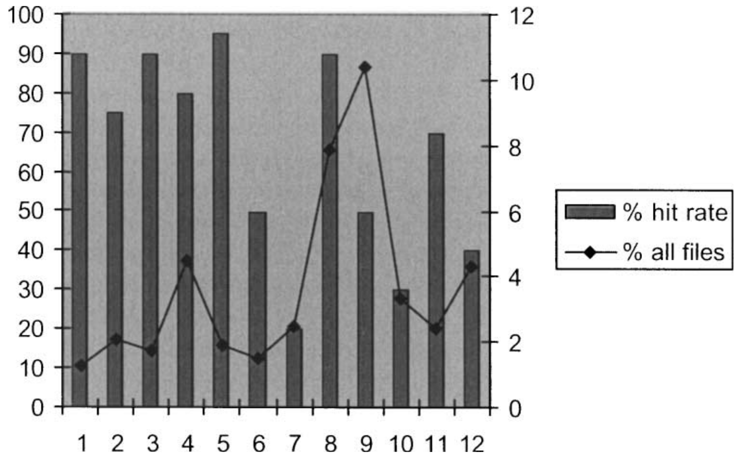

Fig. 3. CAIP clusters and accesses.

that will crawl all the pages on the server and discover clusters. Threshold of connectivity for the cluster definition is a subject of future research. We predict that having defined the clusters in a more systematic way will increase hit rate even further. It may also discover more clusters and files belonging to clusters.

We also note that the Page rank prefetcher did not do well for non-clustered request. In this case the hit rate was about 17%. The random prefetcher resulted in a hit rate of about 30%. This is expected as the Page rank prefetcher is based on the premise that page link information determines accesses, which is true for clustered accessed but typically not true for random accesses.

5. Conclusions and future work

In this paper we presented the Page rank-based prefetching mechanism for clustered Web page accesses. In this approach, we rank the pages linked to a requested page and use this determine the pages to be prefetched. We also presented an experimental evaluation of the presented prefetching mechanism using server logs from the University of California, Berkeley Computer Science Division (for year 2000) and Rutgers University Center for Advanced Information Processing (for year 2000). The results show that the Page rank prefetching does better than random prefetching for clustered accesses, with hit rates 90% hit rate in some cases. We have also shown that these clusters are quite common on both servers we explored. They constitute about 50% and 70% of all the files on the server. Accesses to pages in the clusters are about 15% and 40% of all the accesses.

We are currently building a spider for discovering page clusters. This work is also investigating the appropriate depth and breadth thresholds for cluster identification. We are investigating the type of Web sites that can benefit from the Page rank prefetching approach. Finally, we are implementing a distributed version of the prefetcher so that it can be efficiently deployed in a cluster environment.

References

[1] S. Brin, L. Page, The anatomy of a large-scale hypertextual web search engine, in: Proceedings of the Seventh World Wide Web Conference, Apr. 1998.

[2] S. Brin, L. Page, The Page Rank Citation Ranking: Bringing Order to the Web, January 29 1998.

[3] S.D. Conte, C. de Boor, Elementary Numerical Analysis, an Algorithmic Approach, McGraw-Hill, 1980.

[4] Z. Jiang, L. Kleinrock, An adaptive network prefetch scheme, IEEE Journal on Selected Areas in Communications 16 (1998) 358.

[5] Z. Su, Q. Yang, Ye. Lu Zhang, What Next: A Prediction System for Web Requests using N-gram Sequence Models, 1999.

[6] I. Zukerman, W. Albrecht, A. Nicholson, Predicting user's request on the WWW, in: UM99—Proceedings of the Seventh International Conference on User Modeling, 1999.

[7] The GOOGLE search engine. Available from <http://www.google.com>.

[8] Available from <http://www.w3.org/MarkUp/>.

[9] T.M. Kroeger, D.D.E. Long, J.C. Mogul, Exploring the bounds of Web latency reduction from caching and prefetching, in: Proceedings of the 1st USENIX Symposium on Internet Technologies and Systems, Dec. 1997, pp. 13–22.