Journal of information and communication convergence engineering
http://jicce.org

J. Inf. Commun. Converg. Eng. 12(3): 161-167, Sep. 2014 Regular paper

A Review of Web Cache Prefetching

YuFeng Deng and Sathiamoorthy Manoharan*, *Member, KIICE*

Department of Computer Science, University of Auckland, Auckland, New Zealand

Abstract

Web caches help to reduce latencies arising from slow networks through storing and reusing what was used before. Repeat access to a cached resource does not incur network latencies. However, resources that have never been used will not be found in the cache. Cache prefetching is a technique that helps to fill a cache with still-unused resources in anticipation that these resources will be used in the near future. Typically these unused resources are related to the resources that have been accessed in the recent past. While web caching exploits temporal locality, prefetching attempts to exploit spatial locality. Access to the prefetched resources will be cache hits, and therefore reduces the latency as perceived by the user. This paper reviews the cache infrastructure supported by the hypertext transfer protocol and discusses web cache prefetching in general, including Mozilla's prefetching infrastructure. It then classifies and reviews some prefetching techniques.

Index Terms: Prefetching, Web caching, Web server performance

I. INTRODUCTION

Caches keep items of interest locally so that these items can be used locally without having to fetch them from remote sources. These items could be items used in the past (temporal locality) or items that are spatially close to the items used in the past (spatial locality). Caches are widely used in many cases where the latency of obtaining remote items is considerably larger than the local processing time. A CPU cache is a good example of this. CPU caches exploit both temporal and spatial localities in programs.

If the requested item is found in the local cache, it is called a cache hit; otherwise, it is called a miss. An associated metric is the cache hit rate, which is the percentage of requests that hit in the cache. The cache hit rate measures how well the cache is working.

After a cache miss, the requested item needs to be fetched from the remote site because it is not there in the local cache. Prefetching attempts to reduce such cache misses by fetching in advance the items that are anticipated to be used in the future.

Web caching relates to caching resources and responses from a Web request. It focuses on reducing network latency and conserving network bandwidth. A Web cache can be deployed at the client, server, or any intermediaries in between (such as proxies and gateways) [1-3]. All the popular Web browsers cache applicable server responses so that repeat requests can be served locally from the cache.

Unlike CPU caches, Web caches only use temporal locality and cache objects that were accessed in the past. Web prefetching overcomes the limitation of passive caching by proactively retrieving the cache resources or responses for future requests. This requires being able to predict what those future requests will be. Resources and responses for the predicted requests are prefetched and placed in the cache anticipating a near-future use. Prefetching is initiated when the network is deemed lightly used. Once these prefetched resources are in the cache, the

Received 20 February 2014, **Revised** 10 March 2014, **Accepted** 26 May 2014
***Corresponding Author** Sathiamoorthy Manoharan (E-mail: mano@cs.auckland.ac.nz)
Department of Computer Science, University of Auckland, Auckland, New Zealand.

Open Access http://dx.doi.org/10.6109/jicce.2014.12.3.161 print ISSN: 2234-8255 online ISSN: 2234-8883

This is an Open Access article distributed under the terms of the Creative Commons Attribution Non-Commercial License (http://creativecommons.org/li-censes/by-nc/3.0/) which permits unrestricted non-commercial use, distribution, and reproduction in any medium, provided the original work is properly cited.

Copyright ⓒ The Korea Institute of Information and Communication Engineering

cache can serve them instantly on request.

Prefetching does not reduce the data exchange rate; rather, it increases this rate as the prediction algorithms are not 100% accurate and will fetch some unnecessary resources. Similarly, prefetching does not reduce the latency. However, it reduces the user-perceived latency by exploiting the network idle time to download the resources before the user needs them.

Cho stated that prefetching is often more effective in providing a better user experience than increasing the bandwidth [4]. However, prefetching may not work well for congested shared networks because others may be using the network during your idle time [5].

II. GOALS OF PREFETCHING

The primary purpose of prefetching is to reduce the user-perceived latency by caching the resources before the users request them. In the ideal case, a user will use the same resources (CPU, memory, bandwidth usage, etc.) as before but remove all the wait time. In other words, a user gets the online resources instantly as she requests them.

