AT&T Labs Research

subject: **Improving End-to-End Performance of the Web Using Server Volumes and Proxy Filters**

date: **February 6, 1998**

from: **Edith Cohen**
edith@research.att.com

Balachander Krishnamurthy
bala@research.att.com

Jennifer Rexford
jrex@research.att.com

TECHNICAL MEMORANDUM

The rapid growth of the World Wide Web has caused serious performance degradation on the Internet. This paper offers an end-to-end framework by collectively examining the Web components – clients, proxies, servers, and the network. Our goal is to reduce user-perceived latency and the number of TCP connections, improve cache coherency and cache replacement, and enable prefetching of resources that are likely to be accessed in the near future. In our scheme, server response messages include *piggybacked* information customized to the requesting proxy. Our enhancement to the existing request-response protocol does not require per-proxy state at server or per-server state at the proxy, and can be implemented without changes to HTTP 1.1. The server groups related resources into *volumes* (based on access patterns and the file system's directory structure) and applies a proxy-generated *filter* (indicating the type of information of interest to the proxy) to tailor the piggyback information. We present efficient data structures for constructing server volumes and applying proxy filters, and a transparent way to perform volume maintenance and piggyback generation at a router along the path between the proxy and the server. We demonstrate the effectiveness of our end-to-end approach by evaluating various volume construction and filtering techniques across a collection of large client and server logs.

Keywords: Web, piggybacking, caching, coherency, prefetching, volumes, filters

1 Introduction

The exponential rate of growth of the World Wide Web has led to a dramatic increase in Internet traffic, as well as a significant degradation in user-perceived latency while accessing "Web pages." Web servers store hundreds or thousands of pages each of which consist of one or more typed resources (text, inline images, applets, etc.). As the popularity of the Web increases, these servers support an increasing number of requests to transfer resources and validate copies that are cached in other locations. Additionally, these data transfers impart a heavy load on the network links and routers between the server and client sites. The round-trip delays in establishing a TCP connection and initiating a request, as well as the waiting time at the server and the limited network bandwidth for transferring the response message, translate into user-perceived latency. In this paper, we offer an end-to-end approach to improving Web performance by collectively examining the various Web components – client, proxies, servers, and the network. We present efficient techniques for organizing and transmitting useful information about resources at the server, tailored to the needs of the contacting proxy.

Previous research has focused on improving the performance of individual components in the web, with limited use of the information present in other components. Beginning with proxies, we discuss each of the Web components to examine the potential use of external information. Proxies have emerged as an important intermediary between large groups of clients and servers. In the process of relaying traffic between client browsers and remote servers, a proxy can cache resources in the hope of satisfying future client requests directly at the proxy. Since many resources are requested multiple times by one or more clients, proxy caching can greatly reduce user-perceived latency, as well as the load on the network and the server. Recent performance studies have shown that proxy caches have a 30–50% hit rate [1,2] . New cache allocation and replacement schemes improve cache performance by extending the conventional LRU policy to incorporate resource size and other notions of cost [3–5] . Cache performance is constrained by the proxy's limited knowledge about resources that are likely to be requested in the near future and their cachability.

Even when the cache can satisfy the client request, the proxy often has to validate the cached resource to avoid returning out-of-date information. Often, the proxy enforces a time-to-live on the cached resource to limit the likelihood of returning a stale copy [6,7] . Beyond the time-to-live period, the proxy must check with the server to see if the resource has changed, which imposes additional load on the server and increases user-perceived latency. In fact, studies show that 15-25% of *all* server response messages are simply Not Modified responses to validate cached resources [8,9] . To avoid the latency and overhead of establishing a TCP connection for each Web transfer, extensions to HTTP permit proxies to maintain persistent connections to servers, which enable pipelining of multiple requests and responses [10–12] . For example, a persistent connection permits embedded images in a HTML document to be downloaded without new connections. To reduce the number of TCP connections, a proxy can allow multiple clients to share a single persistent connection to a server, and allow multiple servers to share a single connection to a client. Although a uniform timeout mechanism (say, 60 seconds) can be used to close persistent connections, this approach does not favor proxies that are likely to generate requests in the near future.

