Hinted caching in the Web

Jeffrey C. Mogul
mogul@wrl.dec.com
Digital Equipment Corp. Western Research Lab.
250 University Ave., Palo Alto, CA 94301

Abstract

The World Wide Web, like any practical distributed system, benefits greatly from caching. Many studies have shown relatively poor cache performance, because existing caches depend on temporal locality, and reference patterns in the Web do not have particularly high temporal locality. We can improve Web caching by exploiting spatial locality. This requires protocol changes to allow servers to provide hints to caches, both to support prefetching and to improve allocation and replacement policies.

1. Introduction

The World Wide Web (WWW) depends in many cases on the use of caches to both hide latency and conserve bandwidth. Caching in the Web is evolving from a naive model to one that considers the semantics of various kinds of operations and data values. The Hypertext Transfer Protocol (HTTP), especially in its most recent revision [3], includes features that support the efficient validation of cached values, as well as explicit server and client control over what values may be cached, and for how long.

HTTP could provide more support for improving the performance of caches. The protocol transmits some information useful for making replacement decisions, such as the most recent modification date, but caches must use heuristics to decide what values to replace (and what values to store in the first place). Least-recently-used and other cache algorithms that depend on observing the reference stream often do not accurately predict future references, because there is relatively little temporal locality of reference in the Web (compared to, say, a distributed file system [2]).

In this paper, I discuss several techniques that have been proposed by myself and by others for improving caching in the Web by adding features to HTTP. These techniques allow Web caches to exploit the relatively low temporal locality characteristic of Web retrievals, and also to exploit the available spatial locality.

2. Think globally, act locally

Because the Web is truly world-wide, it suffers more from high latencies and low bandwidths than perhaps any other distributed system. Also, it is composed of a huge number of autonomous and administratively implementations. These two aspects of the Web conspire to strongly separate knowledge about the semantics of resources, and the patterns of references to them, from the systems that must act on this knowledge in order to make caching work well. An HTTP server could know or learn a lot about reference interarrival times, modification intervals, correlations between references, etc.; this information cannot easily be inferred by a cache. Yet the caches must have this kind of information if they are to make the best use of the network.

If the servers transmitted such knowledge to the caches, the caches could use these hints to optimize their behavior. The current HTTP protocol provides no mechanism to do this, but it could.

3. Allocation and replacement

The most basic policy decision that a cache must make is how to allocate its (finite) storage space. The cache must decide whether to store a resource in the first place, and it must decide when to remove a stored resource to make way for something more valuable.

The problem for Web caching is compounded by the highly variable size of the resources transmitted (ranging from tens of bytes to tens of megabytes). This makes a simple LRU scheme untenable, because a single large entry can wipe out thousands of smaller entries that are nearly as recently-used, and collectively are more likely to satisfy a future cache hit. Williams et al. have shown that a cache removal policy based on object size provides a higher hit rate than an LRU policy, on a variety of reference traces [13]. Their preferred policy simply removes the largest cache entry, when a new insertion requires a removal.

The decision also depends on the cost of servicing a cache miss, which varies dramatically both among different resources and also (because of network load variations) for retrievals of a single resource. This tends to favor caching of large resources or those that took a long time to load, other things being equal. (Williams et al. did not measure cache miss costs, but only studied hit rates.)

An HTTP cache might also consider the expiration date of a cache entry. HTTP servers may assign such dates to indicate that after a certain time, a cache must validate its cached copy with the ‘‘origin server’’ before using the copy. When the cache makes a validation request (e.g., a ‘‘conditional GET’’), the server may either return an updated copy of the resource, or it may indicate that the cached copy is still usable. The latter is often far cheaper than a complete reload, and so a cache may benefit from holding onto expired resources.

Thus, in order to make a sensible allocation or replacement decision, a Web cache must compute a cost estimation that includes the expected duration until the next reference to the resource, the expected cost of reloading or revalidating the resource, the expiration time of the resource, and the size of the resource to decide if it is worth caching this resource instead of some number of other resources. Of those factors, all but the first one are available locally at the cache.

But how does a cache know when the next reference will come? Simple LRU schemes attempt to predict which objects will be referenced soon, but not how soon. A cache could maintain a database of reference times for a large set of resources, but unless the cache has a lot of clients, it may not have seen a significant number of prior references for any given resource (and the universe of resources for which observations must be kept could be quite large).

