Prefetching in a Web Browser

Wendy Chien

6.033 3/98

Karger/Kwon TR1

Abstract

Prefetching is a way to speed Web browsing by downloading pages linked to the displayed page and storing them in the cache in case the user follows one of those links. This method is only effective if enough of the prefetched pages are later chosen by the user and if few enough prefetch requests are made so that the network does not become congested. An algorithm which randomly chose a constant number of links to follow and used multiple threads to request those links was selected to produce the most future hits. The constant number of links to follow is determined by measuring the change in data transfer rate to ensure the load on the network is reasonable.

1 Introduction

Exploring pages on the Web often takes a long time because of the time it takes for the browser to retrieve data from a server. In general, a cache is often used to enhance performance of data access. A cache is a small but fast to access place in memory compared to the rest of memory which is often large but takes a long time to access. A cache stores data which hopefully will be accessed soon in the future. Deciding what to put in the cache usually falls under the two theories of temporal locality and spatial locality. The idea of temporal locality is that data which has been accessed recently will probably be accessed again soon. Spatial locality states that data related to recently accessed data will probably also be accessed soon in the future.

Caches can be applied at several different levels and in several different applications, including a web browser. To speed up the retrieval time of a web page, web browsers currently base their ideas on temporal locality and use caching to store in local memory the pages that the user has recently visited. Therefore, if the user wants to access a page which happens to be in the cache, the browser will get the page from memory instead of from the original server and thus saving a lot of time. To further enhance performance, the browser designers want to employ the idea of spatial locality and also use prefetching in the implementation of the web browser. Instead of just caching pages that the user has already visited, prefetching also predicts the user will visit sites referenced by the sites he or she has visited and stores those pages in memory as well. So when the user visits one of those sites, the page will already be stored in local memory and can be displayed faster.

There are many considerations when trying to modify a web browser to allow for prefetch-

ing, including deciding how to balance maximizing the effectiveness of prefetching while minimizing the extra load placed on the network and server and how the end product will change from the user's point of view.

2 Design Considerations

The issues in adding prefetching capability can grouped into the three main categories of effectiveness of prefetching, traffic on the network and load on the servers, and transparency to the user.

2.1 Effectiveness of Prefetching

Maximizing the effectiveness of prefetching is the main goal of this design. Most design decisions will lead back to how the prefetching performance will be affected. The main issues which directly determing the effectiveness are the use of multiple threads, the actual prefetching algorithm, and the cache replacement strategy.

2.2 Traffic on Network and Load on Servers

If a large number of prefetch requests are made, then the network will become congested and Web accesses will become very slow. Therefore if prefetching is to be effective, the number of prefetches made has to be limited and the number of unnecessary requests must be kept at a minimum. Issues which involve controlling the excess traffic on the network and load on the server include stopping a prefetch if it becomes unnecessary, and measuring the performance of the connection and server to determine the number of prefetch requests to send.

2.3 Transparency to the User

The prefetching mechanism should mostly be running in the background and not complicate the interface. Nevertheless, the user need not be completely ignorant of the browser's prefetching. In fact, the user might wish to input his or her opinion on the number of prefetched pages to store or whether to be notified of pages that are being prefetched. The main design issues under this concept are what options the user has in regard to prefetching, how the interface will be modified, and what visual cues will be made available to the user.

3 Possible Implementations

3.1 Prefetching Strategies

3.1.1 Web Pages

Web pages can have several different components, but can be described as being comprised of text and non-text items. The text is transferred first, and then each of the other components are downloaded separately. In general, the text is the main part of the page but is still quick to transfer. Non-text items are usually very large and require a lot of time to transfer. The idea to only prefetch the text portion of the page is born out of these two characteristics. Text-only prefetches take less time to transfer and take up less space in the cache. The disadvantage is if the page is accessed, the browser will need to connect again with the server to get the pictures on the page. Prefetching the whole page would enable the whole page to be displayed faster if it is accessed, but would require more time to download and more space in the cache.

