Personalized Web Prefetching in Mozilla*

Wei Zhang David B. Lewanda Christopher D. Janneck
Brian D. Davison

Department of Computer Science and Engineering
Lehigh University
Bethlehem, PA 18015 USA

Abstract

This paper presents the design and implementation of a Web prefetching module in Mozilla, an open-source and cross-platform browser. We have incorporated two kinds of predictors: a history-based predictor and a content-based predictor. These two predictors can analyze a user's behavior and the contents of recent HTML pages to predict likely next links; thus, they provide personalized predictions which are then utilized to determine which resources should be prefetched.

1 Introduction

Millions of people now browse the Web. However, it is not as fast as people sometimes expect. To improve its performance, people would like to reduce the latency of transferring HTML pages. One approach, namely prefetching [4], is to download them in advance and then store them into the local cache.

This paper explains the design and implementation of a prefetching module in Mozilla [16], an open-source and cross-platform web browser. The basic idea of this module is to analyze the user's behavior and the recently opened Web pages to generate a set of URLs that the user is likely to visit with the next click. The modified browser is able to download these pages

*Technical Report LU-CSE-03-006, Department of Computer Science and Engineering, Lehigh University, Bethlehem, PA, 2003. The research work of the authors was supported in part by the National Science Foundation under grant ANI-9903052. Information about this project can be retrieved from http://wume.cse.lehigh.edu/ or by emailing the last author at davison@cse.lehigh.edu.

and the resources contained in them beforehand, including scripts, images, etc.; then, it stores them into the local cache. If the next visited page is in the cache and is still valid, it will be displayed without delay because the browser doesn't need to contact the remote Web server for these resources again.

However, it is not feasible to download all possible links contained in the current HTML page because it obviously wastes much bandwidth and cannot be done within a reasonable time. So some mechanism to determine which links to fetch is required. In our system, we term this mechanism a predictor, as we wish to predict the resources that the user will retrieve next. We have implemented two kinds of predictors: a history-based predictor [7] and a content-based predictor [6]. Ideally, the predictors would predict the next link precisely, but this is not always possible. For instance, a person browsing CNN may type the URL for an entertainment page, which is not visited by the person before and has no apparent connection with the CNN web site. In this case, the predictor will be unable to correctly predict the request.

In addition, the accuracy of the predictors is quite different. The history-based predictor is typically much more accurate than the content-based technique [5]. Therefore, we give higher priority to history-based results. If this history-based predictor is unable to supply enough predictions (currently 5 URLs), we will augment those provided with the results from the content-based predictor. So, the two predictors work adaptively [3]. It is also one of distinguishing features of our implementation of Web prefetching in Mozilla.

The rest of this paper is organized as follows. We first review related work in Section 2. We then state the design of the prefetching module in Section 3. We describe the two predictors in Section 4. The mechanisms of retrieving and caching will be explained in Section 5 and 6. The statistics sub-module will be outlined in Section 7. Finally, we will explore future work and conclude in Section 8.

2 Related Work

While many researchers have abstractly investigated prefetching (e.g., [12, 8, 11]), most have not built working implementations. Klemm designed a prefetching agent called WebCompanion [13] which can estimate the round-trip time to remote servers and prefetch the links with high round-trip time. Additionally, Kokku *et al.* [14] propose "a non-interfering deployable Web prefetching system". This system employs a JavaScript code embedded in

Web pages to retrieve a URL list from hint servers and then prefetch Web objects consulting the URL list. To reduce the interference with Web servers, they actively probe the workload on the server side.

Concurrent to our development work, the open-source development of Mozilla has added mechanisms for link prefetching [10].[1] Browsers based on Mozilla (version 1.2 and above) now support a mechanism to explicitly indicate what contents need to be prefetched by using a specific tag in an HTML page or in an HTTP header. This approach requires content providers to suggest URLs for prefetching. In contrast, our implementation does not depend on the cooperation of content providers, but instead determines its own URLs for prefetching.

3 Design of Web Prefetching Module

In our design, we aim to develop a portable, configurable, and efficient module.

- *Portability*: This module should be easy to port to browsers other than Mozilla or Web systems such as Squid [17], a Web caching proxy.
- *Configurability*: The parameters for the prediction models can be adjusted for a variety of environments.
- *Efficiency*: The module should be able to prefetch data promptly, but meanwhile, do not consume much bandwidth. Users will not notice its action.

