World Wide Web: Internet and Web Information Systems, 5, 25–40, 2002

Optimizing Web Servers Using Page Rank Prefetching for Clustered Accesses [*]

VICTOR SAFRONOV and MANISH PARASHAR {safronov,parashar}@caip.rutgers.edu
Department of Electrical and Computer Engineering, Rutgers, The State University of New Jersey, 94 Brett Road, Piscataway, NJ 08854-8058, USA

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.

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 engineering but also the lifestyles of people and economies 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, i.e. by definition, new files will never be in the cache. Web prefetching, which can be considered as "active" caching, builds on regular Web caching and helps to overcome its inherent limitation. It attempts to guess what the next requested page will be. For regular HTML file accesses,

[*] The research presented in this paper was supported in part by the National Science Foundation under Grant Numbers EIA-0103674 (NGS) and ACI-9984357 (CAREERS) awarded to Manish Parashar.

prefetching techniques try to predict the next set of files/pages that will be requested, and use this information to prefetch the files/pages into the server cache. This greatly speeds up access to those files, and improves 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 with 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 [9] 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 to by the requested page. We then use this information to determine the page(s) to be prefetched. Note that the most pointed to page may not have been requested before. Therefore, the approach we describe here is prefetching and not simple caching. This paper makes the following contributions:

- It introduces the concept of Web page clusters and presents heuristics for identifying clusters.
- It defines a Page Rank based mechanism to predict accesses to page in Web page clusters. The predictions are used to drive a Web page prefetching mechanism that prefetches pages into the server cache to improve access times.
- It designs, implements and evaluates a distributed cluster-based architecture for the *Page Rank* prefetching server. The architecture provides good scalability and further improves server speed.

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. Section 3 also presents the algorithm and 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

2.1. Overview

If the World Wide Web is to be approached from a client–server view then, as the name suggests, Web server is the server part of the scheme and a browser is the client. In a typical interaction a user will request a file from a server either by clicking on a link or typing the request in manually. The browser translates it into an HTTP request, connects to the proper server, sends the request and waits for a reply. Meanwhile the Web server has been waiting for requests. It accepts the connection from the client, parses the HTTP request and extracts the name of the file. The server then gets the file from its cache or from its disk, formats an HTTP reply that satisfies the request and sends it to the browser. The browser then closes the connection.

Access to disk is much slower than access to memory. Just as in the case of OS file systems, caching techniques are used in Web servers to reduce disk accesses. One difference is that Web server file accesses are read-only due to the nature of the application. In this context the cache is a collection of files that logically belong on the disk but are kept in memory to optimize performance.

Web prefetching builds on Web caching to improve the file access time at Web servers. The memory hierarchy made possible by the 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.

In this paper we address prefetching rather than caching. Caching assumes a page has been requested at least once, while prefetching tries to guess which page a user will request in the future. We chose server side prefetching for the following reasons. The cost of a miss in the case of client-side prefetching is much larger than that for server-side prefetching. As explained in Section 2.2.1, client side misses can almost double the load and bandwidth requirements at the server in the worst case, and can actually result in the deterioration of the users' experience. In the case of server side prefetching, misses only result in wasted cache space at the server, and there is a good chance that another user may eventually request that page anyway. The algorithm presented in this paper makes decisions about which pages to prefetch based on the access popularity of the pages. This access popularity can be best determined on the server side. Server side access time is becoming a significant part of the overall Web access time and time saved on the server side can be important. This is evident while accessing a very busy server or when using a fast connection. Finally, our algorithm does not preclude caching on a proxy or on the client side and can be used to complement it.

2.2. *Related work*

Existing prefetching approaches can be classified as client-side, proxy-based or server-side. Table 1 summarizes the main features, advantages and disadvantages of each of these approaches.

