Reducing Web Latency with Hierarchical Cache-based Prefetching

Dan Foygel and Dennis Strelow
Computer Science Department, Carnegie Mellon University
{dfoygel, dstrelow}@cs.cmu.edu

Abstract

Proxy caches have become a central mechanism for reducing the latency of web document retrieval. While caching alone reduces latency for previously requested documents, web document prefetching *could mask latency for previously unseen, but correctly predicted requests. We describe a prefetching algorithm suitable for use in a network of hierarchical web caches; this algorithm observes requests to a cache and its ancestors, and initiates prefetching for predicted future requests if prefetching is likely to reduce the overall latency seen by the cache's clients. We introduce a novel cost-benefit model that allows us to judge the value of any cached or prefetched document, which we use to state a formal prefetching policy. Extensive simulations were run to judge the improvements offered by prefetching, and our approach is quantitatively compared to the method currently in use.*

1. Introduction

Proxy caches have become a central mechanism for reducing the latency of web document retrieval. Like processor caches, a web proxy cache satisfies new requests for previously retrieved data using a local copy. Satisfying a document request from the cache can significantly decrease latency if the proxy is closer to the client than the document's origin server (either physically or in terms of network topology), or if the origin server is heavily loaded.

While caching alone reduces latency for previously requested documents, web document *prefetching* could mask latency for previously unseen, but correctly predicted requests. Prefetching has been used to great advantage in file systems[5], and researchers have already performed some initial investigations of server-based web prefetching[4][6][2].

We have designed and prototyped a prefetching algorithm suitable for use in a network of hierarchical web caches such as the National Laboratory for Applied Network Research's Global Caching Hierarchy of Squid caches. This algorithm observes requests to a cache and its ancestors, and initiates prefetching for predicted future requests if prefetching is likely to reduce the overall latency seen by the cache's clients.

2. Related Work

As mentioned in the introduction, proxy caches can reduce the latency seen by clients that request a document recently retrieved for another of the cache's clients. Commercial web caches include NetAppliance's NetCache[3], which descends from the Harvest cache project. NetCache's public domain sibling Squid[7], under development by the National Laboratory for Applied Network Research (NLANR), is also derived from the Harvest project and forms the testbed for the prefetching method described here. Squid caches can be deployed in a hierarchical manner, and NLANR coordinates a nationwide hierarchy of caches with eight top-level, regional caches physically located at the original NSF supercomputing centers and other locations.

Some attempts have been made to decrease web document latency using prefetching, and our approach is closely related to these. Padmanabhan and Mogul[4] describe a method in which servers observe client access patterns and send prefetch suggestions to their clients, who decide whether to perform prefetching. In the "Top-10" approach advocated by Markatos and Chronaki[2], servers cooperate with clients and hierarchical caches to deliver their most popular documents before they are actually requested. The servers described by Schecter, et al.[6] attempt to predict which *dynamically generated* documents will be requested by clients, and reduce latency by generating documents before they are requested.

The server-based prediction algorithm used by Padmanabhan and Mogul is based on a prediction algorithm for file system prefetching developed by Griffioen and Appleton[1]; similarly, our cache-based algorithm is inspired by the file system prefetching method described by Patterson, et al.[5]. This method uses programmer-supplied prefetching hints, rather than attempting to predict accesses, but employs a cost-benefit model to intelligently divide the limited number of available buffers between the LRU cache and prefetching.

3. Predicting Future Requests

We now describe the design of a prefetcher that is intended for inclusion in the caches of a hierarchical network, such as NLANR's nationwide Squid hierarchy. This design has two major components. The first, described in this section, is a distributed algorithm that observes HTTP requests to each participating cache c and, based on these observations, attempts to predict upcoming requests to c. The second major component, described in section 4, is an arbiter that decides whether or not prefetching a document suggested by the predictor will actually decrease the overall latency seen by clients.