Recent studies have begun to consider the potential benefits of using server information to enhance the proxy's policies. For example, the server can aid the proxy in cache allocation and replacement decisions by sending estimates of the time between successive accesses to a resource [13] . Similarly, using access patterns to predict future requests, the server can speculatively disseminate resources to the proxy [14] or guide proxy prefetching decisions [15–17] , at the expense of increasing the load on the network and the server. In addition, the server can improve cache coherency by sending a list of resources that have been modified [18,19] or validating a list of cached resources at the proxy [9] . In this paper, we focus on an *end-to-end* information exchange that involves both the proxy and the server, with an emphasis on the following key practical questions:

- What additional information should be exchanged?
- How can this information be maintained and exchanged efficiently?
- How can new Web policies exploit this information?

An efficient exchange between servers and proxies is enabled by the server *piggybacking* information about its resources onto regular response messages, avoiding establishment of any new TCP connections; in fact, small piggyback messages can often be transmitted without requiring any new packets. The server constructs *volumes* of related resources, based on the file system's directory structure and the likelihood that pairs of resources are accessed together. A proxy-generated *filter* is applied to customize the piggyback information to the requesting proxy. Our generalized piggyback information exchange enables a proxy to periodically query and learn about interesting events that occur on selected server sites. The key contributions of our approach are:

- *Scalable protocol:* We enhance the request-response exchange between the proxy and the server without requiring changes to the underlying HTTP protocol. Our mechanisms do not require per-proxy state at the servers, per-server state at the proxies, or new TCP connections.
- *Server volume:* We propose and evaluate techniques for servers to group related resources into volumes, based on the file system's directory structure and the likelihood that resources are accessed together, and combinations of these schemes. We present efficient data structures and algorithms for constructing and maintaining volumes and applying proxy filters.
- *Volume thinning:* Grouping based on resources accessed together can result in large volumes which need to be trimmed and tailored depending on the proxy application. Since constructing *optimal* volumes is NP-complete, we introduce heuristics for thinning volumes to improve accuracy and reduce redundancy in the information sent. Our thinned volumes perform significantly better than previous work in this area.
- *Proxy filter:* We introduce several methods to tailor volume information to the proxy in order to reduce the frequency and size of piggyback messages. We present efficient proxy data structures for constructing these filters based on knowledge of recent volumes piggybacked by the server.
- *Transparent volume center:* We describe how volume maintenance and piggyback generation can be performed transparently at a router or gateway along the path between the proxy and server. This volume center can construct volumes and apply filters on behalf of several servers, allowing piggyback messages to include information about resources at multiple sites.

To illustrate the usefulness of our proposed scheme, we present several example proxy policies for cache coherency, cache replacement, and prefetching that exploit the information in the piggyback messages. The performance of these policies depends on generating accurate predictions with minimal overhead caused by the piggyback traffic. Hence, our performance evaluation focuses on the volume construction, filtering, and piggybacking techniques. While previous performance studies of the Web have focused on either server logs or client/proxy traces, we evaluate a collection of both client logs (from Digital Equipment Corporation and AT&T) and server logs (from Amnesty International USA, Apache Group, Marimba Corporation, and Sun Microsystems). Unfortunately, any one proxy trace or server log does not have sufficient information for an end-to-end evaluation of any particular application of our protocol, since the proxy and the server each lack knowledge of activity at the other site. In the absence of a coordinated end-to-end trace, we evaluate our protocol by selectively joining the information available in the client and server logs.

The rest of the paper is organized as follows. Section 2 describes how the proxy and server piggyback information on request and response traffic, respectively, and how the proxy generates filters to customize the piggyback messages from the server. Section 3 presents several techniques for the server to construct volumes, based on the file-system's directory structure and the likelihood that pairs of resources are accessed together. Section 4 presents an in-depth performance evaluation of our proposed volume-construction, filtering, and piggybacking techniques based on a collection of proxy and server logs. These results are used to discuss several proxy applications that exploit the piggybacked information in Section 5. Section 6 discusses practical issues concerning the deployment of our piggyback scheme within HTTP 1.1, as well as the operation of the transparent volume center. Section 7 concludes the paper with a discussion of future research directions.

2 Exchanging Filter and Volume Information

Our generalized piggybacking protocol employs proxy filters and server volumes to generate customized information without maintaining complex data structures or requiring new TCP connections. After presenting the information exchange, we describe how the proxy controls the frequency and size of piggyback messages, and tailors their contents to the characteristics of the cache.