Origin servers, on the other hand, can easily observe a significant fraction of references to the resources they own. If ‘‘hit-metering,’’ a technique by which a cache reports back to the origin server the number of references to a cached copy of a resource, becomes popular, then an origin server might know essentially the complete reference pattern. And an origin server needs only store a bounded number of observation summaries, since it only stores observations for its own resources.

So an origin server could make a reasonably informed prediction of how much time will pass until the next reference to a resource, and it could attach this prediction as a hint whenever it sends a response to a cache. A cache, however, may only see a small fraction of the references to a server, and so we need to provide some way for the cache to adjust the server’s prediction accordingly, such as normalizing the reference rate by the number of users.

We could suppose that the server’s observed reference rate for a resource is proportional to the number of users of that server. That is, each of the server’s users has an equal probability of referencing the resource. So if the server measures the size of its user base (perhaps by counting unique client addresses, although this is distorted by the existence of proxies), and includes this along with the reference rate hint, the cache could weight the hit rate accordingly. The cache might assume that all of its own users had an equal chance of referencing the resource, or it could keep track of how many of its users referenced each origin server, which allows a more realistic weighting.

A reference-rate hint attached to an HTTP response might have the form:

```
RefRate: mean_interarrival_time stddev #observations #users
```

and a specific example might be:

```
RefRate: 375 129 4738 623
```

It might be necessary for the server to age its observations so that old hits are not given too much weight. For example, a Web page for the Superbowl (the US National Football League championship) might see 1 million hits on a single day in January, but two weeks later might be of no interest to anyone. We would not want to cache that page in a mistaken belief that the mean interarrival time is always 80 msec.

Some resources may be subject to highly periodic references. For example, the Dilbert cartoon [1] changes daily, and most of its users will re-retrieve the page once a day. While a cache *might* infer this kind of reference pattern by simple observation, it would be far easier for the service author to explicitly declare the per-user interarrival time, based on the semantics of the server. So a periodic-reference rate hint might have the form:

```
RefPeriod: interarrival_time
```

and the value for a daily cartoon might be:

```
RefPeriod: 86400
```

Although caches typically use an expiration time to control the use of revalidation, they can also use it to control allocation or replacement. However, since the existing `Expires:` header does not distinguish between hard expiration (‘‘this resource will cease to be valid’’) and soft expiration (‘‘your copy should be revalidated’’), it does not help the cache decide between discarding the cached copy (because it is useless) or keeping it (because it is likely to be revalidated). The service author may well have specific semantic information to resolve this question, so the HTTP protocol ought to provide a way to transmit this kind of hint.

There may be no *a priori* way of knowing if such hints will actually improve cache behavior, although trace-based experiments could help. Since the benefit is unknown, if the HTTP protocol is revised to include hints then their format should be as compact as possible, while still providing as much information as can be easily collected.

4. Prefetching

Improved allocation and replacement policies can only help if cached values are re-referenced at least once. Numerous studies have suggested that even with an arbitrarily large Web cache, cache hit rates do not get much above 30%-50% [4, 10, 13]. Some Web pages, such as those returned in response to queries, are inherently uncachable, but many others are simply not worth reading twice (such as a daily cartoon). Caches that depend on temporal locality probably will never get cache hit rates above a modest value.

The situation is not hopeless. Reference streams have both temporal and spatial locality, and cache designers for CPUs and file systems have long known how to exploit spatial locality, using techniques such as large cache lines or prefetching.

One increases cache line size to benefit from the likelihood that two co-located data items will be referenced at about the same time. For the Web, Touch [12] proposes an analogous technique called ‘‘source anticipation.’’ In this technique, upon receiving a request for an object, the server not only transmits the requested object but also all ‘‘linked items,’’ anticipating the client’s future requests. Source anticipation may work in some cases, but seems inappropriate in general because there is no *a priori* information about which of a large set of linked items the client is likely to request.

Sequential prefetching in a CPU cache or file system also exploits co-location, anticipating that if one cache line is needed, the next one might well be needed soon. Some CPU designs are more sophisticated, allowing a program to issue a prefetch instruction (such as the Alpha’s FETCH [11]) to hint to the cache what words will be needed soon. Unlike large cache lines or sequential prefetching, a prefetch instruction captures spatial locality at a higher level of abstraction. Patterson et al. have suggested a similar mechanism for file systems [9].