2.2.1. Client-side prefetching. In the client-side prefetching approach, the client determines pages to be prefetched and requests them from the server. Client-side prefetching is

Table 1. Summary of prefetching approaches

	Architecture	Advantages	Disadvantages
Client-side	• Effectively is part of the browser	• Devoted entirely to one user • Can be very fast • Can cache requests from multiple servers	• Requires browser code modification or plug-in • Can increase server load and demand bandwidth without user benefit
Proxy	• Sits in the middle between the server and the browser	• Usually devoted to a group of users with similar interests • Can cache requests from multiple servers • Can be built into a hierarchy	• Can increase server load and demand bandwidth without user benefit • Cache coherency protocol may become very complicated. Additional messaging is required
Server-side	• Part of the server	• No increase in bandwidth demand • Simple cache coherency protocol • Known and limited number of potential pages to cache	• Increase in server complexity • No easy way to track user patterns/document popularity across multiple servers

presented by Jiang et al. in [5]. 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 the client-side prefetching approaches is expensive. Cache coherency deals with the following issue. If a file in cache has changed on the server the new version of the file needs to be presented to the user instead of the stale cached version. This requires checking with the server on the state of the file(s) in the cache (possibly through a special protocol). As a result there is an increased complexity on the client and the server side, as well as increased traffic between the two.

2.2.2. ***Proxy prefetching.*** The proxy-based prefetching approach uses an intermediate cache between the server and a client [7]. 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. This overhead gets even more significant when multiple levels of proxy caches are employed.

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 a 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. By the same token a proxy cache dedicated to a particular organization will do a good job following that organization's preferences. Another advantage is that requests from multiple servers can be cached.

2.2.3. Server-side prefetching. 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 straightforward. 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 significant in terms of wasted bandwidth. In the case of server side schemes there is no complicated protocol and no extra messaging outside the server. As the file system is either local or mounted, in this case, 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 efficient and easy to use mechanisms to check file attributes such as creation and modification times and dates. This assists in maintaining cache coherency. Another advantage of the server side schemes is that, while client side schemes make decisions on which files to prefetch based on the particular user's preferences, the server side prefetching makes decisions 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 [10]. 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 [11] uses Artificial Intelligence related techniques to predict user requests. They implement a learning algorithm, such as a variation of Markov chains, and use a previous access log 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. Finally, this approach 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 clustered 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 (or pages) are text files containing HTML [4] syntax, and tend to be relatively small in size. A key feature of HTML is the ability to embed links to other Web pages. As a result, there is a good chance that each Web page that a user views contains links to other Web pages. Unless the user is not interested in the subject or does not want to surf further he or she is likely to click on one of the links and request another file from a server. From this point of view each page can be represented by a node in a directed graph and each URL link in that page is an arc to another node. 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 [9] search engine.

3.2. Page Rank algorithm

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.

The prefetching scheme presented in the paper addresses server side prefetching and so is applied to pages on the server of interest. Note that in this context the server is any collection of entities serving related Web pages and may actually be a stand-alone machine or cluster of machines. While with the proper protocols and communication infrastructure we can put any number of the traditionally defined Web servers under the “single server” umbrella for our purposes, we do need to limit our universe. If we considered every link we could end up processing the entire Web graph. This is not necessary for our prefetching application where we need to make a decision which pages on the local server to prefetch.

Once the matrix has been populated, Page Rank 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 presented below.

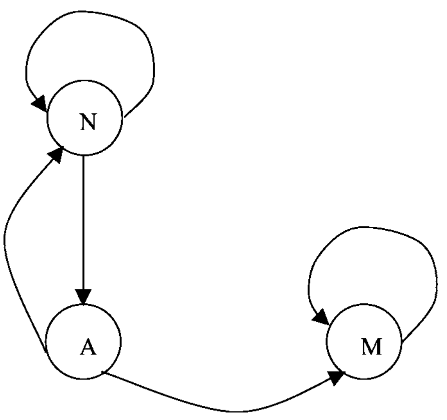

Figure 1. Example graph of a cluster.

To illustrate the Page Rank algorithm used for prefetching, consider the Web page graph shown in Figure 1. This graph shows a cluster of three pages, A, M and N; A is linked to N and M, N is linked to A and to itself, M is linked only to itself.

