Prefetching Hyperlinks

Dan Duchamp

AT&T Labs – Research

Abstract

This paper develops a new method for prefetching Web pages into the client cache. Clients send reference information to Web servers, which aggregate the reference information in near-real-time and then disperse the aggregated information to all clients, piggybacked on GET responses. The information indicates how often hyperlink URLs embedded in pages have been previously accessed relative to the embedding page. Based on knowledge about which hyperlinks are generally popular, clients initiate prefetching of the hyperlinks and their embedded images according to any algorithm they prefer. Both client and server may cap the prefetching mechanism's space overhead and waste of network resources due to speculation. The result of these differences is improved prefetching: lower client latency (by 52.3%) and less wasted network bandwidth (24.0%).

1 Introduction

The idea of prefetching Web pages has surely occurred to many people as they used their browsers. It often takes "too long" to load and display a requested page, and thereafter several seconds often elapse before the user's next request. It is natural to wonder if the substantial time between two consecutive requests could be used to anticipate and prefetch the second request.

The related work section of this paper cites 14 distinct prior studies of prefetching for the Web. In each of these studies, any *transparent* prefetching algorithm (meaning that the user is uninvolved) is also *speculative*. Speculative means that some system component makes a guess about a user's future page references based on some knowledge of past references, gathered from that user alone or from many users.

This paper examines a new method for prefetching Web pages into the client cache that is also transparent and speculative. While the basic approach to prefetching is the same in all studies, major differences among prefetching methods lie in the details. The major characteristics that distinguish this study from prior ones are:

1. We have implemented our ideas, twice in fact.
2. Clients send reference information to servers, which then disperse aggregated information to other clients in near-real-time. The reference information indicates how often hyperlink URLs embedded in pages have been previously accessed relative to the embedding page.
3. Servers aggregate the reference information in near-real-time rather than, say, overnight, allowing for prefetching decisions based on up-to-date usage patterns.
4. Clients initiate prefetching according to any algorithm they prefer; they also control how to age reference information.
5. Prefetching is not limited to URLs on the same server, or to URLs previously accessed by the same client.
6. Many un-cacheable pages may be prefetched, including pages generated dynamically, by query URLs, or those having cookies.
7. Both client and server can cap the prefetching mechanism's overhead and waste.
8. In one implementation, proxies are used to avoid changing either the browser or Web server.
9. HTTP is extended.
10. The prefetching algorithm continually measures bandwidth available to the client and limits prefetching requests to a fraction of the available bandwidth.

Most of these characteristics are not shared by most prior studies.

The result of these differences is improved prefetching: lower client latency (52.3% reduction) and much less waste (62.5% of prefetched pages are eventually used).

Section 1.1 gives a brief sketch of how prefetching works. Section 2 summarizes prior work in the area. Section 3 describes some conclusions, drawn from reference traces, that support certain design decisions, and Section 4 describes the design as well as two implementations. Section 5.1 analyzes the costs and benefits of this approach to prefetching.

1.1 Summary of Prefetching Method

Upon their first contact in "a while," a client and server negotiate the terms of prefetching: whether it will happen, and when and how much information they will exchange to support it.

Once terms have been negotiated, clients send *usage reports* to servers promptly but as part of the critical path of a GET request. Usage reports describe the fact that one or more URLs embedded within a page was recently referenced from that page. For example, if a client references page P and then references page Q based on an HREF embedded in P, then the usage report will indicate that P referenced Q; the usage report will also include other information useful for prefetching, such as the size of Q and all its embedded images, and how much time elapsed between the reference to P and the reference to Q.

The server makes a best effort to accumulate the information from all usage reports that pertain to the same page, P; the usage reports are kept ordered by time. Whenever the server delivers P to a prefetch-enabled client, it attaches a summary (called a *usage profile*) of the information that it has obtained from earlier usage reports for that page, from all clients. The summary, whose format was negotiated earlier, indicates how often HREFs embedded in page P have been referenced, relative to the number of references to P in the same time period(s). Time is measured by references; thus, for example, a client can negotiate to receive usage profiles that describe the references of embedded HREFs relative to the last 10, 25, and 50 references of the page P.

A client that receives a usage profile along with page P may choose whether or not to prefetch any HREFs embedded in P, according to any algorithm it prefers. The decisions whether and how to prefetch rest with the client because the client best knows its own usage patterns, the state of its cache, and the effective bandwidth of its link to the Internet.

2 Related Work

Here we survey 14 prior separate efforts relevant to prefetching in the Web, divided into three categories: software systems; papers describing algorithms, simulations, and/or prototypes; and papers that establish bounds.

2.1 Software Systems

Smiley is similar to our work in that prefetching is decided by the client based on usage statistics about embedded HREFs. The client also continually monitors its available bandwidth. One major difference is that images embedded within prefetched pages are not themselves prefetched. The Smiley implementation is only a demonstration, and results [16, 17] are obtained by a simulation study of accesses to two frequently accessed pages at UCLA.

Major differences in the gathering and use of usage statistics between Smiley and our work are that there is no method for clients to inform servers of their usage patterns, and that usage statistics are gathered not in real time but over many days and then they are not aged.

Although Smiley does not include a method for clients to inform servers of their usage patterns, client-side and server-side observations of usage patterns are merged in the simulation. Jiang and Kleinrock conclude that using server-side statistics in combination with those of the client yields a higher hit rate than using client-side statistics alone. A basic hypothesis of our work is that an individual client can prefetch accurately based upon usage statistics gathered from a large population. The Smiley results suggest that our hypothesis is sound.

Wcol [5] is a research prototype available on the Web. It prefetches embedded hyperlinks top-to-bottom without regard to likelihood of use. Embedded images of prefetched pages are also prefetched. Bandwidth waste can be capped by configuring Wcol to prefetch no more than a certain number of hyperlinks, and no more than a certain number of images embedded within prefetched hyperlinks.