We break up the module into three major parts: Prediction, Retrieving and Statistics. Prediction includes two predictors and adaptively generates a set of results for prefetching. Retrieving is responsible for fetching and caching data. The remaining part is statistics that counts the number of visited pages and calculates the hit rate in cache.

To achieve portability, our codes generally employ the standard C/C++ library. They are independent of Mozilla. With slight modifications, this module can be easily integrated into other applications. such as Squid. One exception that must be specific to the application is caching. The details of caching are unique to the applications. Therefore, we designed an interface for caching to make it transparent to this prefetching module. For

[1] See http://www.mozilla.org/projects/netlib/Link_Prefetching_FAQ.html

another application, we just need to replace the caching sub-module instead of rewriting the entire module.

Configurability is mainly related to the history-based predictor, for it needs to maintain a permanent state for each user. The state includes the model parameters, the present context for the user, and a very large hash table. Most of these parameters can be configured by the developer. For instance, the depth of the model tree can be tuned in order to adopt a different situation. By manual modification of these parameters, the predictor may be able to achieve a higher accuracy (leading to a higher cache hit rate and better user performance).

Efficiency is another design goal we expect to attain. The module should download the prefetching resources as fast as possible, but minimize interference with other operations of Mozilla.

The basic ideas of the integration are presented above. In the following sections, we will explain the three parts respectively in additional detail.

4 Predictors

4.1 History-based Predictor

The history-based predictor employs the history information of user's actions (that is, the sequence of requested Web resources) to predict the next likely ones. The major component in this predictor is a PPM (Prediction by Partial Matching) model [1, 18]. So the history-based predictor is also called PPM predictor.

The history-based predictor maintains three data sets: a hash table, model trees, and historical contexts. The hash table translates from URLs to internal numerical identifiers, and is quite large (currently one million entries) to accommodate many URLs. Model trees record user actions, the history information, in a form easily used for prediction. The historical context tracks requests recently made by the user.

Another important function in this predictor is the storage and recovery of the model status. In order to build up a significant history on which to base future predictions, the model needs to be stored between instantiations of the browser. Thus, when Mozilla initially launches, the history model is reconstructed to the status of the last time Mozilla was used. Since the size of the hash table is comparatively large, we separate the storage and recovery routines into a new thread, which does not affect the loading of Mozilla; otherwise, users may experience a longer delay to launch Mozilla.

The PPM model has multiple parameters for developers, including the depth of the model, the size of hash table, etc. We can adjust these parameters in order to get a more accurate hit rate and attain a trade-off between efficiency and cost.

4.2 Content-based Predictor

In contrast to most Web prefetching techniques which rely on previous references to requested objects to determine what to prefetch next, content prediction uses contextual clues based on the user's choices in a Web browsing session to prefetch data. By analyzing the content of the current page and recently viewed pages, the content-based predictor builds a model of a user's interests by preserving the text of previously viewed pages and their relative words, as shown in Figure 1. When a new page is loaded, the words in and around hypertext anchors in the current page are compared with the model to determine the most likely links to be followed by a user.

The implementation of the content-based predictor provides a method to extract top ranked links based on textual similarity between the words surrounding the links in the current page and the words found on the previous pages. By default, up to 20 words before and after a link are used, as well as the text of the previous 5 pages. Each of these variables is configurable, and can be changed to determine if there is better performance with different settings.

The predictor is built on an HTML-parser utility library. Words are stored in bags of words, which are hash tables that store the word and its associated frequency. Each page is represented as a bag of words, and the bags are stored in a queue of finite length. After the queue is filled to its maximum size, the oldest bag of words is removed before adding the next. This page queue must be maintained between calls to the retrieving function. After the words have been extracted, the HTML is again parsed and the hypertext links are extracted, along with the set of words before, in, and after the hypertext anchor. Finally, a textual similarity comparison is performed by building a score based on the sum of matches of a word in a link and the same word in the queue of words collected. The matching URLs are returned in order.

This library implements Porter stemming [15] as an option to reduce words to common stems for more frequent matches. In limited testing, Porter stemming does affect the results, and so by default Porter stemming is used, but is easily disabled.