2.1 Piggybacking Protocol

Additional information about resource characteristics and access patterns at the server could improve the effectiveness of proxy policies for cache replacement, cache coherency, and prefetching. The server has considerable knowledge about each resource, including the size and content type, as well as the frequency of resource modifications. By accumulating information about requests from a large number of proxies and clients, the server can gauge the popularity of each resource and the likelihood that certain resources are accessed together. On the other hand, the proxy has knowledge about its pool of clients and their access patterns, including the requests that are satisfied without contacting the server. In addition, the proxy has information about the size of its cache, as well as the policies for cache allocation, replacement, and coherency. The proxy may not cache certain types of content (such as images) or resources that exceed a certain size.

To bridge the knowledge gap between servers and proxies, we propose that servers should send information about their resources customized to interested proxies. However, the exchange of this additional information should not impose an excessive burden on any of the web components. Though the server may measure access patterns across its collection of resources, the server cannot afford to maintain state for each proxy; similarly, the proxy cannot afford to maintain state for all servers. To limit the load on the network, this additional information exchange should not significantly increase the bandwidth consumption or the number of TCP connections. The combination of proxy filters, server volumes, and piggybacking provides an effective way to send useful information to the proxy without requiring complex data structures. Each piggyback element contains the identifier, size, and Last-Modified time of a single resource at the server, from the same volume as the requested resource. The proxy filter controls how many piggyback elements are received, as well as which resources are included in the list.

The proxy stores the unique resource identifier of each resource r in the cache, along with the Last-Modified time (indicating the version of the resource at the server) and the expiration time (indicating when the cached resource requires validation before use). To illustrate the operation of the piggybacking protocol, we describe the handling of a client GET request for a resource r:

Proxy receives a client request: If the cache has a copy of resource r and the expiration time has not been reached, the proxy returns the resource directly to the client. A cache miss triggers a regular GET request to the server, and a cache hit with an *expired* copy of r generates a GET request with an If-Modified-Since modifier and the Last-Modified time of the proxy's copy of the resource. In either case, the proxy piggybacks a filter onto the GET request to aid the server in customizing the volume information for resource r.

Server receives a proxy request: If the server receives a GET request with an If-Modified-Since modifier, and if the proxy-specified Last-Modified time is greater or equal to the Last-Modified time at the server, the server simply validates the resource by sending a Not Modified response. Otherwise, the server transmits an OK response with a fresh version of the resource as it would in the case of an unmodified GET request. In either case, the server constructs a piggyback message with the volume id and information about related resources based on r and the proxy-specified filter. An element in the piggyback list includes a resource identifier p, its Last-Modified time, and size.

Proxy receives a server response: Upon receiving the server response message, the proxy returns the resource to the client and updates the cache. When caching a new copy of r as part of an OK response, the proxy saves the Last-Modified time and assigns the expiration time (Δ seconds in the future, where Δ is the freshness interval). The proxy also updates the expiration time upon receiving a Not Modified response from the server. Next, the proxy processes each resource p in the piggyback list. If the proxy has a fresh copy of p, the expiration time is updated. If the cached resource is out-of-date, the proxy can delete the stale copy or prefetch a new version of the resource. The proxy can also prefetch p if it is not in the cache.

2.2 Proxy Filters

Despite the potential benefits of the piggyback messages, a direct application of the protocol may repeatedly send the same information to the proxy when there are several accesses to the same server in a short period of time. This introduces unnecessary overhead on the server, the proxy, and the communication network. To avoid excess traffic, the protocol should limit the frequency of piggyback messages to each of the proxy sites. If the server were to explicitly regulate the transmission of piggyback messages by tracking recent transmissions to each proxy, it would have to maintain per-proxy state. Instead, the protocol provides an effective way for the proxy request messages to implicitly control the pacing of piggyback responses from the server. To control piggyback traffic without maintaining state, the proxy can randomly set an enable/disable bit, or employ simple frequency control techniques, such as disabling piggybacks from servers which have sent piggybacks within the last minute. The frequency control techniques can be randomized or augmented with information about usefulness of recently piggybacked responses. These techniques do not require the proxy to maintain per-resource or per-volume state, making them particularly efficient for servers with a large number of volumes, as would occur with probability-based volumes discussed in Section 3.2.