3.1.2 Threads

Prefetching pages can be done either using a single thread or using multiple threads. Under both strategies, the threads handling the prefetch requests should be given the lowest priority of all the threads so it does not interfere with the regular activities of the browser. Beyond this, their implementations diverge.

If only a single thread is used to prefetch then, it would get all the pages serially. Given a set of links to access, the thread would group together the pages that are found on the same server. It then opens a connection with the server which holds the most references found in the set and begins to download the pages. When a page is transferred it inserts the page into the cache. The thread will continue to run until the set of links to access becomes empty or it is told to terminate. The main advantage of this method is the simplicity. With only one thread, there is little overhead and fewer concurrency issues. Another advantage is that because only one file is being transferred, it does not have to share the bandwidth other files and will have the highest data transfer rate possible. The importance of a full quick transfer becomes more apparent in section 3.2.1.

Using multiple threads employs a different strategy. One method to use multiple threads would be to assign each thread to a prefetch request. Each thread would receive the URL of a link, open a connection to the appropriate server, send a request and wait for the response. Because all

requests are sent out simultaneously, links on the same server will not be accessed by the same thread. However to ameliorate this problem, it is possible to group the links by server, and instead of assigning one thread per link, each thread is assigned to a server and manages all the requests to that server. Again each thread will open a connection to their server and send requests for pages. The threads that are assigned to servers which hold multiple links selected for access will request those pages serially. When the transfer of a page is complete, it is inserted in the cache. The prefetching threads should have equal priority among themselves in terms of scheduling and execution time. All threads will continue to run until they have processed all links assigned to them or they are called to stop executing. If a thread is in the middle of a transfer, then any information received is discarded and not stored in the cache. The main benefit to using multiple threads is maximizing the bandwidth if the browser's connection is fast enough to handle more than one request at a time. (More detail on bandwidth and its effects on multiple requests is given in section 3.2.1.) Another benefit is that much of the idle time that occurs while waiting for a connection can be spent processing another request so the browser's productivity is increased. However, multithreading also creates several complexities including the overhead of managing the threads and the concurrency problems. Because many threads will be writing to the cache, locking the cache before making any modifications is important in ensuring its integrity. Despite these drawbacks, multithreading the prefetching adds a great deal of functionality by increasing the number of prefetches that are made.

3.1.3 Prefetching Algorithm

There exist many different strategies in deciding which links to prefetch, but they all share several characteristics. First off, any strategy will eliminate the types of pages that the browser definitely should not prefetch. The first kind of page is one which contains time critical data. These pages can be recognized by the "?" in their URL. The second type is a page that is already in the cache either from an access or a prefetch. Before attempting a prefetch the browser must check that the page is not already in memory. A prefetching algorithm should only deal with choosing which links to prefetch among the remaining eligible links on the current page.

One method would be to fetch all the pages referenced by the current page. While this idea is simple and effective, it is also unreasonable because this strategy would fail for pages with many links. If the browser tries to prefetch a large number of links at once it would create too much traffic, and transferring a file will take a long time because all the files are sharing the bandwidth.

The rest of the algorithms discussed here revolve around the issues of how many links to follow and which links on the current page to follow. A feasible idea is to select a constant, k, number of links to follow, which is dependent on the bandwidth. The exact values for k for different bandwidths can be calculated empirically but it should be less than the bandwidth divided by the average size of a page. Determining which links to prefetch is more difficult. There is little information available about where the user might go. A method to predict where the user will go is to scan the recently visited sites to find a common topic and then check if any of the links on the current page also contain information concerning that topic. This idea would use an immense amount of time and resources and would not necessarily even be effective. Another idea is to follow the window of view and pick links that are currently being viewed by the user. This strategy raises questions of when to prefetch the k links and if it will have enough time to prefetch a page before the user clicks on it if it does not begin prefetching until it comes in the view of the user. Also if the user is scrolling down the page, there will be no constant view so the browser will just be picking k links from all the links on the page. A final idea is to see which pages in the browser's history are referenced by the current page and to randomly choose k links from those pages. If there are not at least k links which also appear in the history, then the browser should choose all the ones that are in the history and then randomly select the rest of the k sites from the remaining links on the page. Because the browser has so little information about where the user wants to go, a random selection of sites will probably be as successful as any other algorithm.