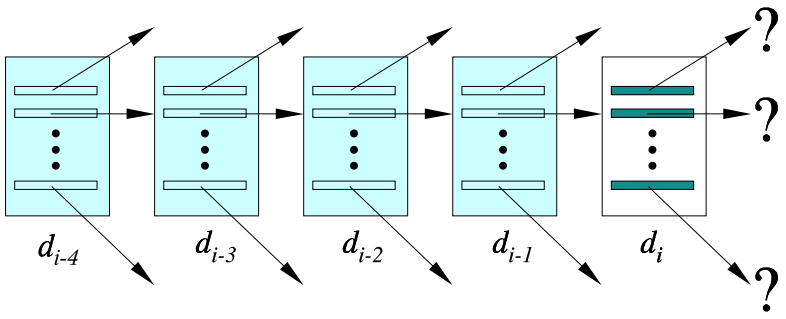

Figure 1: Sequence of HTML documents request. d_{i-n} (n is between 0 and 4) are recently requested pages. Using the correlations of content on these pages to the links of the current page, the model predicts the next likely pages [6].

5 Retrieving

Retrieving is the function to fetch HTML pages and resources contained in HTML pages beforehand, including images, scripts, compressed files, etc. Two designs for retrieving were originally conceived. One is to employ Mozilla's network library[2] to load them. It is easy to do, just like calling a C++ library. However, its drawback is that it is not portable. Codes are tightly combined with Mozilla. When Mozilla evolves, the base classes are often changed. Consequently, we have to update our own codes as well. So we design an alternative, which fetches pages using codes independent of Mozilla. Figure 2 shows its basic mechanism.

As shown in Figure 2, Web prefetching works like a coordinator gluing predictors, connection pool and caching. The codes outside Mozilla are all independent of Mozilla except caching, which will be explained in Section 6. Only several lines of codes are inserted into Mozilla codes to acquire the present URL and the HTML page, which are used by predictors.

A connection pool is the core part in retrieving. It supports multiple simultaneous TCP connections. It is used to create and destroy the connection, and transfer the data via network for the transfer units, which receive pages and pass them to the caching module. All requests will be put into a queue, from which the connection pool picks up one request for an idle socket.

When the data arrives from Web servers, the retrieving sub-module

[2]See http://www.mozilla.org/projects/netlib/

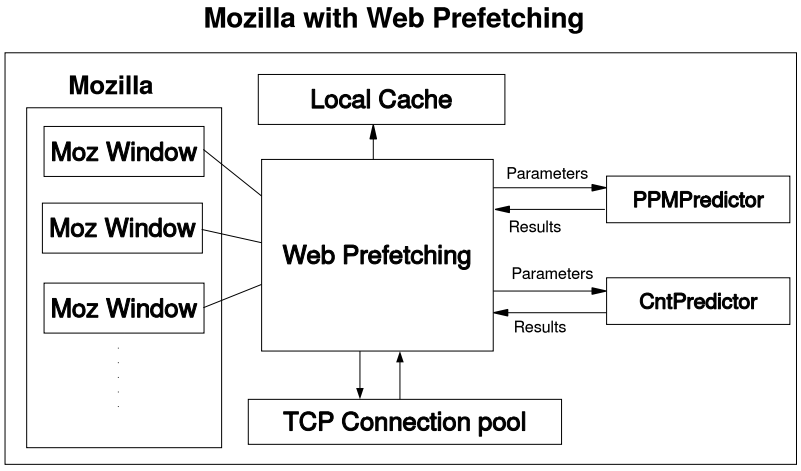

Figure 2: Web Prefetching Architecture. PPMPredictor and CntPredictor are the classes respectively for the history-based predictor and the content-based predictor

checks to determine if it is an HTML page, and if so, extracts all embedded resource links and adds these URLs into the queue for retrieval.

When a user visits a page in Mozilla, the *Mozilla Window*, as shown in Figure 2, will invoke the Web prefetching mechanism. It will pass the current URL and the HTML content of the page to the Web prefetching coordinator. The coordinator employs the history-based predictor to generate a set of URLs. If the number of returned URLs doesn't meet the requirement (currently, 5 URLs), it will ask the content-based predictor for additional URLs. All of these returned URLs are put into a downloading queue. The connection pool can help download the URLs in the queue. Finally, the downloaded content will be stored in the local cache based on the Mozilla caching policy.