The distributed prediction algorithm estimates, for every pair of documents a and b, the probability that b will be requested soon, given that a is the most recently requested document. A document b is then identified as a good prefetch candidate if a is requested, and b has a high probability of being requested given that a has been requested.

The algorithm has three components. The first, described below in Section 3.1, is a local prediction module that executes separately within each cache c and estimates, for every pair of documents a and b, whether b is likely to be requested soon given that a is the most recently requested document, *using only the requests to cache c.* The second component is a protocol for communicating predictions between caches when a child cache does not have enough access information to make informed predictions, and is described in Section 3.2. The third, not described in detail here, is a straightforward method for combining the statistics from a cache and its ancestors to produce final predictions. This combination is designed so that suggestions from the parent dominate if the local cache has not accumulated enough history to generate meaningful prefetch suggestions.

3.1. Local Prediction

The local prediction method is similar in spirit to the dependency graph technique of Padmanabhan and Mogul[4] and the point profile method of Schecter, et al.[6]. If a document b is requested at time t, then:

- The most recent request time t_b for b is set to t.
- The count of requests c_b for b is incremented.

In addition, for each other recently requested URL a (where “recently” is determined using some time window w):

- The count $c_{a,b}$ of co-occurring requests between a and b is incremented.
- The delay between the most recent requests for a and b, $d_{a,b} = t_b$ - t_a, is added to a sum of delays $s_{a,b}$.
- A time-weighted mean delay $\bar{d}_{a,b}$ between requests for a and b is updated according to:

$$\bar{d}_{a,b} := \alpha d_{a,b} + (1-\alpha)\bar{d}_{a,b} \quad (1)$$

for some $\alpha, 0 < \alpha \leq 1$. The probability $p_{a,b}$ that document b will be requested within w seconds after a is requested can then be estimated simply as $p_{a,b} = c_{a,b}/c_a$. The other statisics, such as the time-weighted mean delay, are maintained for use by the arbiter.

3.2. Communicating Probabilities

If the number of requests for a document a to a cache c is not large, then the probability of an upcoming request for document b given a may be estimated poorly by c using the local prediction method described in the previous section. We therefore require a protocol for propagating the probabilities from c's parent p to c in the cases where c does not have enough historical data to compute reliable probabilities.

Our method is as follows. When a request for a misses in cache c, the request is forwarded to c's parent p. When p replies to c with the document, it includes an additional HTTP `Pragma` response header for each document likely to be requested after a. Our pragma header format is guided by the format given for extension pragmas in the proposed HTTP 1.1 standard, and is:

`Pragma: Prefetch = "`$c_a \;\; z_b \;\; s_{a,b} \;\; c_{a,b} \;\; \bar{d}_{a,b} \;\; u_b$`"`

where c_a, $s_{a,b}$, $c_{a,b}$, and $\bar{d}_{a,b}$ are the number of requests for a, the sum of delays, number of co-occurrences, and time-weighted mean delay described in the previous section, and z_b and u_b are the size and URL, respectively, of document b. The size z_b of document b is included for use by the arbiter.

Probabilities sent by the parent p are recorded in c, so that combined prefetch suggestions on subsequent requests for a that hit in cache c can be generated without contacting p.

4. Cost-benefit Model

The preceding section describes an effective methodology for generating prefetch suggestions for any document we have previously seen a sufficient number of times. However, not all prefetch suggestions should be used, and we need a principled method for finding the ones that are sufficiently accurate. In addition, prefetch suggestions compete for space with the cache of previously seen documents, and we would like to be able to arbitrate between these two space utilization strategies. To make our comparison fair, we assume that a constant amount of space is available for local document storage, and any space that is allocated to the prefetch buffer must therefore be taken away from the actual cache. With these goals in mind, we introduce a formal cost-benefit model that will allow us to judge the value of any cached or prefetched document. Comparing the values of currently cached documents with those of potential prefetch targets will allow us to formulate a policy for maximizing the benefits of prefetching.