PeakJet2000 [26] is the second major version of a commercial product. PeakJet runs on the client machine. It maintains a separate cache and provides a set of tools for speeding Web access, some of which require user action. True prefetching exists in two modes, "history-based" and "link-based." The user picks the mode. History-based prefetching prefetches an embedded link only if that client has used it before (i.e., it performs an IMS GET of a cached page). Link-based prefetching prefetches all embedded HREFs.

NetAccelerator 2.0 [24] is a similar commercial product for Windows clients. Unlike PeakJet, it prefetches into the browser's disk cache. Its prefetching algorithm is the same as PeakJet's link-based prefetching: all hyperlinks and their embedded images are prefetched.

2.2 Algorithms and Simulations

The method proposed by **Bestavros** in [2] is that "a server responds to a clients' [sic] request by sending, in addition to the data (documents) requested, a number of other documents that it speculates will be requested by that client in the near future."

A Markov algorithm is used to determine related "documents." The server examines the reference pattern of each client separately. Two documents are considered related if the client has accessed them in the past within a certain time interval. When a document is fetched, the server also pushes to the client any other document that is transitively related to it and whose likelihood of use is greater than a threshold value.

Usage statistics are gathered over 60 days, updated daily. Documents are considered related if requests from same client come within 5 seconds of each other. Among the results of this study are (1) that more recent usage statistics yield better results, as does more frequent updating of the "related" relation; (2) and "speculation is most effective when done conservatively." That is, with each incremental decrease in client latency, the extra bandwidth consumed and extra load imposed upon the server becomes greater.

The main point of the **Dynamic Documents** prototype [18] was to investigate how implementing documents as programs rather than static files might provide a means to help mobile clients adjust to enormous variations in bandwidth. Prefetching was added more as demonstration of a possibility rather than as a serious proposal. Pages in the history list are prefetched, with the result that bandwidth use is increased approximately 50% but only 2% of prefetched pages are used.

In the work of **Padmanabhan and Mogul** [25], a server maintains per-client usage statistics and determines related-ness through a graph-based Markov model similar to that of Bestavros. The graph contains a node for "every file that has ever been accessed" and is updated off-line, nightly for example. Related-ness is determined by edges through the nodes weighted by the probability that one will be accessed soon after the other. Whereas Bestavros defines "soon" by an amount of time, Padmanabhan and Mogul define it by a number of accesses from the client that occur in between.

When a GET is serviced, the server calculates a list of its pages that are likely to be requested in the near future, using some probability threshold. This list is appended to the GET response, and the client decides whether to actually prefetch. Another HTTP extension allows the client to indicate to the server that a certain GET is a prefetch, so that the server will not recursively compute related-ness for the prefetched page.

Trace-driven simulations show that average access time can be reduced approximately 40%, at the cost of much increased network traffic (70%). Another result suggests that prefetching is more beneficial than increasing bandwidth. That is, when prefetching causes a 20% increase in traffic, the latency is lower than it would be without prefetching but with 20% extra bandwidth. A third result is that prefetching increases access time variability, but very little.

The basic idea of **Top10** [21] is for servers (and proxies) to publish their most-accessed pages ("Top 10"). Downstream components (clients and proxies) prefetch some fraction of the list. The approach is parameterized two ways. One parameter indicates how many times a client must have contacted a server before it will prefetch at all. The other parameter indicates the maximum number of pages it will prefetch from a server. Results are by trace-driven simulation with traces from 5 sites. As more pages are prefetched, the percent of prefetched pages that are eventually used rises quickly and levels off at between 3% and 23%, depending on the trace. This suggests that the size of "TopN" should be small.

Fan et al. [13] evaluates several techniques for reducing client requests and observed latency. The evaluation is based mostly on trace-driven simulations of dialup users [15]. The authors have also implemented their prefetching ideas using CERN `httpd`. Pages are prefetched into a simulated browser cache only from a shared proxy cache, never from servers, so no extra wide-area traffic is generated. Consequently, prefetching is limited by the degree to which one client is expected to use a page that it or some other client sharing the same proxy has used in the past, and which is still in the proxy's cache.

Simulation results indicate that (perfect) prefetching and delta-compression [22] reduce latency considerably more than either HTML compression or merely increasing the size of the browser's cache. Using all three techniques with a finite browser cache resulted in only 30.3% latency reduction. However, prefetching reduced the number of client requests by 50%.[1] The 50% request savings, in turn, is limited by the fact that pages are prefetched only from the proxy.

Image objects were prefetched more often than HTML objects (64-74% versus 13-18%), and the prediction accuracy was higher for image objects (approximately 65% for JPEG and 58% for GIF versus 35% for HTML and "other" types). One possible explanation for this pattern is that this work uses a Markov-model prediction algorithm. Such algorithms view objects as independent, and often simply re-discover which images are embedded in an HTML page.

In the implementation, there is a proxy on the client side and one on the modem side. As in our work, the path of a request is browser to client-side proxy to modem-side proxy to server. Also like our work, The client-side proxy apparently piggybacks hit info on requests. After

[1] Latency savings are less than request savings because cached pages — those that can be prefetched — tended to be smaller than pages that had to be fetched from servers.

the modem-side proxy processes a request, it keeps the connection open, generates a list of URLs to prefetch, and "pushes" them into the client-side proxy's cache. The proxy prefetches only items in its cache. The predictor at the modem-side proxy remembers the client's last few requests but does not know the state of its cache, resulting in possible duplication.