6 Caching

Caching is the final step of Web prefetching. It stores the HTML pages and associated resources into the local cache, which is Mozilla's cache in this case. Unlike the other parts of the Web prefetching module, caching is unable to be independent of Mozilla. We have to use the cache library included in Mozilla network library to achieve caching. So, it is specific to

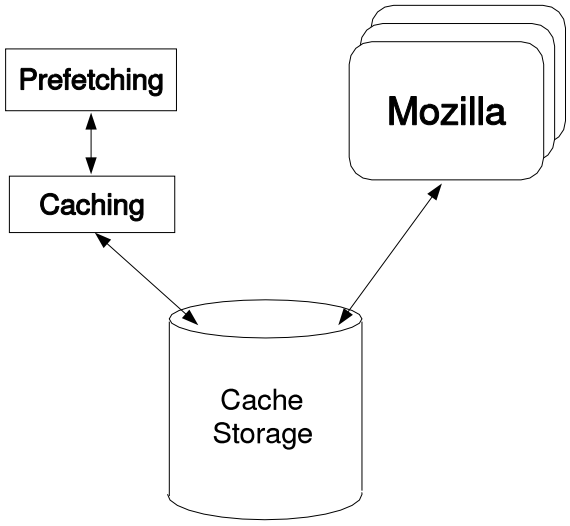

Figure 3: Caching Mechanism. The same cache storage is shared between Web prefetching and Mozilla. Mozilla also caches its downloaded resources. So before caching the prefetched objects, they will be checked by its cache key.

the Mozilla browser. In order to attain the portability of the entire module, we define a generic class for the caching object. Any specific caching object derives from it. The working mechanism is demonstrated in Figure 3.

7 Statistics

To evaluate the accuracy of the predictors, we implement a statistics submodule. It can count three kinds of data. They are the number of all objects received (N), the number of objects in cache (N_c), and the number of objects prefetched and cached (N_p). To distinguish prefetched objects in the cache from those retrieved by Mozilla itself, we mark the prefetched objects with a tag, named "prefetching-cache". By checking this tag, the statistics submodule can properly count the number objects that were downloaded by the prefetching system.

Once an object has been served from cache, we reset this flag as the object would have been fetched at that time, and any subsequent cache hits are not due to prefetching.

Utilizing this statistics, it is convenient to calculate the hit rate in cache

and the hit rate of prefetched objects as follows:

> Hit rate of the cache = N_c / N
> Hit rate of the prefetching = N_p / N

One entry in the trace logs is illustrated as follows:

> *UID(524): Loading URL(http://samba.anu.edu.au/rsync/)*
> *UID(524): Total=175, InCache=50, Prefetched=28, Hit in the prefetched=16.00%, Hit rate of the cache=28.57%*

The first line in this log entry records the user id and the present URL. *Total* in the second line is the number of all downloaded objects. *InCache* is the number of the objects that are in the local cache. *Prefetched* is the number of the objects that are in the local cache and were prefetched. The remaining numbers are hit rates, N_p and N_c respectively.

8 Remaining Issues

As a preliminary implementation, the current version still has some outstanding issues. For example, the connection pool doesn't check whether or not links are in the local cache. It blindly downloads every link in the queue. Therefore, the connection pool should be modified to be able to search in the local cache and find out whether the current link in the queue has been cached. It is not necessary to download the cached entry again.

It also doesn't track the throughput through the prefetching module. The only way employed to limit the downloading rate is to limit the number of concurrent connections, presently set to two. Thus, it may still consume noticable bandwidth and interfere with Mozilla. One possible solution would be to have the connection pool delay requests for a certain amount of time. However, such a change may affect the qualify of service from the prefetching module. So it is quite important to look for a tradeoff between the bandwidth usage and the quality of service.

The prefetching queue is currently unbounded, and thus may grow large, with many old entries. To address this issue, we can employ a priority queue instead of a FIFO queue and set a timestamp for each entry.