Exploiting spatial locality in the Web is a simple form of prefetching; resources close to the resource currently being consumed are fetched to the cache with the anticipation that these will be used in the near future.

Here is an example of an ideal prefetching scenario where spatial locality is exploited. A person is reading online a 100-page book in the HTML format. It takes him 1 minute to finish reading a page. The browser takes 10 seconds to download a new page. Without prefetching, this person will need to wait for 10 seconds after clicking the next page button, which leads to a total wait time of 1,000 seconds. Suppose that the browser is smart enough to download the next page when the person is reading the current page. When the person clicks the next button, the browser gets the content from its local cache and displays it. There is no wait time at all. It still costs the browser 1,000 seconds to download the 100 pages. However, this is time not seen by the user; the user-perceived latency has been removed. This is prefetching at its best.

In reality, the above example hardly happens. For instance, the person may want to skip a chapter because she is not interested in its content. There is no obvious and exact way to know what someone wants in the near future. Downloading all the links from the current page is not an option. First, there is not enough time for the browser to fetch everything. Modern Web pages contain a lot of links that may point to large multimedia files (movies, music, etc.). The browser will take considerably more time (than the reading time, for example) to fetch all the links. Second, downloading many resources from the Internet will use a significant amount of bandwidth and cause network congestion. Third, many browsers doing the same thing will overload the server. For these reasons, downloading everything in the current page will only increase the user-perceived latency in most cases.

While the primary goal of prefetching is to reduce the user-perceived latency, there are a number of secondary goals that are often as important as the primary goal itself. These secondary goals are as follows: 1) reducing network bandwidth usage, 2) reducing the use of unnecessary local storage, and 3) not increasing the server load.

The basic principle of prefetching is to attempt to satisfy the primary goal as well as the secondary goals to a reasonable extent. What actually is reasonable can be quantified through metrics measuring the prefetching effectiveness.

A. Prefetching Effectiveness

Effectiveness of prefetching can, in part, be measured using the metrics used for measuring the effectiveness of Web caches. The two common metrics used for Web caches are the *hit rate* and the *byte hit rate*. Hit rate is the percentage of requests that hit in the cache. However, this does not tell how much bandwidth or latency has been saved; small files can be hits and large files can be misses. Rather than counting just the requests, the byte hit rate is based on counting the number of bytes of the hits. Cache hits for large objects contribute more to the byte hit ratio than cache hits for small objects. Therefore, the byte hit rate is a measure of how much bandwidth is saved by caching and prefetching.

A prefetch-specific measure is the *prefetch hit rate*; the percentage of the items that are actually used with respect to the number of items that are prefetched. Just like the hit rate above, this prefetch hit rate can further be focused upon by considering the percentage of useful bytes to the total of the prefetched bytes. This measure, the *prefetch byte hit rate*, can help us quantify how suitable the local storage is used for prefetching.

A conservative prefetching may fetch only those objects that are highly likely to be used and thus result in a high prefetch hit rate. However, this will lead to a low cache hit rate. Therefore, the effectiveness of prefetching needs to be measured by both hit rates.

III. PREFETCHING INFRASTRUCTURE

A large part of the prefetching infrastructure is based on the Web caching infrastructure provided by the hypertext transfer protocol (HTTP) [2]. To this end, this section reviews both the Web caching infrastructure and the prefetching infrastructure.

A. HTTP Caching Infrastructure

HTTP caching is supported by the entities in the HTTP request-response pipeline—browsers, caching proxies, and gateways. Browser caches are normally private (per user), while the other caches are public (or shared). Caching proxies are usually located at the network edge close to the end-user. They reduce the outgoing Web traffic, resulting in a better response time for the user. Gateways are located at the network edge close to the origin server. They reduce the load on the server, particularly when dynamic Web contents are involved. Prefetching can equally be employed at any or a selection of these entities. Therefore, client-driven [5], server-driven [6], proxy-driven [7], collaborative [8], and multiple-entity [9] prefetching architectures have been proposed in the literature. In collaborative prefetching architectures, entities collaborate with each other in arriving at prefetching decisions, while in multiple-entity prefetching architectures, prefetching is carried out at multiple entities independently.