Cohen and Kaplan [6] investigate three other types of prefetching: opening an HTTP connection to a server in advance of its (possible) use (pre-connecting); resolving a server's name to an IP address in advance of opening a connection to the server (pre-resolving); and sending a dummy request (such as a HEAD) to the server in advance of the first real request (pre-warming). Pre-connecting is motivated by the substantial overhead of TCP connection establishment. Pre-resolving is a subset of pre-connecting: the only part of a GET request done in advance is to translate the server's name into an IP address. Pre-warming is a superset of pre-connecting: its purpose is to force the server to perform one-time access control checking in advance of demand requests. In a trace-driven simulation, the three techniques reduced the number of "session starting" HTTP interactions whose latency exceeded 4 seconds from 7% to 4.5%, 2%, and 1%, respectively. This work represents a more conservative approach to prefetching than our own: much less complex, more likely to work without unintended consequences, and less capable of reducing latency.

Cunha's work [11] presents a very simple browser prefetch mechanism plus two mathematical models taken from prior work on other topics, that are used to indicate whether the mechanism should be invoked. His dissertation [12] provides additional detail not supplied in the paper, including recognition of the complications of file size and the need to age the usage information.

The prefetch model is that only the client is active in gathering usage information and making prefetch decisions. Hence, a user prefetches only pages that he has accessed before. The earlier references resulted in Markov chains with three types of links indicating whether objects that were accessed within a time window are unrelated, related by one being embedded in the other, or related simply by being likely to be accessed at about the same time.

The mathematical models – based on DRAM caches and linear predictive coding for speech processing – attempt to classify a user's behavior as "surfing" or "conservative." Surfing behavior references many different URLs whereas conservative behavior frequently re-references the same URLs. Very high hit rates (e.g., over 80%) are possible when the user's behavior fits the model.

2.3 Bounds

One of the results of the **Coolist** papers [27, 28] is a mathematical analysis of how accurate prefetch predictions must be (or, alternatively, how lightly loaded the network must be) in order for prefetches not to interfere with demand fetches and thereby *increase* the average latency of all fetches. The formula is $R < E/(1+E)$, where R is network utilization without prefetching and E is the ratio of hit rate with prefetching to traffic increase caused by prefetching. The papers also present a taxonomy of prefetching approaches that range from conservative to aggressive in their use of bandwidth, and "Coolist," a prefetcher that allows the user to choose the level of aggressiveness for prefetching user-specified groups of pages.

The surprising conclusion of **Crovella and Barford**'s trace-driven simulations [9, 10] is that prefetching makes traffic burstier and thereby worsens queueing. This is surprising because, generally speaking, a side effect of prefetching is to smooth short "spikes" of demand fetching into longer "trickles" of prefetching. The explanation lies in the definition of prefetching: at the start of a session *all* the files to be accessed in that session are prefetched immediately, creating an initial burst of demand. Crovella and Barford provide a solution, "rate-controlled prefetching," that smoothes out traffic. Rate-controlled prefetching approximates realistic prefetching, where the lookahead is limited. The analysis of rate-controlled prefetching also uses a better definition of prefetching – pages are prefetched one at a time and not with perfect accuracy. Such rate-controlled prefetching significantly reduces queue length over a wide range of prefetch accuracies.

An important study by **Kroeger et al.** [19] establishes bounds on the latency reduction achievable through caching and prefetching, under idealized conditions. Their most widely noted conclusion is that, even employing an unlimited cache and a prefetch algorithm that knows the future, at most 60% latency reduction can be achieved.

However, this study assumes (1) that "query events and events with `cgi-bin` in their URL cannot be either prefetched or cached" and (2) prefetching will always miss on the first access to a particular server. Neither is true in our work. The first assumption does not apply to our work because it is appropriate for a cache shared by several users, but our algorithm prefetches into a user-specific cache. Because many URLs represent queries or dynamic content (16.3% as shown in Section 3.2), removing this assumption in particular could yield an upper bound significantly above 60%.

3 Preliminary Experiments

This section describes preliminary experiments that were undertaken to reduce the number of unsupported assumptions and design decisions, and to test the extent to which pessimistic conclusions of certain earlier studies [4, 19] apply to this study.

Unfortunately, the data needed for the experiments described in this section is not present in well known existing logs such those from DEC [8], Boston University, Berkeley [15], and AT&T. Accordingly, we had to gather our own logs. A separate trace was gathered at AT&T Labs over several months in 1999. A "snooping" proxy produced, for every GET request, the following information: requesting client; URL; IMS request or not; `Referer` field, if present; time request was received by snooper; time first response byte was received; time last response byte was received; status code; Content-type field; lengths of header and content; time to expiration, and how computed; the number of embedded URLs; and whether the response would be cached and, if not, why not. In addition, all pages were permanently logged, including in cases where an unaltered proxy would not have cached it. It was necessary to log content because some experiments described in this section need it. For example, Experiment E determines, among other things, whether the URL of a referenced page is embedded as a hyperlink in any page accessed within the previous 30 seconds.

The trace consists of 92,518 references generated by members of the author's department (6 clients) over a span of about 5 months. All clients were attached to the same high speed LAN at 10Mb/sec. The LAN is attached to the Internet via a partial DS3. Drawing traces from a high speed environment is desirable because inter-reference times are likely to reflect the user's actual think time rather than bandwidth limitations of the local environment.

3.1 Cross-Server Links

Experiments A and B, respectively, address the assumptions in [19] that "prefetching can only begin after the client's first contact with that server" and that "query events and events with *cgi-bin* in their URL cannot be either prefetched or cached." Neither prohibition applies to this work. How significant are the effects of lifting these prohibitions?

Experiment A. The experimental question is: in those cases where one page refers to another, what fraction of referenced links (and bytes) are from sites different from the site of the referring page?

Analysis was done by parsing site information from the URLs. Sites are deemed to be different if both their names and IP addresses differ at the moment that the log analysis program runs. In some cases, months elapsed between the gathering of log data and the last run of the analysis program. It is assumed that during that time very few host pairs flip-flop between being same and different.

The results are that 28.9% of referenced hyperlinks, representing 19.9% of bytes, are to URLs on other servers. This seems high, and might reflect references skewed to commercial sites packed with advertisements on other sites.