To make prefetching work in the Web, we need an oracle that operates at the right level of abstraction to predict the sequence of future references. For any Web server with a moderately large and relatively homogeneous user base, it should be possible to predict the reference stream of one user by observing the behavior of previous users.

In this approach, the server builds up a prediction model based on past reference streams. Given a new reference, the model then predicts the next few references, and assigns them probabilities. If the probability of a given prediction is high enough then the client should prefetch the object. The latency of the prefetch can be partially hidden by the user’s ‘‘think time.’’

Several file system projects have employed this technique. Griffioen and Appleton [5] use the most recent reference to predict some number of future references. Kroeger and Long [6] use one or more recent references to predict the next reference. Both projects found improved cache hit ratios.

Can the same approach be applied to the Web? Working with the author, Padmanabhan [7, 8] implemented a prediction engine, applied it to traces from an actual Web server, and simulated mean latency improvements of up to about 60%. (The potential improvement may be higher, since this study assumed a relatively low network latency.)

While only the server has enough history to make useful predictions, only the client has enough knowledge to decide whether to act on them. For example, the client may already have the predicted objects in its cache, or the user may have already requested something unrelated. In our prefetching proposal, the server makes the predictions and transmits them to the client as part of the response to a request. The predictions include the expected probability of reference, the size of the object, and other useful information. The client software then uses this hint to decide what to prefetch. The client may set a minimum threshold for the expected probability, below which it will not act on the prediction.

Because predictive prefetching inevitably retrieves objects that will not be used, it increases the total bandwidth required for a given reference stream. Padmanabhan's study showed bandwidth requirements increasing roughly in proportion to the decrease in latency. For example, one can set the client's prefetching threshold to a level that results in a 40% reduction in mean latency, and as a result, the total number of bits transferred will increase by about 40%. Note that simply increasing the available bandwidth by 40%, without using prefetching, does *not* improve latency nearly as much as prefetching does.

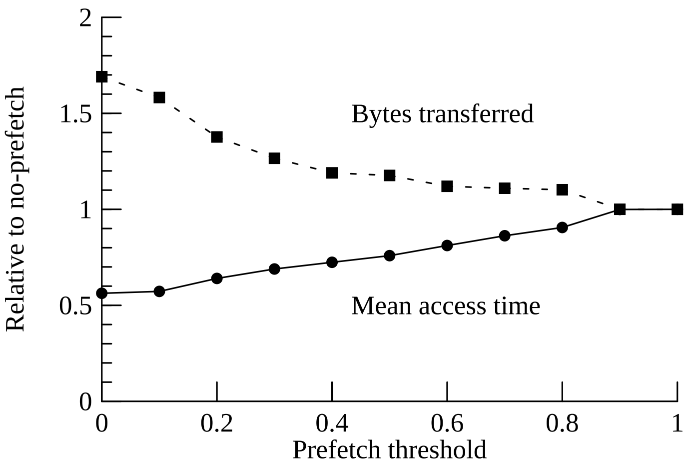

Figure 4-1: Effect of prefetch threshold on latency and utilization

Figure 4-1 shows how the simulated mean access time and mean network utilization vary with the prefetch threshold. A prefetch threshold of 1.0 corresponds to the non-prefetching case. (The data points are a reformatted subset of those plotted in figures 5 and 6 in [8].)

In some cases, this bandwidth increase makes prefetching inappropriate, but many network paths have excess capacity. Judicious use of this capacity for prefetching improves apparent latency without degrading service for anyone. It should be possible to inhibit prefetching over shared paths that are currently congested. Also, many users connect to the Internet via non-shared, low-bandwidth dialup links, and it is entirely appropriate for a client to use prefetching to fully utilize the bandwidth of such a link.

Note that although the previous work on predictive prefetching [5, 6, 7, 8] has assumed static values for the prefetch threshold, the client's threshold could vary dynamically, depending on the available bandwidth and the predicted retrieval latency.

While the initial results of this work are promising, many of the details are unresolved. For example, what is the best way to collect and represent past history? How should prefetch thresholds be set? What algorithms should clients use to decide what to prefetch?

5. Bandwidth and latency sensing

To decide whether to prefetch something, or to estimate the cost of reloading a cache entry that is eligible for replacement, a Web cache must estimate of the bandwidth and latency of the relevant network path. (We assume that it knows the size of the object, either from the prefetch hint or from the existing cache entry.)