3.1.4 Cache Replacement Strategy

The first choice in designing the cache comes in how to handle prefetched pages in conjunction with visited pages. There are three main implementations for a cache which holds both types of pages, one which ignores the difference in type of the pages, one that uses the type in creating a special replacement strategy, and one that separates the two types of pages into smaller sub-caches.

Using one cache and ignoring the difference between accessed pages and prefetched pages is the simplest of the three designs. All pages retrieved by threads would be stored in the same cache. This concept is very modular because the cache can operate in the same manner as before. However, because it is running the same replacement strategy on both types of pages, it is only taking into account the size of the file and the date the file was last accessed or put in the cache. Therefore the browser inaccurately assumes that the user is equally likely to go to a page that he has visited before

or a page that was only referenced by a page the user visited.

A second design calls for acknowledging the difference between the two types of files and changing the replacement strategy accordingly. The simplest strategy under this design would be to prefer one type of page over the other in all conditions. If accessed pages were always preferred over prefetched pages then once the cache became full and pages needed to be replaced, only prefetched pages would be flushed. When enough pages have been accessed there will be no prefetched pages left in the cache and prefetching would result in no changes in performance. A more complicated strategy which juggles the three parameters of size, date, and type needs to be devised. To create such a strategy, a judgment needs to be made on the relative importance of the two types of files.

Because it is hard to judge which kind of page should hold priority over the other, another solution would be to create two caches, one for the prefetched pages and one for the visited pages. The cache for the visited page is the original browser cache and the prefetched page cache stores only pages that are prefetched. This design requires more administration than the previous two. When a prefetched page is accessed by the user, the page must be inserted into the accessed page cache and deleted from the prefetched page cache. Also, when executing a lookup on the entire cache, two searches must be done instead of one. In this design, the difference between the two types of pages is acknowledged but no preference is given to one or the other and the replacement strategies remain simple. In fact, the replacement strategy of the prefetched page cache can be different from the accessed page cache. The replacement strategy of the prefetched page cache can also be changed without affected the other cache.

The replacement strategy for the prefetched pages should be least recently used (LRU). Storing the prefetched pages even after the user moves beyond the page where they are referenced is useful in case the user decides to hit the back button and return to a page to follow more links. Because the user will most likely follow links that are closely related to topic he or she has more recently visited, the pages with the most recent storing dates should stay in the cachel. There is, however, a case where LRU will not perform as well as well as the most recently used (MRU) strategy. If the user is going back several links to previously visited pages, then he will probably not go to pages he has most recently prefetched. Instead, he will follow links from pages he visited a long time ago and to which he has returned. In this case MRU will be the most helpful because it will keep the older prefetched pages and discard the newer ones, so the user will still have the older pages at his dis-

posal. Unfortunately, the browser cannot predict whether the user will next go forward or backwards so it cannot dynamically switch replacement strategies.

3.2 Network Concerns

3.2.1 Measuring Performance

Knowing the performance of the browser's connection or the load on the server is important in determining how many prefetch requests to make. However, calculating the performance and figuring out where a bottleneck occurs is a complicated task.

Different approaches should be taken depending if the bottleneck occurs on the browser side or the server side. If the browser's connection is causing the problem, then adding another prefetch request will cause the transfers to slow down significantly. If it is the server, then an additional prefetch request to a different server would not slow the rate of transfer by much. For the browser to find out which end is the bottleneck requires calculating a lower bound for its own bandwidth and measuring the rate of data coming from a server to see if the server is transferring data at a slower rate than the browser can receive. If so then the server is likely causing the delay. If not, then the browser cannot conclude that their connection must be slower because there are many other variables on the network that could cause such a delay. The browser would need to store the data rates and make several calculations to make a guess as to what is causing the slowness on the net.

The simple approach would be to keep track of the rate of transfer of the prefetches, If this rate decreases then the browser should also lower the number of prefetches. If the rate increases, then more prefetch requests can be added. The browser would only have to store the average bits per second rate that the last three sets of prefetches used and compare it to the current rate. This will probably not maximize the use of the bandwidth and be slightly less efficient than the first method, but it is easier to implement and requires less complicated tasks.