3.2 Non-cacheable Links

Prefetch-ability in this work is not the same as cacheability in the literature. The reason is that the prefetcher brings pages into a *client-specific* cache. Because the cache is not shared, it is possible to cache cookies, query URLs and dynamic content. Studies of techniques for shared caches exclude such pages, with considerable effect. For example, one study [14] found that, in one sample, 43.1% of documents were uncacheable, mostly because of cookies (30.2%), query URLs (9.9%), obvious dynamic content (5.4%), and explicit cache-control prohibitions (9.1%).[2]

Experiment B. The experimental question is: what fraction of referenced links (and bytes) are query URLs or "obvious" dynamic content?

The definition of "obvious dynamic content" is derived from the one most often seen in the literature: a URL is assumed to specify dynamic content if it contains `cgi`. This heuristic is outmoded, as there are ever more tools for producing dynamic content, and these tools produce URLs by a variety of conventions, not just `cgi-bin`. The effect of using an old heuristic is that the proportion of dynamic content is underestimated, meaning that we err on the conservative side.

The results are that 16.3% of referenced hyperlinks, representing 18.2% of bytes, are to query URLs or to URLs containing `cgi`.

The results of experiments A and B are not meant to challenge or invalidate conclusions such as those in [19], because the bounds in that paper were developed under conditions that are unrealistically favorable. These results are meant to show merely that the theoretical limit to latency improvement via prefetching is probably higher than 60%, for two reasons. First, [19] assumes a shared cache and therefore does not prefetch certain pages that can be prefetching into a private cache. Second, [19] excludes the possibility of prefetching from a "new" site.

[2]The individual numbers do not add to 43.1% because some documents are uncacheable for several reasons.

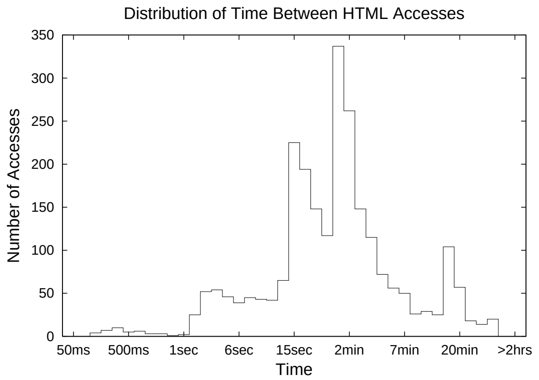

Figure 1: Time Between Referer and Referee (HTML)

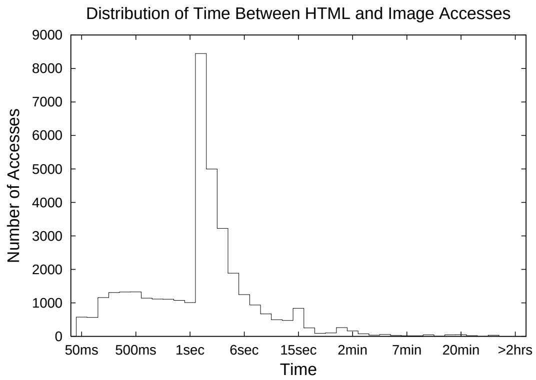

Figure 2: Time Between Referer and Referee (Image)

3.3 Inter-reference Time

Prefetching is not a purely algorithmic problem. Even given an oracle that could predict future accesses perfectly, prefetching results might be imperfect if a page were demanded during the time between the prediction of its need and its arrival. In such a case prefetching might still lower the observed latency, but conventionally such partial success is counted as a prefetching failure.

Experiment C1 determines the distribution of the time interval between referring and referred-to pages. Since Web data expires after a while, another timing issue is the distribution of expiration intervals. Because prefetched data can expire before it is demanded, prefetching too far in advance is also a problem. Experiment C2 determines the distribution of expiration times.

Experiment C1. The experimental question is: in those cases where one page refers to another, what is the distribution of the time between (1) the end of the transfer of the referer and (2) the beginning of the transfer of the referee? This is the amount of time available to the prefetcher.

The results are broken down into two categories: HTML referencing HTML (Figure 1), and HTML content referencing image content (Figure 1). The breakdown is intended to capture the difference between references to embedded hyperlinks and images, respectively. As shown, the browser-generated references to embedded images happen much more quickly than user-generated references to hyperlinks. The median inter-reference time between two HTML pages is 52 seconds, while inter-reference time between an HTML page and an image page is 2.25 seconds. However, a significant fraction of images (32.3%) are requested within one second or less of the referencing HTML page. This suggests that when an HTML page is prefetched, its embedded images should be prefetched too. Doing so substantially increases the penalty for being wrong.

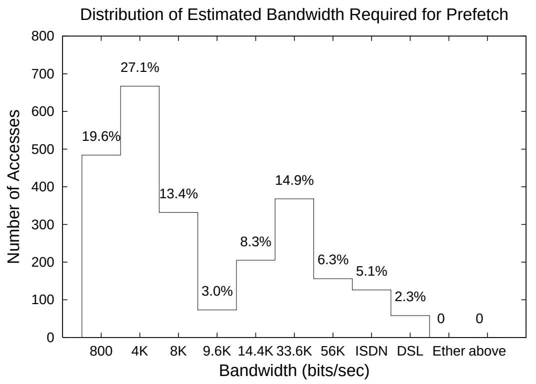

Figure 3: Bandwidth Required for Prefetching One Page

Figure 3 is the distribution of "total page size" divided by inter-reference time. That is, references in the log were analyzed to determine the size of an HTML page plus all its embedded images, whether or not they were actually referenced – an overestimate of what is needed to completely prefetch a page. Then this number was divided by the time between when that page's referrer was loaded and when the page was requested. This quotient is the maximum amount of data a prefetcher would have to deliver divided by the amount of time available to it: the bandwidth needed to prefetch one page. The median bandwidth is around 5KB/sec, safely within the capacity of even a dialup modem, suggesting that several pages could be prefetched while remaining within bandwidth constraints.