Network paths often include links with highly disparate bandwidth and latency; one cannot infer from the full-path bandwidth alone what the individual link characteristics are. Web paths in particular often include backbone links with high bandwidth and speed-of-light delays, intermediate links with high queueing delays, and dialup ‘‘tail’’ links with low bandwidth and speed-of-light delays. One should prefetch over a low-bandwidth path if the bandwidth bottleneck is only in a single-user tail link, while the rest of the path has excess capacity.

Web caches are often placed between links of different performance, as well as at the ultimate client. Because such caches are active participants in the HTTP protocol, we could use their help to determine the relevant link characteristics.

If a client retrieves a 10KB document and this takes 10 seconds, it can infer that the bottleneck link has a bandwidth of 8 Kbit/s. Suppose that a proxy cache sits between a high-speed backbone link and a dialup link to that client. If the proxy has observed that it took just 80 msec to retrieve the document from the origin server, it can infer that its path to the server has at least 1 Mbit/s of available bandwidth. If it sends this observation to the client, piggybacked onto the response, then the client knows it is probably safe to prefetch other documents from this server. That is, while prefetching will tie up its own tail link, it will not significantly interfere with other users.

However, if the proxy observes that its path to the origin server is also very slow, then it can warn the client and thereby prevent inappropriate prefetching.

A more sophisticated approach might look for link congestion rather than link bandwidth, perhaps by observing the TCP congestion window and round-trip time variance. The caches can also monitor the round-trip time (either at the TCP level or at the HTTP level) to help predict the retrieval latency for small objects.

Even if the path between the proxy and origin server is congested, if the path between client and proxy is underutilized then it would be appropriate to prefetch any documents already cached at the proxy. The HTTP/1.1 proposed standard includes an ‘‘only-if-cached’’ directive that may support this technique [3].

6. Conclusions

Minor changes to the HTTP protocol, along with active participation by both origin servers and caches, should allow Web caches to exploit spatial locality of reference. This should greatly improve the effectiveness of Web caches.

References

[1] Scott Adams. The Dilbert Zone. http://www.unitedmedia.com/comics/dilbert/.

[2] Mary G. Baker, John H. Hartman, Michael D. Kupfer, Ken W. Shirriff, and John K. Ousterhout. Measurements of a Distributed File System. In *Proc. 13th Symposium on Operating Systems Principles*, pages 198-212. Pacific Grove, CA, October, 1991.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. *Hypertext Transfer Protocol -- HTTP/1.1*. Internet Draft draft-ietf-http-v11-spec-05, HTTP Working Group, June, 1996. This ‘‘work in progress’’ is a preliminary draft, and will be updated.

[4] Steven Glassman. A Caching Relay for the World Wide Web. In *Proc. of the First International World-Wide Web Conference*, pages 69-76. Geneva, May, 1994.

[5] James Griffioen and Randy Appleton. Reducing File System Latency using a Predictive Approach. In *Proc. 1994 Summer USENIX Conf.*, pages 197-207. Boston, MA, June, 1994.
[6] Thomas M. Kroeger and Darrell D. E. Long. Predicting File System Actions from Prior Events. In *Proc. USENIX 1996 Technical Conference*, pages 319-328. San Diego, CA, January, 1996.
[7] Venkata N. Padmanabhan. Improving World Wide Web Latency. Master's thesis, University of California at Berkeley, May, 1995.
[8] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *Computer Communication Review* 26(3):??-??, July, 1996.
[9] R. Hugo Patterson, Garth A. Gibson, Eka Ginting, Daniel Stodolsky, Jim Zelenka. Informed Prefetching and Caching. In *Proc. 15th Symposium on Operating Systems Principles*, pages 79-95. Copper Mountain, CO, December, 1995.
[10] James E. Pitkow and Margaret M. Recker. A Simple yet Robust Caching Algorithm Based on Dynamic Access Patterns. In *Proc. Second WWW Conference '94: Mosaic and the Web*, pages 1039-1046. Chicago, IL, Oct., 1994.
[11] Richard L. Sites and Richard T. Witek. *Alpha AXP Architecture Reference Manual, 2nd Ed.* Digital Press, Boston, MA, 1995.
[12] J. D. Touch and D. J. Farber. An Experiment in Latency Reduction. In *Proc. IEEE Infocom*, pages 175-181. Toronto, June, 1994.
[13] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal Policies in Network Caches for World-Wide Web Documents. In *Proc. SIGCOMM '96 Symposium on Communications Architectures and Protocols*. Stanford, CA, August, 1996.