4.1. Network Model

In order to calculate the benefits of prefetching, we must select the *common currency*[5] that will be used to measure the benefit. Since our ultimate goal is to mask document latency, we will select latency as our currency and attempt to minimize it. To simplify our analysis, we assume that the only latency associated with a document request is the network delay. In particular, we choose a linear network model, where the latency for a document of size S is

$$L(S) = N_C + N_B * S \tag{2}$$

(N_C is the constant network overhead and N_B is the constant describing the effective bandwidth of the network). Furthermore, we assume that there is no latency associated with locally stored documents, so all of the latency is masked if the document can be found in the cache (or prefetch buffer).

4.2. Predicting Future Requests

Our ability to compute the benefit of keeping a document in the local cache depends on our estimate of the likelihood that this document will soon be requested again. Similarly, in order to compute the benefit of doing a prefetch, we need to estimate the likelihood that this document will be requested soon after the current document. In either case, we are interested in the probability that one event is followed by another one, and we would like to incorporate our intuition that this probability decays with time.

To that end, we model the distribution of delays between the two events as a Gamma random variable. Specifically, the probability of delay x between the two requests is represented as

$$P(x) = \alpha\lambda^2 x e^{-\lambda x} \tag{3}$$

where α is the total probability that the second document will be requested, while λ is the most likely value for the delay. This particular probability function allows us to capture the intuitive idea that the delays have a heavy-tail distribution with an arbitrary mean, controlled by λ.

However, for our particular application, we are really interested in the probability that a particular request will happen at some point in the future. In other words, we need to know the integral of our probability function from some time t until ∞, which is equal to

$$\int_{x=t}^{\infty} P(x) = \int_{x=t}^{\infty} \alpha\lambda^2 x e^{-\lambda x} = \alpha(\lambda t + 1)e^{-\lambda t} \tag{4}$$

Note that the integral is always equal to α when $t = 0$ but falls off at different rates, depending on the value of λ.

Thus, in order to compute the probability that a particular event will happen in the future, we need to know the values of α and λ associated with that event. Note that α is simply the total probability that a particular event will occur, and as such is easily calculated. However, the optimal λ cannot be calculated without access to all the previous data. In an effort to conserve space, we will simply store the mean delay for the distribution and then use a value of λ that will produce the same mean delay. The relationship between λ and the mean delay $\overline{x}$ is computed by noting that the integral in equation 4 will be equal to $\frac{1}{2}\alpha$ at the mean, yielding $\lambda\overline{x} \approx 1.67834699$.

4.3. Value of a Prefetched Document

At this point, we are ready to introduce our formal definition of the value of prefetched documents.

> *The expected benefit of prefetching document A is the probability that it will be requested multiplied by the latency that would be masked if it is.*

To calculate the probability, recall that the historical analysis (section 3) parametrized the distribution of the delay values between the request for the two documents, producing estimates for α and $\overline{x}$. Combining equations 2 and 4, we conclude the that expected benefit of prefetching is

$$E(V_A) = \mu * \alpha(\lambda t + 1)e^{-\lambda t}(N_C + N_B * S_A) \tag{5}$$

where μ is a constant discount factor (discussed below) and the rest of the terms are defined above. The expected cost of ejecting a prefetched document is actually computed by the same formula, since the value of the document is independent of whether or not the document has actually been prefetched. In fact, the only difference is how we define the age of the document, t. Let t_{REQ} be the time at which the document was requested and let t_{CURR} be the current time. Then, for documents that have already been prefetched, t is simply $t_{CURR} - t_{REQ}$, whereas if a document must actually be prefetched from a remote server, $t = t_{CURR} - t_{REQ} + (N_C + N_B * S_A)$ to account for the network latency.

4.4. Value of a Cached Document

To compute the value of a cached document, we use a very similar method to the one above.

The expected benefit of keeping document A in the cache is the probability that it will be requested again multiplied by the latency that would be masked if it is.