HTTP's caching infrastructure governs what is cacheable and what is not. The response to a GET request is cacheable by default, while the response to a POST request is not cacheable by default. Cache control directives exist to change the default behavior.

With cacheable entities, there are two primary indicators to support caching: an expiry time and a validator. The expiry time tells the caching entity how long the cached resource could be served out of the cache without revalidation from the origin server. Once the expiry time is reached, the resource should be revalidated by the origin server before use; if the cached copy is older than the server copy, then a fresh copy must be obtained. The validator helps with this. It is a unique hash value that represents the instance of the resource, and when the resource changes in any way, the hash would too. The cache keeps this server-supplied validator with the cached resource, and when the resource is deemed expired, the cache will conditionally request the resource by using the validator. If the cached validator is older than the server's, then the server would send a new copy of the resource; otherwise, the server would simply indicate that the resource has not changed and provide a new expiry time.

Prefetching must take into account the cacheability of a resource. A non-cacheable resource should not be prefetched as doing so will invalidate the intentions of the resource owner. Similarly, if a resource is marked for private caching, this resource can only be prefetched to a private (e.g., browser) cache and not to public caches (e.g., caching a proxy or gateway).

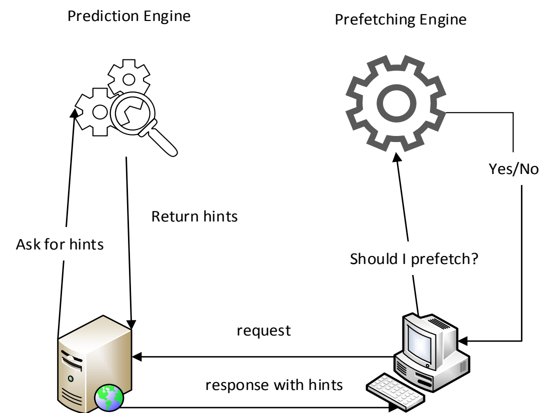


Fig. 1. A hint-based prefetching infrastructure.

GET https://www.google.co.nz/search?sclient=psy-ab&...
Accept: */*
Referer: https://www.google.co.nz/?gfe_rd=cr...
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/5.0...
Host: www.google.co.nz
DNT: 1
Connection: Keep-Alive
Link:</images/big.jpeg>;rel=prefetch

Fig. 2. Prefetch hints in an HTTP request.

B. Mozilla's Prefetching Infrastructure

Prefetching does two things: predict what is needed and prefetch to cache it. These two tasks are managed by a prediction engine and a prefetch engine, respectively. The prediction engine decides what to prefetch, typically by using intelligent prediction models. On the other hand, the prefetching engine decides when and whether to prefetch. It is usually integrated with a cache engine. The prefetch enginemonitors and uses hardware and user data (e.g., CPU utilization, bandwidth utilization, and user behavior) to make the prefetching decision. Fig. 1 illustrates a prefetching infrastructure based on a server-side prediction engine supplying prefetching hints.

Mozilla proposed a prefetching infrastructure utilizing such a prediction and prefetch engine architecture, as depicted in Fig. 1. The infrastructure is based on hints: the servers supply prefetching hints, and the clients use these hints to prefetch resources if they can (e.g., have idle network time for prefetching). The prediction engine should ideally be located on the server side; however, Mozilla does not allow such an implementation. The prefetch engine is integrated into Firefox, Mozilla's Internet browser. There are two methods to specify the prefetch hints. The first method adds a custom header called *Link* to the HTTP headers with the value of the suggested links. Fig. 2 shows the use of *Link* in an HTTP request.

```
<html>
  <head>
    <link rel="prefetch" href="/images/big.jpeg">
  </head>
  <body>
    Web content
  </body>
</html>
```

Fig. 3. Prefetch hints in an HTML document.

The second method is to include the suggested links in the HTML body by using the HTML <*link*> tags (see Fig. 3). In the absence of a server-side prediction engine, these hints can be manually specified for each resource. Firefox reads the hints from the HTTP headers or the HTML payload to decide whether and when to prefetch the links provided by the hints. It has a built-in prefetch engine that can detect whether the network is idle, and if so, it decides when to prefetch the resource. However, the idle time detection is based on Firefox's own usage of the network. In other words, it does not know whether there is another program running on the same machine using the network. Furthermore, Firefox does not know whether there are other users in the same LAN using the shared network.