In this case, the equation to be solved to compute the Page Rank is as follows:

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

As can be seen from the graph, N links to A and to itself. Hence, the first column (column of node N) has 1/2's in rows corresponding to N and A. M links only to itself. Therefore, the second column has 1 in M's row. By the same token A has links to M and A producing corresponding arrangement in its column.

The solution of this equation is computed iteratively until it converges. Convergence here means the difference between the *norm* of the current resulting vector and that from the previous iteration is less than a small threshold (Δ). Notation $|\mathbf{R}|_1$ shows the *first norm* of vector **R**. The first norm (1-*norm*) is defined as follows: $|\mathbf{R}|_1 = \sum_{i=1}^{n} r_i$. It is used to describe the size of the vector.

Therefore, if **M** is the matrix and **R** is the $[n, m, a]$ vector, the following algorithm is executed:

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

The number of iterations for the solution to converge in our experiments was typically less than 20. For the above example, the solution of the 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 requested pages 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.

1. 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.
2. 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.
3. 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.

3.3.1. Web page clusters. 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. The Page Rank scheme excels in prefetching pages in these clusters. A separate Page Rank calculation is performed for each cluster. As soon as the server determines that a requested page belongs to a cluster, it is scheduled for the Page Rank calculation. In our implementation 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 greater chance that these files are related and are interlinked, and their hierarchies are sufficiently wide and deep. A more sophisticated approach would be to run a spider that crawls all the pages on the server and discovers the clusters based on some optimal criteria of the width and depth of the page linkage graph.

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.

3.4. Computational 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 newly requested page, find all the pages it links to and all the pages that link to it. A length n array is used to help keep track of pages in memory. Let n be the number of links on a page. Our observations show that it is rare for a page to have more than 20 links to pages on the same server.
   - 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. We can safely make an assumption that n is the maximum number of links on a page. Then the worst case performance is $O(n^2)$.
   - 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, that n is the maximum number of links on a page, we have a worst case performance of $O(n^2)$.
   - The two operations above are consecutive and can be combined into one with the same $O(n^2)$ complexity. Furthermore, ordering the array would not change the worst-case performance.
   - Recalculate the matrix values. This has an $O(n^2)$ complexity as well.
2. Matrix multiplication.
   - Iterative matrix-vector multiplication and addition. This typically converges in less than 20 iterations.
   - The cost of multiplying an $n \times m$ matrix by an $m \times p$ matrix is $O(nmp)$. We have $n \times n$ by $n \times 1$, therefore, our multiplication algorithm's cost is $O(n^2)$.

As a result, we have the overall complexity of the Page Rank prefetching algorithm as $O(n^2)$.

Note that for $n = 200$ a single-threaded implementation processed 90 requests per minute on an 850 MHz PIII with 256 MB RAM running Windows 2000. This is equivalent to serving a month worth of requests in less than 5 hours.

Hardware costs, and memory costs in particular, have been decreasing rapidly. It is not uncommon for a large company with substantial Web presence (such as Schwabb) to have a dedicated Web server farm of multiprocessor machines with more than 1 GB or RAM each. Even desktop PCs come with 256 RAM standard at this point. If the computer resources are really an issue it should be possible, in theory, to optimize the algorithm using various sparse matrix techniques.

3.5. Design and implementation of a cluster based prefetching server

3.5.1. Design issues

3.5.1.1. Parallelism. A key motivation for implementing the server on a cluster of machines was to exploit the inherent parallelism in the Page Rank prefetching algorithm and maintain server scalability. Page Rank computations for different page clusters can be performed in parallel each on its own dedicated machine. Furthermore, the associated matrix computations can also be parallelized. This introduces a new level of parallelization that is not bounded by the number of page clusters. Both levels of parallelization can be employed simultaneously to achieve maximum performance gain.

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 communication overheads. A single server would have processed the requests sequentially resulting in an overall runtime equal to the sum of the computation times for each cluster.

3.5.1.2. Matrix size. Our experiment showed that a matrix size $n = 200$ resulted in the most appropriate balance between speed and effectiveness of prefetching. 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.