Our estimate of probability is similar in spirit to how we treat prefetch suggestions. We assume that the distribution of delays is a Gamma random variable, where in this case we're considering the delay between two requests for the same document. We calculate $\overline{\alpha}$ by computing what percentage of cached documents are requested again, and $\overline{\lambda}$ is computed based on the mean delay, as described previously. If the last request for the document came at time t_{LAST} and the current time is t_{CURR}, let's define the age of the cached document as $t = t_{CURR} - t_{LAST}$. Again combining equations 2 and 4, we conclude that the expected benefit of caching is

$$E(V_A) = \overline{\alpha}(\overline{\lambda}t + 1)e^{-\overline{\lambda}t}(N_C + N_B * S_A) \qquad (6)$$

The expected cost of ejecting a cached document is actually computed by the same formula, since the value of the document independent of whether or not the document has actually been cached. Note that in these cases, $\overline{\alpha}$ and $\overline{\lambda}$ are global cache parameters, while in the case of prefetching, each suggested target had its own values of α and λ.

4.5. Comparing Costs and Benefits

The above definitions of the value of cached and prefetched documents allows us to formulate the following cost/benefit policy:

At every time step, compute the value of every locally stored document and every potential prefetching or caching target. Of all the possible document sets that fit into the available storage, select the one with the highest total value as the new locally stored document set.

5. Implementation

We have implemented the distributed prediction algorithm and cost-benefit model described in sections 3 and 4. The software architecture of this implementation closely follows the organization presented there, and the modules can be linked to Squid (section 5.1) or to a simulator designed to test the cost-benefit model (section 5.2).

5.1. Integration with Squid

As a proof of concept, we have integrated the distributed prediction (local, parent, and combined estimation) modules with Squid, using a dummy cache arbiter that approves all suggested prefetches.

When a client request is received from a child cache or a user's browser, the URL is forwarded by Squid to the local estimation module, to the cache arbiter, and to the parent cache. The local estimation module updates its co-occurrence probabilities based on the URL and current time. The cache arbiter forwards the URL to the combined estimation module, which gets prefetch suggestions from the local estimation module, and retrieves the stored parent-suggested probabilities from the ancestor estimates module. The combined prefetch suggestions are then returned to the arbiter, which performs the cost-benefit analysis and returns the actual prefetch suggestions to Squid, which sends the corresponding HTTP requests to the parent cache.

When an HTTP *reply* is received by c, any co-occurrence probabilities supplied by the parent in `Pragma: Prefetch` headers are extracted and supplied to the ancestor estimation module. The URL is also supplied to the combined estimation unit, which returns probabilities for the URL to be included in the reply to the child cache. Squid includes these `Pragma: Prefetch` headers in an augmented reply to the original request. If the probabilities returned from the parent are significantly different than the probabilities originally used to initiate prefetches, additional prefetches may be performed.

5.2. Simulator Implementation

While the Squid implementation allowed us to test the realizability of our design, we needed a far more controlled setting to actually test its performance. To that end, we developed a simulator that recreates the behavior of Squid by processing its access logs.

The squid simulator maintains a list of documents stored locally and a list of potential prefetching or caching targets. For every request in the access log, the simulator adds the requested document as a caching suggestion. It then uses the predictive prefetching module to generate a list of suggestions for this document and adds those to the target list as well. At this point, it updates the values of all the documents in both sets, and then attempts to reallocate space to maximize the total value.

While the policy described in section 4.5 is guaranteed to find the optimal document set, an algorithm that would follow that policy is infeasible to implement due to high running costs. We could approximate this policy by simply selecting the documents with the highest expected value, but that would ignore the fact that some documents are much larger than others. Intuitively, we would like our selection algorithm to maximize value while at the same time minimizing size, since we are limited in the total amount of available space. We approximate this strategy in the simplest possible way by selecting the documents with the highest value-to-size ratio. In particular, if the best document in the target set has a higher value-to-size ratio than the worst document in the local set, we will eject documents from the local set until there's enough room to accommodate the target document.