Experiment C2. The experimental question is: what is the distribution of expiration times, and how are the times computed? Prefetching can be harmful if prefetched content expires before it is demanded.

As Figure 4 shows, the most common expiration time

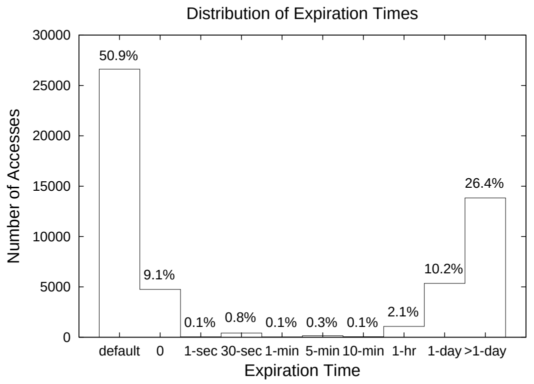

Figure 4: Time Until Expiration

is zero (60.0%). Typically (50.9%), zero is determined by default because the server has provided no information (`Expires` or `Last-modified` headers) upon which to base an estimate. The rest of the time (9.1%), the server has provided identical `Expires` and `Date` headers.

Strictly speaking, a zero expiration time should doom prefetching because prefetched content will be expired once it is demanded from the cache. However, since the meaning of a zero expiration time is "use only once," in our implementation we take the following steps. First, as explained in Section 4.1, a special header (`Prefetch: Prefetch`) is present in prefetched pages. The cache freshness-check code has been altered to ignore an explicit zero expiration (i.e., Expires = Date) when such a page has the prefetch header. When a such a page is eventually read, the cached page is re-written with the current time in the expires field and the `Prefetch: Prefetch` header overwritten.

The median non-zero expiration time (more than one day) far exceeds the average inter-reference time (52 seconds), indicating that too-early prefetching is not a practical concern. The median expiration time is so large because in many cases expiration is based only upon `Last-modified` headers, and many pages have not been modified in months.

3.4 The Naive Approach

The naive approach to prefetching based on embedded links [5, 26, 24] is to prefetch all embedded links or some number of them, without regard to the likelihood of use. Experiment D determines that some pages can have a very large number of embedded links, representing many bytes.

Experiment D. The experimental question is: what is the distribution of the number of links per page, how many bytes do these links represent, and what fraction of links/bytes are actually accessed?

As the snooper recorded the content of each page, it also parsed the pages and logged each embedded URL and its reference type (HREF, IMG SRC, etc.). The log analyzer determined which of these URLs were later accessed from that page. It also determined the size of each embedded HREF URL and its embedded images by calling a utility similar to the popular `webget`. Therefore, the sizes were determined, in some cases, weeks or months after the fact. It is assumed that size changes occurring the interim are negligible, or at least not substantially skewed in one direction or another.

The average number of links per page is 22.6. The average size of these links is 7760 bytes. The fraction of links and bytes accessed is very low (5.4% and 3.8%, respectively), again perhaps because of commercial portal sites that are packed with links. These results reinforce the importance of accurate prefetch predictions.

3.5 Markov Prediction Algorithms

Some prior work (e.g., [2, 13, 25]) has used Markov modeling to make reference predictions. A Markov prediction algorithm makes no use of structure information such as the `Referer` field, but instead regards earlier references as an unstructured string drawn from the "alphabet" of page IDs and attempts to discover patterns in the string. Recurrence of the initial part of a detected pattern triggers prefetching of the remainder of the pattern string. Markov algorithms seek patterns within a "window" of prior references, where the window typically is measured by time or number of references.

The two advantages of the Markov approach are that it does not depend upon the `Referer` field (which may not always be present) and that it can discover non-intuitive patterns. However, there are significant disadvantages to Markov algorithms. One is that such algorithms often have a high cost, measured in storage and compute time. The reason is that a Markov algorithm typically represents pages as graph nodes and accesses as weighted graph edges. With no structure to guide the search for patterns, predictive ability is improved by enlarging the window: retaining as many nodes as possible and searching as many paths as possible. Second, naive implementations of Markov algorithms have trouble aging their data. We hypothesize that Markov algorithms suffer a third drawback of being too general – they discover patterns that, to a large degree, are already obvious in the HTML of recently accessed pages and might be more simply extracted therefrom.

Experiment E compares the fraction of Markov "dependencies" among pages (that is, patterns where an access to page X tends to be closely followed by an access to page Y) that are also embedded links within recently ref-

Time (sec)	10	30	60	120
Embedded	70.0%	74.6%	76.0%	77.2%
References	1	3	5	10
Embedded	39.6%	66.3%	73.2%	78.3%

Figure 5: Percent of Markov Dependencies Also Present as Embedded Links Within Recently Accessed HTML Pages

erenced HTML pages.

Experiment E. The experimental question is: what fraction of "dependent" pages/bytes might also be easily detectable as embedded links? Two definitions of dependency are taken from much-cited works, [2] and [25]. Two references represent a dependency if they are made by the same client within a certain time or a certain number of prior references, respectively. As explained below, only a certain class of dependencies is considered. The results are shown in Figure 5.

The experiment computed, for every URL accessed, whether that URL was present as an embedded link within an HTML page that was previously referenced either within a certain time interval (10, 30, 60, and 120 seconds), or within a certain number of prior HTML pages (1, 3, 5, and 10). For each case, the table reports the fraction of URLs that were present as embedded links in those earlier HTML pages.

The interpretation of these results is not straightforward. First, there is no single "Markov algorithm" to compare to. For example, a Markov algorithm will declare two references to be dependent if they occur consecutively *with more than a certain probability*. Varying the threshold probability yields different algorithms. Second, Markov-based algorithms can find dependencies between any two pages, not just between an HTML page and another page. Therefore, Experiment E sheds some light on Markov algorithms versus our approach of examining links embedded in HTML, but it is not a definitive evaluation.