3.5.1.3. Cache organization. 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 were too low wasted computational effort. For example, we found that prefetching a fraction of the pages 0.5 and higher did little to increase the hit rate but caused a lot of files that were never used to reside in the cache only to be replaced later. On the other hand, values less than 0.1 produced a marked decrease in the hit rate.

3.5.2. 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 extra optimizations such as any optimizations for the matrix multiplication algorithm. It additionally maintained runtime statistics (i.e. hit rate). The architecture of the server is shown in Figure 2. The main components of the server are

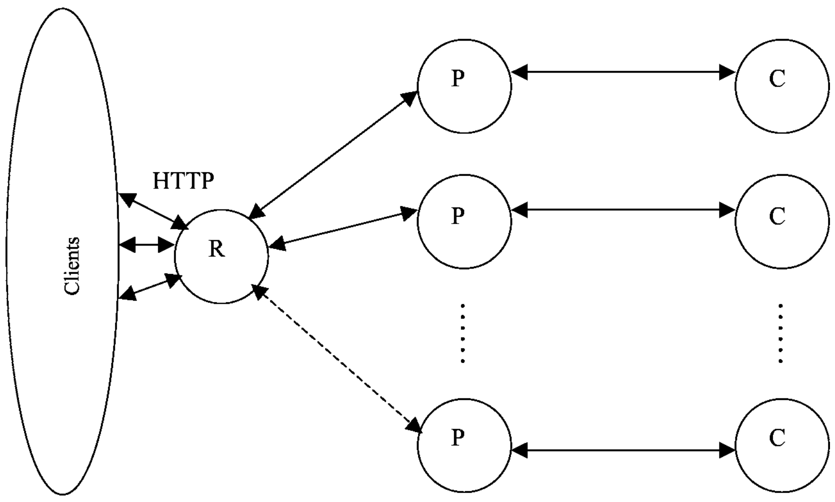

Figure 2. Server architecture.

the Router (R), and the 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. 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 message trace diagram is illustrated in Figure 3. Both the Router and the Prefetcher are multi-threaded for further efficiency. The internal structure and a few details of operations for each component are given below.

3.5.2.1. HTTP handler (Basic Server). This component performs the functions of a regular HTTP server with caching and custom prefetching. It could be used as stand-alone simple Web server. The HTTP handler operates as follows. When it receives an HTTP request it parses it to get the file name and checks if the file is in the cache. If so it verifies it is the file's freshness using a simple timestamp check. If the file is not in the cache or is stale it is fetched from the disk. The handler then formats a proper HTTP reply and sends it to the requesting object (prefetcher in our case). Also, when the HTTP handler receives a prefetch request from the Prefetcher it will get the files from the disk and put them into its cache. The cache is implemented as a user-level memory cache indexed by the filenames.

3.5.2.2. Prefetcher. Prefetcher is the component in charge of making a decision about which files need to be prefetched. As it passes the response to the client back to the Router it parses it and creates a list of "href" links to local pages in the page. It should be noted that the HTML parser has to be very forgiving. Special provisions have to be made to accept anchors with or without quotes and other attributes. Very few pages were found to follow strict HTML syntax since browsers tend to overlook many HTML syntax errors. The parser also converts relative paths into absolute ones for ease and uniformity of processing. The