Finally, we address the issue of parameter estimation in the simulator. The network constants $N_C = .571$ seconds and $N_B = .0257$ seconds/KB were estimated a priori by examining a Squid access log and finding the best-fit line to the download time vs. document size distribution. The global cache constants $\overline{\alpha}$ and $\overline{\lambda}$ were computed dynamically based on the algorithm outlined above. The individual prefetch suggestion constants α and λ were also estimated dynamically using the same method. The remaining parameter, μ, was left as a free variable to allow us to experiment with various linear combinations of prefetching and caching.

6. Results and Discussion

The simulator described above allowed us to test our prefetching algorithm in a highly controlled setting. However, in order to generate reproducible results, we needed to standardize our input to the simulator - the access log. To that end, we developed a quasi-synthetic log by recording the requests to the CNN web site from 50 short browsing sessions, which generated 3139 requests (255 unique) for a total of approximately 17MB (2.5MB unique). We ran our simulator for a variety of possible cache sizes and values of μ, the constant from equation 5 that can be thought of as the relative weighting of prefetched vs. cached documents.

Figure 1 shows the results of running the simulator for cache sizes between 1 and 4096 KB, varied by powers of 2. The y-axis shows the Cache Hit Rate - the percentage of requests which were satisfied by the locally stored document set. The line labeled “LRU” shows the performance of the strategy adopted by the standard Squid implementation (as well as most other caching applications). In this case, the freshest document is always added to the cache, and older documents are ejected to make room for it (starting with the one least recently requested).

The next line, labeled “No Prefetch”, shows the behavior of the system with the cost-benefit model in place but without prefetching (i.e. $\mu = 0$). To understand how this different from the LRU case, recall that we are now ranking cached documents by the value-to-size ratio, meaning the strategy may prefer a smaller older document to a larger newer one, whereas the LRU approach will always pick a newer document over an older one.

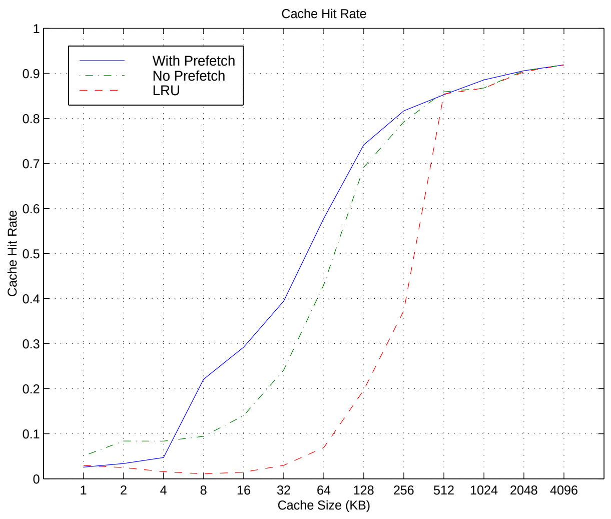

Figure 1. Cache hit rate for various strategies.

The “No Prefetch” strategy performs significantly better than LRU for a wide range of cache sizes. Because it favors smaller documents, it's actually able to fit more documents into the same amount of space, and therefore the cache hit rate is higher. While this may seem like “cheating” in some sense, insofar as we want to maximize the cache hit rate, we have simply shown that LRU is a rather inefficient strategy. However, we might instead be interested in minimizing the average latency for a document, and we discuss the results of that measurement below.

The next line in figure 1, labeled “With Prefetch”, shows the results of adding prefetching to the “No Prefetch” strategy (i.e. $\mu = 1$). Note that while the improvements aren't striking, prefetching does help over a wide range of cache sizes. In particular, the cache hit rate is more than doubled for some very small cache sizes (8KB and 16KB), and this improvement gets progressively smaller as the cache size increases. The actual size of the cache only makes sense in relation to the total amount of data that's being cached, and both are extremely small in this simulation.