3.2.2 Prefetch Stopping

A prefetch should be stopped when the user clicks on a link and leaves that page. There are several reasons why continuing to prefetch when the user leaves the page will cause complications and degradation in performance. In the case that the link the user chooses is not one of the pages in the cache, the browser will have to share bandwidth with prefetching links from the previous page. This will slow the transfer and the degradation in performance affects the page the user is waiting to

see. Because the number of pages that are being prefetched is dependent on how much the browser thinks would be a good transfer rate for each file, the addition of an unexpected fetch will likely cause a significant decrease in transfer rate. Even if the additional fetch did not affect performance, the complications of coordinating the threads prefetching from the previous page and calculating how many threads are remaining in order to determine how many new threads to fork requires significant effort. It would much more tidy for all the threads to stop and clean up before moving onto the next page.

Stopping a prefetch can be implemented like stopping a normal fetch. Because the ability to stop a transfer already exists, the prefetch stop can use the existing implementation for a normal fetch stop. If the current implementation asks the thread to check to see if the stop button has been pushed, the modified version can ask each of the threads to check if a the user has clicked on another site which will set a boolean flag to true. If the flag is true, then the threads will clean up and return. Or the current implementation could be that the thread is sent a message to terminate so it can clean up and return. If this is the case, then in the modified version, the threads managing the prefetches will be sent a similar message. In either case, the old implementation can be used.

3.3 User Interface

The prefetching mechanism can be completely transparent to the user and not require any modifications to the current interface or it can interact with the user using various kinds of visual cues. Because prefetching pages is solely for performance purposes it is unnecessary that the user know that it is implemented at all. However, there is no harm in having the user be aware of prefetching occurring and it might actually be helpful for the user to input his/her preferences for prefetching. Supplying a prefetching options menu would allow the user to decide whether or not he or she wanted to see the activities of the prefetching. If the prefetching is hidden, then no modification is made to the current browser interface. However, if the user is interested in seeing the prefetching then several options can be made available.

Included in the prefetching options menu would be all the other available options for prefetching. The most basic of the options would be to allow the user to turn off prefetching completely. This would be useful if the user noticed a degradation in performance due to the prefetching. A second option for the user would be to set the size of the prefetched page cache. Just as Netscape allows the user to determine how much memory should be devoted to ordinary caching, the modified

web browser can also allow the user to define the amount of memory he or she wishes to allocate for storing prefetched pages.

In addition to several options, the user can also see in the window the results of prefetching. A simple change to the interface would be to highlight the link of a page that has been prefetched. If the user means to look at several of the links but is indifferent to which to visit because they are all examples of the same thing, then indicating which pages have been prefetched would encourage the user to follow those links and increase the probability that one of the prefetched pages will be accessed next.

4 Recommendations

The main issues driving these recommendations are optimizing the prefetching algorithm while keeping the number of prefetch requests to a minimum. Most of the decisions came down to either a trade-off or a compromise between simplicity and functionality.

Only the text portion of a page should be prefetched. The advantages of only transferring the text come in time and space efficiency. The size of the text is relatively small compared to pictures so it takes less time to transfer and also requires less space in the cache. Text is also generally the most important part of the page so having the text available faster is more important than having the pictures faster. The advantage of prefetching the pictures as well is small because users are accustomed to waiting for graphics. However, a considerable amount of time is required to download them and a large amount of space is needed to accommodate them in the cache. These disadvantages combined with the fact that the prefetched page might not be accessed outweigh the advantages of transferring the whole file.

Multiple threads should be implemented in order to increase the capability of the prefetching routine. In choosing between using a single thread or multiple threads to manage the prefetch requests, the simplicity versus functionality trade-off arises. Dealing with only a single thread prefetching pages is much simpler because the browser does not need to deal with scheduling threads and locking critical data structures. Because the design is simpler, the code will be easier to write as well as debug. However, using only one thread is very limiting in terms of functionality. Unless the browsers connection is the limiting factor in the transfer rate, it should be able to fetch more than one page without noticeably decreasing that rate. By increasing the number of pages that are prefetched, the chances of the user picking one of the prefetched pages increases. Also, prefetching multiple

pages simultaneously utilizes the browser's idle time and increases productivity. If, however, the browser's connection is slow, the browser will decrease the number of prefetches made to 1 so it will not slow down the rate of transfer. Though simplicity is important in design, in this case the functionality added as a result of using multiple threads outweighs it.