When the number of volumes is small, the proxy stores a list of recently piggybacked volumes (RPVs) for each server, or for a subset of popular servers. Each list element includes the volume identifier and the time the last piggyback message for that volume was received. The proxy can limit the RPV list based on a timeout or a maximum size basis; and maintain them efficiently as FIFO lists in a hash table keyed on the server IP address. The proxy includes the RPV list in its filter and the server decides if information should be piggybacked or not. A piggyback reply includes the resource's volume identifier, which can be added to the RPV list at the proxy. The appropriate time interval for removing an element from the RPV list depends on the freshness interval Δ in the cache, as well as the frequency of accesses to the server. For example, the proxy should not keep a volume in an RPV list for longer that Δ time units, since this would preclude the server from sending refresh information for resources in this volume. Even smaller time intervals are appropriate to further improve the freshness of the proxy cache, at the expense of additional piggyback traffic.

In addition to limiting the frequency of piggyback traffic, the proxy filter customizes the contents of each piggyback message. Even though server volumes are constructed from anticipated or measured access patterns, these volumes are likely to include some information that is not useful to a given proxy. Different proxies may serve vastly different communities of users, and impose a wide range of policies that affect the type of piggyback information that is useful. The proxy can customize the piggyback message by specifying a maximum number of elements, a limit on the size or content type of the resources included in the list, or a minimum threshold of access frequency or probability. For example, clients on low-bandwidth wireless links are likely to disable the transfer of images and avoid downloading large resources; hence, a proxy serving such a group does not need to receive piggyback information for these items. Similarly, a proxy that must provide up-to-date information, such as stock quotes for business clients, may decide to disable caching of resources that change frequently; since the proxy always contacts the server directly to handle these client requests, the server need not piggyback information about rapidly changing resources.

3 Server Volumes

The effectiveness of the piggybacking protocol depends on the server's ability to group related resources into volumes. Upon receiving a request for a resource, the server should be able to generate a list of other resources that are likely to be requested in the near future. In evaluating volume construction techniques, we consider three key metrics that relate to how various applications would use the piggybacked information:

- *Prediction fraction:* The likelihood that a resource requested at the proxy has appeared in at least one piggyback message in the last T seconds. This metric captures the *recall* of the prediction scheme, or the fraction of client accesses that can capitalize on recent piggyback information, and is important for all of the proxy policies.
- *True prediction fraction:* The likelihood that a resource that appears in a piggyback message will be accessed by a client in the next T seconds. Resources that appear in multiple piggyback messages in the same time interval are counted as a single prediction. This metric captures the *precision* of the volume-construction technique, and is important for prefetching policies.
- *Update fraction:* The likelihood that a resource requested at the proxy has been predicted in the last T seconds *and* appeared in a previous request in the last C seconds, where $C > T$. This metric captures the likelihood that the client request accesses a *cached* resource that has been updated by a recent piggyback message, and is important for cache coherency and cache replacement policies.

In addition to maximizing these three performance metrics, a good volume-construction scheme should limit the average size of piggyback messages. An optimal volume would maximize the fraction predicted or the fraction updated. However, constructing optimal volumes subject to a constraint on either the average piggyback size or the fraction of true predictions, is an NP-complete problem (the proof is presented in the Appendix). Instead, we present heuristics for grouping resources into volumes, and use these performance metrics as the basis of our evaluation in Section 4.

3.1 Directory-Based Volumes

Our heuristics must group related resources into volumes, and rank the resources to determine which information should be sent when a proxy limits the size of the piggyback message. Each volume consists of one or more volume elements, where each element includes a resource identifier, size, and Last-Modified time. The simplest approach to maximizing the fraction of requests that are predicted in advance is to combine all of the server's resources into a single site-wide volume [19] . To reduce the volume size, the server can group resources with the same directory prefix in their pathnames, up to some number of levels. For example, one-level volumes would assign `www.foo.com/a/b.html` and `www.foo.com/a/d/e.html` to the same volume, though `www.foo.com/f/g.html` would belong to a different volume; if volumes were based on zero-level prefixes, all three resources would belong to the same volume. Directory-based volumes are based on the heuristic that resources in the same directory or subdirectory are likely to have related content and/or occur as embedded HREF links in the same (or related) Web pages.