Firefox also adds a custom HTTP request header (X-moz: *prefetch*) to tell a server that the incoming request is a prefetch request so that the server can distinguish prefetch requests from standard requests.

The well-known search engine website Google uses a hint-based prefetching-enabled HTTP server. When Firefox is used to access Google search, prefetching happens in the background during the browser-perceived idle time.

IV. CLASSIFICATIONS AND REVIEWS

A large portion of the prefetching literature is based on building prediction models for prefetching. There are two main aspects used for building prediction models: Web content and Web history.

Therefore, prefetching schemes can be broadly classified into content-based schemes and history-based schemes. Content-based schemes extract the top-ranked links on the basis of the content of the current page and the previously viewed pages. On the other hand, history-based schemes analyze the users' previous actions, such as the sequence of the requested Web resources to determine the next likeliest resources to be requested.

Content-based prefetching can be as simple as ranking keywords associated with links and prefetching the top-ranked links. Advanced content-based prefetching may use a user's interests learnt from his/her activities to aid the prefetching decisions. Chan [10] outlines a non-invasive learning approach to estimate user interest in a page and build user profiles using factors such as the frequency of visitation, bookmarks, page reading time, and the percentage of child links that have been visited.

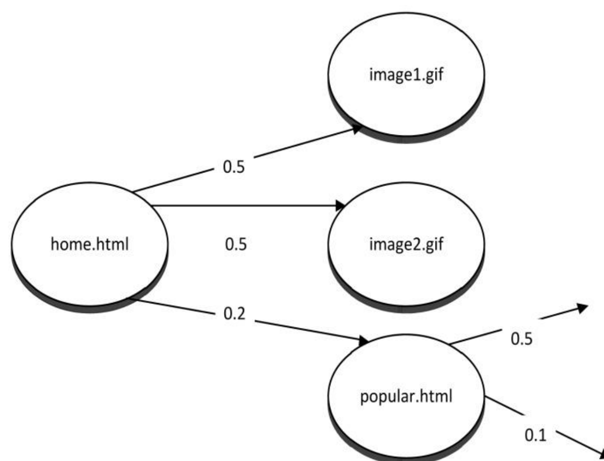


Fig. 4. Dependency graph model.

If a hint-based infrastructure such as that of Mozilla is used, a server-side content-based prediction engine can auto-tag content with the appropriate link tags. Similarly, a client-side content-based prediction engine can simply mark the selected links as prefetchable, which then the prefetch engine can use when appropriate.

History-based prefetching, on the other hand, constructs a prediction model using historical data. The model can be constructed using different characteristics of the historical data. The four most popular methods to construct the model are as follows: top ranked, dependency graph, Markov model, and data mining.

Top ranked is a very simple history-based model. It sorts the resources of a website by a selected resource property and fetches the top *N* resources; here, *N* denotes a predefined threshold number, such as top 10. It does not take the current page into consideration. No matter which page/resource in the site was accessed first, the top *N* resources of the site will be prefetched. For instance, prefetch by popularity is based on the visit count of the Web links.

Dependency graph prefetching is one of the first prefetching techniques. A dependency graph is a directed graph that represents the dependency relationships between several objects. It can be used for deriving an evaluation order. Usually, historical data are used for constructing a dependency graph model for prediction. However, such a model can also be constructed by analyzing Web content. This method can only look one step further, which means that it will only predict the next page rather than the next *P* pages on the basis of the current page. The main problem with this method is that its prediction accuracy is low.

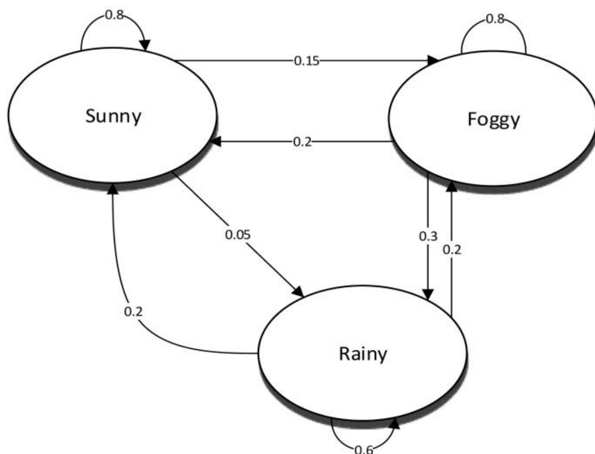


Fig. 5. A Markov model.