Experiment E considers only dependencies $X \rightarrow Y$, where X is an HTML page. In this subset of dependencies that a Markov algorithm might find, the results indicate that the (presumably) simpler method of inspecting HTML can locate a high percentage of dependencies. For instance, 39.6% of such dependencies can be discovered by inspecting only the single prior HTML page; 70.0% can be discovered by inspecting the HTML pages that were referenced in the preceding 10 seconds. These results are conservative, since it is not certain that any particular Markov algorithm would succeed in locating 100% of the dependencies.

4 Design

This section adds to Section 1 by explaining two topics in greater depth: the protocol used to exchange usage reports and usage profiles and the prefetching algorithm used by the client.

4.1 Information Exchange Protocol

One new HTTP header is defined, `Prefetch`. There are five Prefetch directives: `Negotiate`, `Report`, `Profile`, `Prefetch`, and `Halt`. Some directives take arguments. In particular, usage reports are provided as arguments to the `Report` directive and usage profiles are provided as arguments to the `Profile` directive.

Both client and server can limit the amount and define the format of prefetching information they exchange, and specify when such exchanges may take place. When initiated by a client, `Prefetch: Negotiate` should be sent along with a message to which a response is expected, such as the first GET. `Prefetch: Report` may be sent by the client along with any message after negotiation has finished. `Prefetch: Profile` is sent by the server only on GET responses, since the profile pertains to only those URLs whose body is included in the message.

The `Negotiate` directive can be specialized through a number of arguments that specify whether usage reports will be sent occasionally, periodically, after a certain number of page accesses, or once the usage report reaches a certain size.

The most common use of `Negotiate`, `Report`, `Profile`, and `Prefetch` directives is as follows:

```
client               server
------               ------
   ------------------->
       negotiate
   <-------------------
       negotiate
          ...
   ------------------->
        report
          ...
   <-------------------
        profile
          ...
   ------------------->
        prefetch
   <-------------------
        prefetch
```

Three other arguments to `Negotiate` specify the desired format of future usage profile directives. One argument indicates how to age the data. For example, `last=10,20,50` means that the server's usage profile should, if possible, summarize which embedded references have been used in the last 10, 20, and 50 references to the page. The server is bound only to make a best effort to retain enough data to deliver usage profiles in the format it has negotiated. The other two arguments specify limits on the size of the usage profile: absolute byte count and relative to the attached GET response.

4.1.1 Record Format

A usage report includes the referring and referred-to URLs, when the request took place, how long it took to satisfy, and the size of the result. Only successful references to HREFs are reported.

Usage profiles comprise two sorts of records. One is `Last N` where N is an integer. The other sort of record is similar to a usage report prefaced by an integer, M. The two types of records are intermixed, with one `Last N` record followed by some number of "`M...`" records. This pattern repeats. The meaning of such a sequence of records is that the last N times this page was referenced, it happened M_i times that URL_i was referenced by that page. The sum of M_i need not equal N.

Auxiliary information, such as the elapsed time between references to P and some embedded HREF Q, appear in the summary as medians plus standard deviations.

4.2 Prefetching Algorithm

The client-side prefetch algorithm that has been used for the measurements in this paper is the following. During negotiation with a server, the client asks for the usage profile to summarize the last 10 and 50 references. The client continually measures the speed of its HTTP GET transfers, and maintains a running average; the speed varies depending on many factors so the data is inaccurate, but inaccurate data is regarded as better than no data. Whenever a page is demand-fetched, its embedded HREFs are noted during the parsing necessary to display it. These HREFs are put on a list and the list is passed to the prefetcher along with the usage profile that came in the HTTP header. After page display is complete, the prefetch algorithm runs, comparing the embedded hyperlinks with the usage profile. The comparison ensures that a recently deleted hyperlink mentioned in the usage profile will not be prefetched. The usage profile indicates the size of embedded HREFs (including the size of embedded images), and the prefetcher ensures that it never has GET requests outstanding for more than a certain fraction of the measured average bandwidth available to it: 50%, to be safe considering the inaccuracy of the bandwidth measure. Until the bandwidth limit is reached, the client prefetches URLs that have been accessed among the last 10 references, in descending order of popularity, down to a limit of 25%. That is, if the chances of a page being accessed are less than one quarter, that page is not prefetched. A prefetch request is not complete until the HTML page and all its embedded images have been loaded. A demand fetch aborts all prefetching efforts. As prefetch requests complete, more are issued from the "last-10" list and then from the "last-50" list, again in descending order of popularity down to the limit.

5 Implementation

There are two implementations of this approach to prefetching. In both, the server side is implemented by a proxy. The proxy emulates all the Web servers in the world, keeping track of usage reports and forming usage profiles for all URLs of all Web servers. This was done for testing and debugging purposes so that every Web server could seem to be one that supports prefetching. In practice, a server-side proxy might serve only a single server. The server-side proxy is an altered version of the W3C's `httpd`, version 3.0A.

The two implementations are distinguished by the client side. One implementation is an altered version of the September 4 1998 version of Mozilla which runs as multi-threaded software on UNIX. The other implementation is a proxy, once again an altered version of `httpd` 3.0A.

An important optimization is presently missing from the implementations: there is no need for a client to send a usage report to a server if both referer and referee URLs are on that server. The server (or its proxy) can determine the same information itself provided that the client sends the `Referer` field.

5.1 Evaluation

We have characterized prefetching performance through five measures: prefetch accuracy, client latency, network overhead, program space overhead, and program time overhead. All numbers are taken from the Mozilla implementation.