Based on a static volume definition, the server can maintain volume elements in a collection of FIFO lists partitioned by resource sizes and content type. For example, the volume could have one list for large images, and another list for small text pages, since the proxy filter can specify that the server piggyback message include popular items of certain content types and sizes. Using the last-access-time as the popularity metric for adding, removing, updating, and filtering volume elements, permits constant-time operations for maintaining the volumes at the server. An approximate way to rank volume elements in order of popularity is using *move-to-front* semantics to place a requested resource at the head of its FIFO; this ensures that piggyback messages include the most recently accessed elements in the volume. The server can control the size of volumes by removing unpopular entries from the tail of the logical FIFO. In Section 4.2, we evaluate the effectiveness of applying a popularity filter to directory-based volumes.

3.2 Probability-Based Volumes

Instead of defining volumes based on a static heuristic, such as the directory structure in resource pathnames, the server can construct volumes by measuring access patterns. By observing a stream of requests, the server can estimate the pairwise dependencies between resources[1]. Let $p_{s|r}$ be the proportion of requests for resource r that are followed by a request for resource s by the same source within T seconds. Resource s is included in r's volume if $p_{s|r}$ is greater than or equal to a threshold probability p_t. When a proxy requests resource r, the server constructs a piggyback message from the set of resources s with $p_{s|r} \geq p_t$. The server can estimate the probabilities $p_{s|r}$ from the stream of requests in a periodic fashion, such as once a day or once a week, or in an online fashion if access patterns and resource characteristics change frequently. Previous work has also suggested and evaluated the use of pairwise dependencies to guide prefetching decisions [14, 15] . We extend this work by presenting efficient techniques for constructing and thinning probability-based volumes.

The server computes probabilities from counters $c_{s|r}$ for pairs of resources that occur together, as well as counters c_s for occurrences of individual resources, where $p_{s|r} = c_{s|r}/c_r$. An efficient algorithm can compute these counters across the requests by each source.

Two tunable parameter are

- the *threshold probability* p_t
- the *time interval* T.

The implication probabilities are estimated by evaluating the stream of requests the server gets in a time period. We assume our input data is all from one server and is a list of triples of the form: (p_i, r_j, t_k) meaning that proxy p_i asked for resource r_j at time t_k.

The counting done below is performed for one value of the parameter T, but can be easily modified to work for several values.

We compute the following quantities:

[1]The volume relationship is not symmetric – s can belong to r's volume, though the reverse may not be true.

- For each pair of resources (r_i, r_j), c_{ij} is the number of request for r_i that were followed by a request for r_j (from same source) within time T.
- For each r_i, n_i is the number of requests for r_i.
- For each r_j, m_j is the number of requests that were followed by a request for r_j, from same source, within T time units.
- S is the total number of requests (number of triples in the input).

Input format We separate the input by proxies. For each proxy, we sort tuples by time. The data now has the form:
$(s_1, r_{11}, t_{11}), (s_1, r_{12}, t_{12}), \ldots$
$\diamond$
$(s_2, r_{21}, t_{21}), (s_2, r_{22}, t_{22}), \ldots$
$\diamond$
$(s_3, r_{31}, t_{31}), (s_3, r_{32}, t_{32}), \ldots$
$\diamond \cdots$

Computing the counts: The estimation of the probabilities is sketched in Figure 1. The output of the algorithm is as follows. For all resources r_i that were accessed enough times ($n_i > L$, for some threshold L), we ouput all resources that i implies with estimated probability at least p_t. For all resources r_j we output $\hat{p}_j$, the estimated probability that a request for any resource from our server other than r_j, is followed by a request for r_j within T time units.

Computing probability implications for a set of n resources can potentially require n^2 counters, though many pairs of resources do not typically occur together. To reduce the memory requirements, the algorithm performs random sampling to decide if and when to create a counter, and when to simply ignore a pair of resources. For example, suppose a resource r occurs less than T seconds before resources s. If the counter $c_{s|r}$ does not exist, we create a counter with probability inversely proportional to the frequency of accesses to r and the threshold probability p_t. Pairs that often occur together are likely to have a counter $c_{s|r}$ and an estimate of $p_{s|r}$, without needlessly generating counters for pairs of resources with low implication probabilities.

Further reductions in processing and memory overhead are possible by *combining* directory-based volumes with estimation of probability implications. For example, the probability implication algorithm could generate counters $c_{s|r}$ and estimate probabilities $p_{s|r}$ only for resources s and r with the same directory prefix. Using directory prefixes to reduce the number of counters has the potential to avoid locating pairs that inadvertently occur together because both resources are popular or are accessed during the same time interval by two different clients of the same proxy. To improve the accuracy and reduce the size of probability-based volumes, the volumes can be trimmed by focusing on *effective* predictions. After estimating the implication probabilities $p_{s|r}$, the server can evaluate the potential effectiveness of piggyback information by measuring how often a piggyback message generates a *new* prediction for resource s. This is particularly important when an access to s is often preceded by a *sequence* of requests by the same proxy, as would occur in downloading a Web page with multiple embedded images. We evaluate the effectiveness of these techniques for reducing the volume size in Section 4.3.