Fig. 4 shows an example of a history-based dependency graph. In this example, image1, image2, and popular.html depend on home.html. Fifty percent of the users will access image1 after accessing home.html. Fifty percent of the users will access image2 after accessing home.html. Twenty percent of the users will access popular.html after accessing home.html. If we use 50% as the threshold, image1 and image2 will be prefetched. Note that the sum of the percentages need not necessarily be 100%. The percentage for each page is calculated independently. A user who accesses home.html may access both image1 and image2.

Markov model is a stochastic model based on Markov property. It represents the distribution of the future states based on the current states. A Markov model is constructed from historical data and can be used for prediction. In a Markov model, the order has a large impact on the predictive capability. Increasing the order increases the model complexity. Therefore, higher-order Markov models leave many states uncovered and become unmanageable. On the other hand, a low-order Markov model has a low accuracy. The third- and the fourth-order Markov models have the best balance between accuracy and complexity according to previous research [11].

Fig. 5 shows an example of the first-order Markov model. This example uses the current weather information to predict the next day's weather. If the current day is sunny, there is an 80% probability that the next day will be sunny, 15% probability that the next day will be foggy, and 5% probability that it will be rainy. The sum of the probabilities should always be equal to 1.

Prediction by partial matching (PPM) is the common method used to build a Markov model. It is an algorithm widely used in data compression. PPM can also be used to cluster data into predicted groupings in cluster-based prefetching. For a large website containing millions of Web pages, a Markov model built with standard PPM will require large storage. Some proposals attempt to tune the PPM algorithm to avoid this overhead by restricting the servers to only collect access information for the most recent resources. However, this has the side-effect of a reduced hit rate due to the collection of a relatively less amount of historical information. Another variation of PPM, called longest repeating subsequences (LRS) PPM, only stores long branches with frequently accessed URL predictors. This consumes less storage and provides a relatively high prediction accuracy as compared to a standard PPM approach. However, the overall hit rate is still low because it ignores prefetching less-frequently accessed URLs. Chen and Zhang introduced a popularity-based PPM model using only the most popular URLs as root nodes; this model is shown to have good hit rates and requires less space [12].

Another approach to build the model is to use data mining techniques. Data mining is a computational process of discovering patterns in a large dataset. The overall purpose of data mining is to extract information from a dataset and transform it into an understandable structure. It involves six tasks: anomaly detection, association rule learning, clustering, classification, regression, and summarization. Association rule learning and clustering are the aspects that can be used for constructing a prediction model. Web logs are the main data from where useful knowledge can be extracted for Web prefetching.

Association rule-based prefetching uses historical data to find relationships between resources. It does not consider the order of resources either within a transaction or across transactions. A typical example of this relationship is "A user who accesses the home page is 80% likely to also access the contacts page." The model constructed using this relationship is very similar to a dependency graph. Further, other relationships can be found and used for constructing the model. For example, Yang et al. [13] used frequent access path patterns to extract association rules that improved both the hit rate and the byte hit rate dramatically. They first extracted the users' request sequences from the Web logs and then, applied an algorithm to formulate the association rules.

On the other hand, clustering-based prefetching clusters similar (according to some given criteria) resources in groups. Cluster analysis itself is not a specific algorithm but is a general task to be solved. Various algorithms can be applied to achieve this task. It depends on how the cluster is constituted and how efficient it is to find them. For example, users can be clustered into groups on the basis of their IP addresses. Association rule mining techniques can be used to cluster objects that may lead to having little separation between association rule-based prefetching and cluster-based prefetching. The common clustering algorithms are connectivity-based, centroid-based, distribution-based, and density-based.

Most clustering methods target intra-site Web pages. Therefore, they do not perform well for grouping inter-site Web pages that belong to different sites. Pallis et al. [14] introduced a clustering algorithm called *clustWeb*, which is designed for inter-site Web pages. They also developed a clustering scheme called *clustPref*, which can be easily adapted to Web prefetching.