Mozilla has been altered to read the trace log and replay the HTML accesses with timing that is faithful (insofar as possible) to that in the log. The pages are displayed completely, just as if the user had typed in the same references with the timing evident in the log. The latency and prefetch accuracy numbers were gathered using this mechanism. Latency reduction is calculated by comparing the time between (1) Mozilla's initiation of an HTML page GET, and (2) Mozilla's receipt of the end of the last embedded image for that page. This time is compared to the similar time taken from the trace log. The two times are not exactly comparable because the trace log records the time for a client-side snooping proxy to communicate with a remote web server, whereas Mozilla is recording the time for it to communicate with a server-side proxy which then communicates with the remote web server. However, since this approach places the prefetching implementation at a disadvantage – three parties communicate in series instead of two – we ignore the difference. Mozilla's disk cache size was kept at the default 5MB.

Prefetch Accuracy. The observed prefetch accuracy is high: while less than a majority of prefetched HTML pages (42.6%) are eventually accessed, a much higher fraction of all pages (62.5%) are eventually used. The reason is that many links from a common page share embedded images. So if page X has embedded links Y and Z, and if Y is wrongly prefetched instead of Z, considerable savings may still result if Y and Z share many embedded images. The overall increase in network traffic is considerably smaller (24.0%) than the overall prefetch miss rate of 37.5% because of demand fetches.

The restraint exercised by the prefetch algorithm – not prefetching links that have less than 25% chance of being accessed – governs the tradeoff between lowering latency and wasting bandwidth. Twenty-five percent was found to be the optimum point (from among every five percent) for balancing bandwidth waste against latency reduction.

Latency. Prefetching decreases average total latency to display an HTML page and all its embedded images by more than half (52.3%). This number probably lies between the prefetched-HTML hit rate (42.6%) and the overall prefetched hit rate (62.5%) because, on average, image pages are smaller than HTML pages.

Network Overhead. Usage reports and usage profiles can be lengthy. The average usage report is 66 bytes, while the average usage profile is 197 bytes. Most space is taken up by URLs, especially query URLs. It might be practical to abbreviate URLs in the usage profile, since the same URLs are embedded in the accompanying page content. However, no such method has been investigated.

Space Overhead. Some extra fields have been added to Mozilla's data structure that describes a page; however, this structure is very large and the added space is negligible. On the server side, the proxy's data structures grow in proportion to the number of pages distinct pages served.

Time Overhead. The effect of prefetch code on Mozilla's critical path is negligible, mostly because Mozilla executes a great deal of code for every GET operation. Time added at the server proxy is also negligible. The data structures for maintaining usage profiles are kept in memory, requiring added space but no extra disk accesses until they grow very large.

5.2 Implementation Effort

Mozilla was the largest and most difficult implementation, with a total of 3581 lines of code added or changed:

- Hooks into the parser to discover URLs: 47
- Response timing: 113
- Accumulating usage reports, tracking the state of negotiations with all servers: 786
- Prefetch algorithm: 1312
- Connection management: 751
- Cache management to address the Expires=0 problem: 450
- Short circuit of front-end display code so that pages are fetched but not displayed: 122

Three variations of the W3C `httpd` have been produced: a client-side prefetching proxy, a snooping proxy, and a server-side proxy that maintains usage statistics for all servers. The client-side proxy is the most complicated, though some code is shared with Mozilla; compared to Mozilla there are no front-end complications, and connection management is much easier. The snooping and server-side proxies are simple.

5.3 Privacy Implications

In order to operate transparently, the prefetching mechanism must examine HTML and `Referer` headers. This raises several privacy issues. The first is that the HTML must be available. End-to-end protocols or tunnels that might encrypt, compress, difference, or otherwise obscure HTML content could make the proxy implementation impossible. Second, some privacy advocates are concerned about the `Referer` field and want, at minimum, to be able to configure browsers not to send it. This goal is in conflict with our prefetching mechanism: suppressing the `Referer` field makes a proxy implementation impossible, while having a browser implementation send usage reports defeats `Referer` suppression. Indeed, a third privacy issue is that usage reports contain strictly more information than the `Referer` field. A fourth issue is that some users might feel uneasy knowing that the prefetcher examines their pages and browsing patterns. In this regard, the prefetcher does not pose a threat that is not already present from proxies, firewalls, and servers; nevertheless, knowing that the prefetcher systematically parses their pages might make some users uncomfortable. Finally, perhaps the largest privacy issue is that prefetching depends upon server administrators agreeing to release statistics about how their pages are being used. In many cases such information has commercial value, meaning that web site operators might refuse to release it or demand payment for it.

6 Conclusion

We have shown that accurate Web prefetching is possible based on following HREFs in recently fetched pages. Letting the client control prefetching and aging of usage statistics has many advantages and may be the only practical approach in a world where proxies are commonplace. However, placing control at the client is also problematic because many pages contain large numbers of hyperlinks, and simply prefetching them all is worse than nothing. Also the client cannot be expected to have a good understanding of HREF reference patterns unless the page is one read frequently and/or the page rarely changes.

Our approach to this problem is to have clients pass record of their references up to the relevant server, which then distributes them to all clients. It is hypothesized that HREFs within a page are strongly skewed to "hot" and "cold," so that one client can learn from the usage patterns of others. The results in Section 5.1 bear this out.

The information exchange between clients and servers complicates deployment; however, there are other examples of recent work that depend on a similar flow of information [23, 7], suggesting that the idea may be useful more generally.