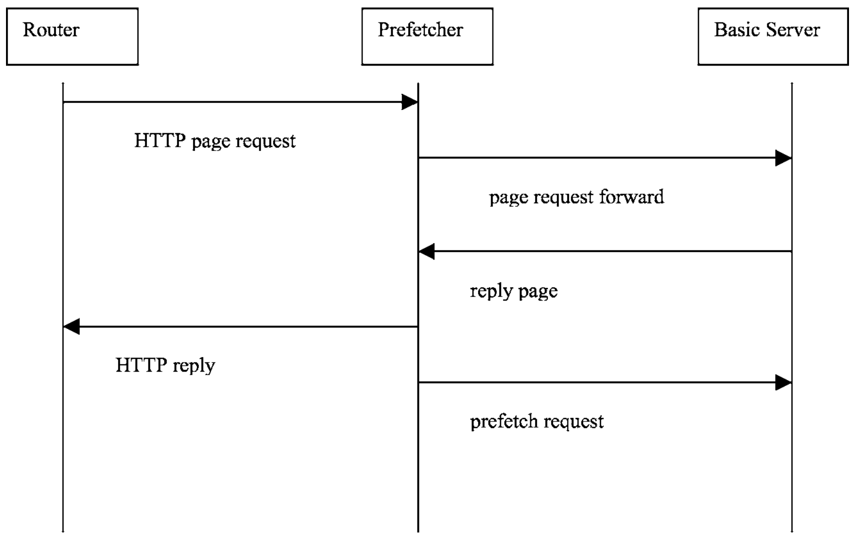

Figure 3. Internal message trace diagram.

resulting list of links, including the link to the current page, is fed into the Page Ranker component. Page Ranker returns a list of new highest-ranking pages. This list is then sent to the HTTP handler to be prefetched into its cache.

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 Prefetcher maintains a running rank of pages on the server based on the pages accessed so far.

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.

To simulate client requests we implemented a simple driver. The driver read server access log, sent an HTTP request corresponding to the original access and waited for the response. This operation is easily parallelizable. We only needed to break the log file into multiple pieces and start the drivers simultaneously for each piece. This simulated multiple clients with repeatable behavior. While the primary objective of our experiment was not to find out how many clients the server could handle multiple clients did speed up the experiment as well as demonstrated our server's scalability.

The objective of this paper is to introduce a new prefetching scheme. We consider server caching to be a subset of prefetching and so do not separate the two. Instead we make a comparison to a random prefetching scheme (also with server caching). In this way we compare two prefetching schemes rather than comparing prefetching and caching. It has also been shown in previous research that on the Web the maximum hit rate achievable by any caching algorithm is just 40 to 50% [5,8]. Our prefetching scheme exceeds this result by a good margin.

4.1. Hit rate

We defined 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

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

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. Figure 4 shows the cluster access for the Berkeley log. Only the clusters with hit rate greater than 10% and with more than a 100 accesses are plotted.

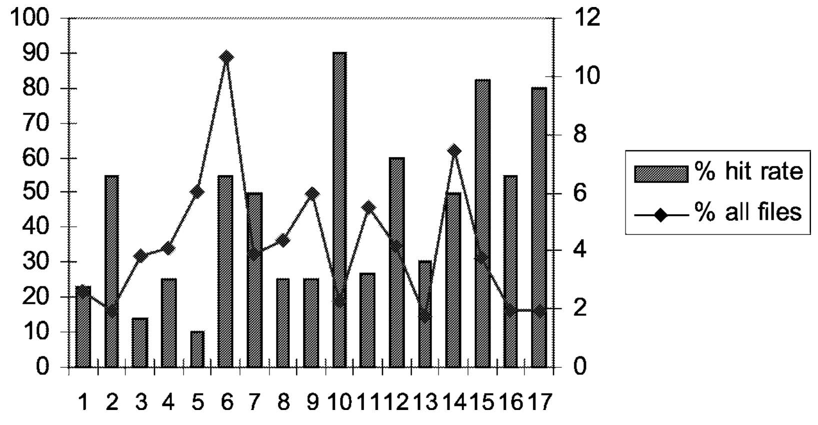

Figure 4. Berkeley clusters and accesses.

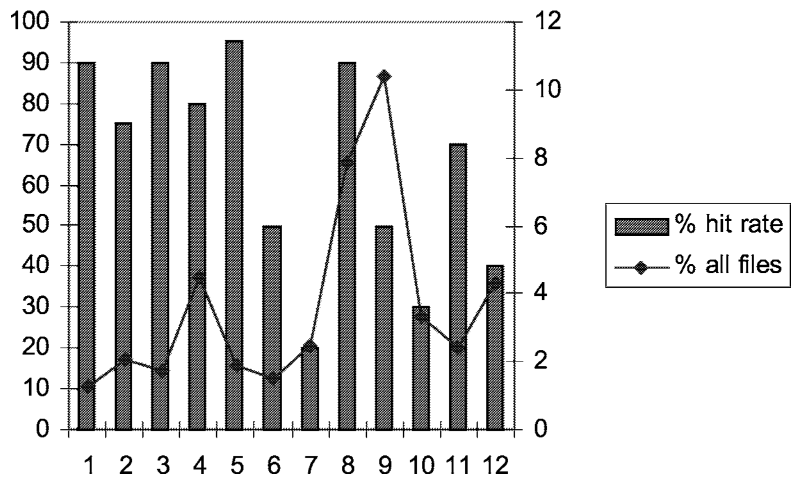

Figure 5. CAIP clusters and accesses.