V. SUMMARY AND CONCLUSION

This paper introduces Web cache prefetching and discusses the motivation and challenges of prefetching. It goes on to discuss the metrics that measure the effectiveness of prefetching, and reviews some state-of-the-art techniques that deal with prefetching. It classifies prefetching techniques into content-based and history-based schemes. It further classifies history-based schemes into four main categories: top ranked, dependency graph, Markov model, and data mining.

Top-ranked prefetching sorts the resources in a particular way and prefetches the top *N* resources. It may work well on some simple websites but is not very useful for most of the websites. Dependency graph is easy to implement but has a low accuracy. Markov models of the order of 3 or 4 have a better balance between accuracy and complexity than those of other orders. Data mining uses association rules and clustering to construct the models. It always ends with too many rules that are not useful.

REFERENCES

[1] H. Liu and M. Chen, "Evaluation of Web caching consistency," in *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE)*, Chengdu, China, pp. 130-132, 2010.

[2] D. Wessels, *Web Caching*. Sebastopol, CA: O'Reilly & Associates, 2001.

[3] M. Rabinovich and O. Spatscheck, *Web Caching and Replication*. Boston, MA: Addison-Wesley, 2002.

[4] G. Cho, "Using predictive prefetching to improve location awareness of mobile information service," in *Computational Science, Lecture Notes in Computer Science*, vol. 2331, pp. 1128-1136, 2002.

[5] A. Balamash, M. Krunz, and P. Nain, "Performance analysis of a client-side caching/prefetching system for Web traffic," *Computer Networks*, vol. 51, no. 13, pp. 3673-3692, 2007.

[6] J. Domenech, J. A. Gil, J. Sahuquillo, and A. Pont, "DDG: an efficient prefetching algorithm for current web generation," in *Proceedings of the 1st IEEE Workshop on Hot Topics in Web Systems and Technologies (HOTWEB)*, Boston, MA, pp. 1-12, 2006.

[7] L. Fan, P. Cao, W. Lin, and Q. Jacobson, "Web prefetching between low-bandwidth clients and proxies: Potential and performance," in *Proceedings of the ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, Atlanta, GA, pp. 178-187, 1999.

[8] X. Chen and X. Zhang, "Coordinated data prefetching by utilizing reference information at both proxy and web servers," *ACM SIGMETRICS Performance Evaluation Review*, vol. 29, no. 2, pp. 32-38, 2001.

[9] E. P. Markatos and C. E. Chronaki, "A top-10 approach to prefetching on the web," in *Proceedings of INET*, pp. 276-290, 1998.

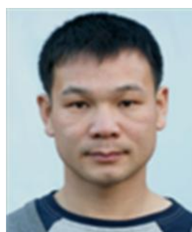
[10] P. K. Chan, A non-invasive learning approach to building web user profiles [Internet], http://citeseerx.ist.psu.edu/ viewdoc/summary?doi=10.1.1.35.2866.

[11] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz, and K. J. Worrell, "A hierarchical internet object cache," Department of Computer Science, Colorado University at Boulder, *Report no. CU-CS-766-95*, 1995.

[12] X. Chen and X. Zhang, "A popularity-based prediction model for web prefetching," *Computer*, vol. 36, no. 3, pp. 63-70, 2003.

[13] Q. Yang, H. H. Zhang, and T. Li, "Mining web logs for prediction models in WWW caching and prefetching," in *Proceedings of the 7th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, San Francisco, CA, pp. 473-478, 2001.

[14] G. Pallis, A. Vakali, and J. Pokorny, "A clustering-based prefetching scheme on a Web cache environment," *Computers & Electrical Engineering*, vol. 34, no. 4, pp. 309-323, 2008.



YuFeng Deng
is a Ph.D. candidate at the Department of Computer Science, University of Auckland. He holds a Bachelor of Technology (Hons) in Information Technology from the same University.

Sathiamoorthy Manoharan
is a senior lecturer at the Department of Computer Science, University of Auckland. He holds a Bachelor of Technology (Hons) in Electronics and Electrical Communication Engineering from the Indian Institute of Technology, Kharagpur, India. He has a Ph.D. in Computer Science from the University of Edinburgh, Scotland.