References

[1] V. Almeida et al. Characterizing Reference Locality in the WWW. In Proc. IEEE Conf. on Parallel and Distributed Information Systems. IEEE, December 1996. `http://www.cs.bu.edu/~best/res/papers/pdis96.ps`

[2] A. Bestavros. Using Speculation to Reduce Server Load and Service Time on the WWW. In *Proc. 4th ACM Intl. Conf. on Information and Knowledge Mgmt.*. ACM, November 1995. `http://www.cs.bu.edu/~best/res/papers/cikm95.ps`

[3] A. Bestavros and C. Cunha. Server-Initiated Document Dissemination for the WWW. *IEEE Data Engineering Bull.*, 19(3):3–11, September 1996. `http://www.cs.bu.edu/~best/res/papers/debull96.ps`

[4] R. Caceres, et al. Web Proxy Caching: The Devil is in the Details. *ACM SIGMETRICS Performance Evaluation Rev.*, 26(3):11–15, December 1998. `http://www.research.att.com/~ramon/papers/wisp98.ps.gz`

[5] K. Chinen and S. Yamaguchi. An Interactive Prefetching Proxy Server for Improvement of WWW Latency. In *Proc. INET 97*, June 1997. `http://www.isoc.org/inet97/proceedings/A1/A1_3.HTM`

[6] E. Cohen and H. Kaplan. Reducing User-Perceived Latency by Prefetching Connections and Pre-warming Servers *Unpublished AT&T technical report*, February 1999.

[7] E. Cohen, B. Krishnamurthy, and J. Rexford. Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters. In *Proc. ACM SIGCOMM 98*, pages 241–253. ACM, September 1998. `http://www.research.att.com/~edith/Papers/sigcomm98.ps.Z`

[8] Traces of Corporate Web Proxies. Compaq Corp. `ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`

[9] M. Crovella and P. Barford. The Network Effects of Prefetching. In *Proc. Infocom 98*. IEEE, April 1998. `http://cs-www.bu.edu/faculty/crovella/paper-archive/infocom98.ps`

[10] M. Crovella and P. Barford. The Network Effects of Prefetching. Technical Report TR-97-002, Boston University, February 1997. `http://cs-www.bu.edu/techreports/97-002-prefetcheff.ps.Z`

[11] C.R. Cunha and C.F.B. Jaccoud. Determining WWW User's Next Access and Its Application to Pre-fetching. In *Proc. Second IEEE Intl. Symp. on Computers and Communication '97*. IEEE, July 1997. `http://cs-www.bu.edu/techreports/97-004-userbehaviorprediction.ps.Z`

[12] C.R. Cunha. Trace Analysis and Its Applications to Performance Enhancements of Distributed Information Systems. PhD Thesis TR-97-004, Boston University, 1997. `http://www.cs.bu.edu/students/alumni/carro/thesis.ps.Z`

[13] L. Fan et al. Web Prefetching Between Low-Bandwidth Clients and Proxies: Potential and Performance. In *Proc. ACM SIGMETRICS Conf.*, pages 178-187. ACM, May 1999. `http://www.cs.wisc.edu/~cao/papers/prepush.ps.gz`

[14] A. Feldmann et al. Performance of Web Proxy Caching in Heterogeneous Bandwidth Environments. In *Proc. Infocom 99*. IEEE, March 1999. http://www.research.att.com/~ramon/papers/infocom99.proxy.ps.gz

[15] Traces of Dialup Users. S. Gribble. http://ita.ee.lbl.gov/html/contrib/UCB.home-IP-HTTP.html

[16] Z. Jiang and L. Kleinrock. Prefetching Links on the WWW. In *ICC 97*. June 1997. http://millennium.cs.ucla.edu/~jiang/Research/Publication/prefetch.ps

[17] Z. Jiang and L. Kleinrock. An Adaptive Network Prefetch Scheme. *IEEE Journ. Selected Areas of Communication*, 17(4):358–368, April 1998. http://millennium.cs.ucla.edu/~jiang/Research/Publication/extended.ps

[18] M.F. Kaashoek, T. Pinckney, and J.A. Tauber. Dynamic Documents: Mobile Wireless Access to the WWW. In *Wkshp. on Mobile Computing Systems and Applications*, pages 179–184. IEEE, December 1994.

[19] T.M. Kroeger, D.D.E. Long, and J.C. Mogul. Exploring the Bounds of Web Latency Reduction from Caching and Prefetching. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 13–22. USENIX, December 1997. http://www.usenix.org/publications/library/proceedings/usits97/kroeger.html

[20] T.S. Loon and V. Bharghavan. Alleviating the Latency and Bandwidth Problems in WWW Browsing. In *Proc. USENIX Symp. on Internet Technologies and Systems*, pages 219–230. USENIX, December 1997. http://www.usenix.org/publications/library/proceedings/usits97/tong.html

[21] E.P. Markatos and C.E. Chronaki. A Top-10 Approach to Prefetching on the Web. In *Proc. INET 98*. July 1998. http://www.ics.forth.gr/proj/arch-vlsi/html_papers/INET98_prefetch/paper.html

[22] J. Mogul et al. Potential Benefits of Delta Encoding and Data Compression for HTTP. In *Proc. ACM SIGCOMM 97*, pages 181–194. ACM, September 1997. ftp://ftp.digital.com/%7emogul/sigcomm97.ps.gz

[23] J. Mogul and P. Leach. Simple Hit-Metering and Usage-Limiting for HTTP. *IETF Network Working Group RFC 2227*, October 1997. http://www.ietf.org/rfc/rfc2227.txt

[24] NetAccelerator 2.0 software. http://www.imsisoft.com/netaccelerator/netacc2.html

[25] V. Padmanabhan and J. Mogul. Using Predictive Prefetching to Improve World Wide Web Latency. *Computer Communication Rev.*, 26(3):22–36, July 1996. http://www.cs.berkeley.edu/~padmanab/papers/ccr-july96.ps

[26] PeakJet2000 software. http://www.peak.com/peakjet2long.html

[27] Z. Wang and J. Crowcroft. Prefetching in World Wide Web. In *Proc. Globecom 96*. IEEE, December 1996. http://www.bell-labs.com/user/zhwang/papers/prefetch.ps.Z

[28] Z. Wang and J. Crowcroft. Prefetching in World Wide Web. In *Proc. Global Internet*, pages 28–32. IEEE, November 1996. http://www.cs.columbia.edu/~hgs/InternetTC/GlobalInternet96/Wang9611_Prefetching.ps.gz