The prefetching algorithm should randomly prefetch a constant number of references on the current page which are also in the browser's history. The selection of the best algorithm was also a decision between simplicity and functionality, however in this case, simplicity was chosen over potential functionality. Complex algorithms such as following the window of view or using various artificial intelligence techniques to predict the next site require intense computation and time, yet do not necessarily yield better results. Predicting the whims of a user is nearly impossible, so the attempt is not worthwhile. Also the time required to run such an algorithm will subtract from the time available in making the prefetches, lowering the probability that the prefetches will be cached before the user is ready visit another page. The only convenient information available to the browser is the history, which can be can be easily used to help guess the next site. Randomly choosing from the links in the history will likely result in the same outcome as any fancy algorithm.

Separating the accessed page cache and the prefetched page cache is the most effective set-up in managing the cached pages. This decision comes as a compromise between simplicity and functionality. The simple design which ignores the difference between the accessed pages and the prefetched pages and uses an ordinary replacement policy fails to optimally replace pages because accessed pages should be given more weight. A replacement strategy which distinguishes between the two types of pages in the same cache is effective but too complex. A combination of these ideas results in a solution which is both simple and effective. The two separate caches allow simple replacement strategies to be run on each, while still taking into account the types of the pages.

LRU is the most effective cache replacement strategy for the prefetched page cache. Unlike the other design decisions, this choice is not between simplicity or functionality. Most ordinary replacement strategies would fit in this design, but they would not result in the most cache hits. Choosing the best replacement policy entails knowing the path of links the user intends to take. As this is difficult to predict, the policy used must work the best under the average tendencies of the user. Usually, the user will progress forward with the links with only a few backtracks. These actions are best supported by LRU.

In deciding how to determine the performance of the network and servers, simplicity and

effectiveness were compared. To gauge where congestion occurs between the server and the browser requires many computations and comparisons of transfer rates of pages from different servers. A simpler method of only using the change in data received per unit time removes much of the complexity but would also not fully take advantage of the available bandwidth. The small gain in performance using the former design is not important enough to sacrifice simplicity especially in the initial design.

A prefetch request should be stopped when the user goes to a different page. Two benefits of doing so are efficiency and modularity. If the prefetches are stopped, then they will not impede the transfer of data for the current page and they will also be able to clean up and terminate so that all the effects of prefetching for the previous page are done by the time the new page is loaded. Because a stop feature is already implemented in the browser, it can be used to stop a prefetch as well.

The user interface should not undergo any major changes. An extra section under the preferences choice should list the prefetching options have been added to allow the user to decide how visible the prefetching should be. Because most of the people who will use the web browser will not care to know the details of prefetching, the default setting should make the prefetching transparent to the user.

If the browser described above is implemented on every computer, the performance of the web will decrease dramatically. Unfortunately, there is no way to implement prefetching in browsers which will not put strain on the Web. Even if each page only generated three prefetch requests and the user follows one of the prefetched links every time, the load on all the components will still be tripled. Thinking realistically, each page will probably require more than three prefetches to be effective and the user will probably not choose one of the prefetched pages every time, so the effect will be much worse. This design should work well if only a few users have prefetching implemented.

Conclusion

The design recommended above balances the two contrasting motivations of maximizing the performance or the prefetching while also minimizing the strain put on the network and servers. Most of the design decisions favor simplicity over functionality to ease the initial implementation. Many of the components of the design which have a large effect on the performance such as the prefetching algorithm, the replacement policy for the prefetched page, and the method of measuring

the performance of the network and servers are modular and can be refined without much change to the rest of the design. This design is intended to provide a good working model which can easily be extended and made more sophisticated.