Note that all three lines in figure 1 plateau at slightly above 90%, even when the cache size is higher than the total size of the requested data. This is attributable to the “compulsory miss” - the cost associated with seeing a particular document for the first time and not knowing anything about it. Prefetching is unable to help with the compulsory miss unless parent-provided prefetch suggestions are utilized, which wasn't done in this implementation of the simulator.

Recall that the original goal of the cost-benefit model was to maximally mask latency. Figure 2 presents the same three strategies as shown in figure 1, except in this case the y-axis shows the average latency for a document (calculated by adding the latency (equation 2) for all cache misses and dividing by the total number of requests). Note that Figure 2 essentially paints the same picture as discussed above - “No Prefetch” is considerably better than LRU, and prefetching provides an additional, smaller improvement (especially for smaller cache sizes).

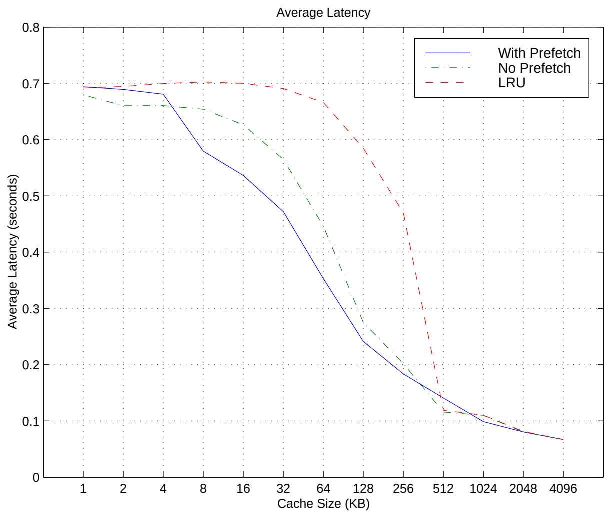

Figure 2. Average latency for various strategies.

7. Conclusions

We have argued that web prefetching has the potential to decrease web request latency significantly, and that a hierarchical cache network is the ideal location to perform prefetching. We have implemented a prototype that employs predictive prefetching and communication between caches, and performed some initial evaluation. Extensive simulations have shown that the cost-benefit model provides for significant improvement over LRU caching, and prefetching provides an additional boost in performance. It should be noted that the increased performance due to prefetching comes at the cost of higher network utilization, so there is clearly an important trade-off involved. However, it is often the case that adding more traffic to an underutilized network link incurs no additional cost (for example, modems used at home), in which case it is clearly beneficial to use this excess bandwidth to reduce perceived latency. In light of this potential application, we believe that prefetching at the cache level is a potentially valuable technique that merits further investigation.

8. Acknowledgements

Special thanks to Garth Gibson, for his careful reading and helpful suggestions during the course of this work.

References

[1] J. Griffioen and R. Appleton. Reducing file system latency using a predictive approach. In *Proceedings of the 1994 Summer USENIX Technical Conference*, Cambidge, MA, June 1999.

[2] E. Markatos and C. E. Chronaki. A top-10 approach to prefetching the web. In *Proceedings of INET '98*, Geneva, Switzerland, July 1999.

[3] Network Appliance. Netcache appliances. `http://www.netapp.com/products/netcache`.

[4] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve world wide web latency. *ACM SIGCOMM Computer Communication Review*, July 1996.

[5] R. H. Patterson, G. A. Gibson, E. Kinting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, pages 79–95, December 1995.

[6] S. Schechter, M. Krishnan, and M. D. Smith. Using path profiles to predict HTTP requests. In *Seventh International World Wide Web Conference*, Brisbane, Australia, April 1998.

[7] D. Wessels. Squid web proxy cache. `http://www.squid-cache.org/`.