Finally, some linked objects are very large. Downloading the entire object may require more overhead (primarily in bandwidth) than is desireable, especially if the file is never actually referenced. On the other hand, even if the object is desired by the user, it may be too large to be prefetched within a reasonable amount of time. Unfortunately, the size of an object is not

known in advance. Thus, we see the need to download only the initial portion (using Range-Request headers in HTTP/1.1 [9] rather than attempting to retrieve the complete object). The size of downloaded objects should be recorded and the downloading module truncates the contents beyond the size limitation, e.g., 100KB.

9 Summary and Future Work

Users often experience latency while surfing Web. We have implemented a method that has been proposed to reduce the latency via Web prefetching. By integrating a Web prefetching module into Mozilla, the average user-perceived latency can be reduced. The prefetching mechanism is automatically invoked after the user visits a page.

In addition to the issues raised above, we note that prefetching does have a cost — not all objects fetched will be used. However, there are other techniques to improve the performance of browsers, such as pre-resolving website hostnames and pre-connecting to the Web servers [2]. We believe these ideas would be welcome improvements to any browser.

References

[1] T. C. Bell, J. G. Cleary, and I. H. Witten. *Text Compression*. Prentice Hall, Englewood Cliffs, NJ, 1990.

[2] E. Cohen and H. Kaplan. Prefetching the means for document transfer: A new approach for reducing Web latency. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, Mar. 2000.

[3] B. D. Davison. Adaptive Web prefetching. In *Proceedings of the 2nd Workshop on Adaptive Systems and User Modeling on the WWW*, pages 105–106, Toronto, May 1999. Position paper. Proceedings published as Computing Science Report 99-07, Dept. of Mathematics and Computing Science, Eindhoven University of Technology.

[4] B. D. Davison. A Web caching primer. *IEEE Internet Computing*, 5(4):38–45, July/August 2001.

[5] B. D. Davison. *The Design and Evaluation of Web Prefetching and Caching Techniques*. PhD thesis, Department of Computer Science, Rutgers University, Oct. 2002.

[6] B. D. Davison. Predicting Web actions from HTML content. In *Proceedings of the The Thirteenth ACM Conference on Hypertext and Hypermedia (HT'02)*, pages 159–168, College Park, MD, June 2002.

[7] B. D. Davison. Learning Web request patterns. In A. Poulovassilis and M. Levene, editors, *Web Dynamics*. Springer, 2004. In press.

[8] L. Fan, Q. Jacobson, P. Cao, and W. Lin. Web prefetching between low-bandwidth clients and proxies: Potential and performance. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '99)*, Atlanta, GA, May 1999.

[9] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, `http://ftp.isi.edu/in-notes/rfc2616.txt`, June 1999.

[10] D. Fisher and G. Saksena. Link prefetching in Mozilla: A server-driven approach. In *Proceedings of the Eighth International Workshop on Web Content Caching and Distribution*, Hawthorne, NY, Sept. 2003.

[11] D. Foygel and D. Strelow. Reducing Web latency with hierarchical cache-based prefetching. In *Proceedings of the International Workshop on Scalable Web Services (in conjunction with ICPP'00)*, Toronto, Aug. 2000.

[12] Z. Jiang and L. Kleinrock. An adaptive network prefetch scheme. *IEEE Journal on Selected Areas in Communications*, 16(3):358–368, Apr. 1998.

[13] R. P. Klemm. WebCompanion: A friendly client-side Web prefetching agent. *IEEE Transactions on Knowledge and Data Engineering*, 11(4):577–594, July/August 1999.

[14] R. Kokku, P. Yalagandula, A. Venkataramani, and M. Dahlin. NPS: A non-interfering deployable Web prefetching system. In *Proceedings of the 4th USENIX Symposium on Internet Technologies and Systems (USITS 2003)*, Seattle, Mar. 2003.

[15] M. F. Porter. An algorithm for suffix stripping. In K. Sparck Jones and P. Willet, editors, *Readings in Information Retrieval*. Morgan Kaufmann, San Francisco, 1997. Originally published in *Program*, 14(3):130-137 (1980).

[16] The Mozilla Organization. Mozilla.org home page. http://www.mozilla.org/, 2003.

[17] D. Wessels. Squid Web proxy cache. Available at `http://www.squid-cache.org/`, 2003.

[18] I. H. Witten, A. Moffat, and T. C. Bell. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann, San Francisco, 1999. Second edition.