In Figures 4 and 5 each point on the X-axis represents a cluster in the order of initial access. Left Y-axis represents the % hit rate (for the bars) while right Y-axis represents a % of all files on the server that files in a given cluster constitute (for the line). That is, in Figure 5 cluster 8 achieves 90% hit rate while files in the 8th cluster are about 8% of all the files on the server.

In case of the CAIP server log 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.

Figure 5 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 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 requests. 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.

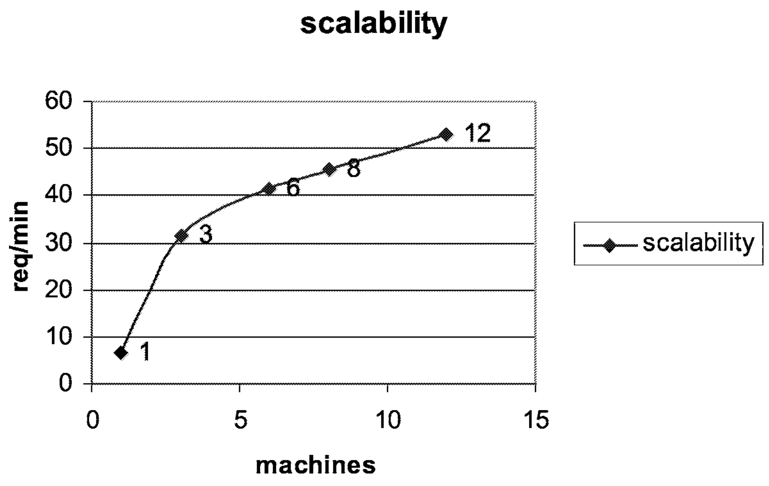

Figure 6. Scalability.

4.2. Server scalability

We ran the scalability part of the experiment on a cluster of identical SUN workstations with 120 MB RAM each. Running with 12 machines in a cluster reduced the overall running time by a factor of 8. Figure 6 demonstrates the scalability results for the CAIP server logs. It demonstrates an almost perfect scalability up to the number of file clusters on the server. This experiment shows that the distributed architecture implemented works very well with the prefetching scheme.

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 the rank to 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. This version will have its matrix calculation parallelized. It can be efficiently deployed in a cluster environment.

References

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

[2] S. Brin and L. Page, "The Page Rank citation ranking: Bringing order to the Web," January 29, 1998.

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

[4] `http://www.w3.org/MarkUp/`

[5] G. Huston, "Telstra Web caching," *The Internet Protocol Journal* 2(3), September 1999.

[6] Z. Jiang and L. Kleinrock, "An adaptive network prefetch scheme," *IEEE Journal on Selected Areas in Communications*, 1998.

[7] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of Web latency reduction from caching and prefetching," in *Proceedings of the First USENIX Symposium on Internet Technologies and Systems*, December 1997, pp. 13–22.

[8] N. Niclausse, Z. Liu, and P. Nain, "A new efficient caching policy for the World Wide Web," in *Proceedings of Workshop on Internet Server Performance (WISP'98)*, Madison, WI, June 1998.

[9] The GOOGLE search engine, `http://www.google.com`

[10] Z. Su, Q. Yang, and Ye Lu Zhang, "What Next: A prediction system for Web requests using N-gram sequence models," *Web Information Systems Engineering*, 2000, 214–221.

[11] I. Zukerman, W. Albrecht, and A. Nicholson, "Predicting user's request on the WWW," in *UM99 – Proceedings of the Seventh International Conference on User Modeling*, 1999.