Initialize:
  For all resources r_j, set $n_j \leftarrow 0$ and $m_j \leftarrow 0$
  For all pairs of resources set $c_{ij} \leftarrow 0$
  Set S to be the total number of requests.

Loop over sources $s = s_1, s_2, \ldots$:
  Initialize per source:
    $\forall i,\ l_i \leftarrow -\infty$ (time of last request for resource r_i by s).
    Create an empty FIFO queue (all requests in the last T time units.)

  loop: For each request (s, r_j, t'):
    Increment n_j
    For all items r_i in the queue with time stamp
      later than $\max\{l_j, t' - T\}$, increment c_{ij}.
    Add to m_j the number of items on the queue with time stamp
      later than $\max\{l_j, t' - T\}$ and different than r_j.
    Set $l_j \leftarrow t'$.
    Discard from the bottom of the queue all items
      with time stamp earlier than $t' - T$.

Output:
  Compute estimates:
    $\hat{p}_{j|i} = c_{ij}/n_i$
    $\hat{p}_j = m_j/(S - n_j)$
  For all resources r_i such that $n_i > L$,
    for all j such that $\hat{p}_{j|i} > p_t$,
      output $(j, \hat{p}_{j|i}, c_{ij})$.
  For all resources r_j, output $\hat{p}_j$

Figure 1: Estimating implication probabilities

Client Logs	**Time Period**	**Requests (millions)**	**Distinct Servers**	**Unique Resources**
Digital	Sep 16–22, 1996	6.41	57832	2083491
AT&T	Nov 8–25, 1996	1.11	18005	521330

Table 1: Client logs

4 Performance Evaluation

The effectiveness of our piggyback protocol hinges on the ability to create and customize volume information under a reasonable amount of piggyback traffic. In the absence of end-to-end Web traces, we evaluate a collection of large client and server logs to quantify the performance and overheads of the volume-construction, filtering, and piggybacking policies. After describing the trace data, we focus on directory-based and probability-based volumes and how they can be used in a variety of applications.

4.1 Client and Server Logs

The set of available client, proxy, and server traces do not permit a complete end-to-end evaluation of our proposed protocol. Ideally, we would like a collection of proxy traces and server logs covering the same time period with a significant amount of traffic between the proxy/server pairs. In the absence of such data, we post-process the server logs to construct pseudo-proxy traces by extracting the source IP address for each request. However, the pseudo-trace is inherently incomplete since the server logs do not include any client requests that were satisfied at the proxy cache (if any), or the proxy's requests to other servers. In addition, the server logs do not include the Last-Modified times of the resources. Thus, it is impossible to accurately simulate a proxy cache from the server logs alone. Likewise, client logs lack information about other resources at the server sites or accesses from other sources, preventing us from determining which server resources would be included in piggyback response messages.

Given these practical limitations, we did not attempt to simulate the traffic mixture at an individual proxy cache. Instead, we focused on the cost-performance metrics that could be gathered from each of our client and server logs. For example, the client logs include the pathname for each access, allowing us to estimate the effectiveness of directory-based volumes across requests to a wide range of servers. However, the client logs did not permit us to quantify the *size* of the piggyback messages, since we did not know the number of resources in each directory at the server sites. As a result, we use the client logs to evaluate the performance of directory-based volumes. The server logs permit us to evaluate both the cost and performance of the volume-construction heuristics. Based on the pseudo-proxy traces extracted from the server logs, we evaluate directory-based and probability-based volumes using the performance metrics discussed in Section 3 and estimate the usefulness of the piggyback information for various proxy policies.

We used client logs from AT&T and Digital [9] and server logs from Amnesty International USA, Marimba Inc., Apache Group, and Sun Microsystems. Table 1 summarizes the key information about the client logs, which have been described in detail in previous work [19, 20] . 15.80% and 18.7% of the requests resulted in Not Modified responses while validating cached resources, for the Digital and AT&T logs, respectively. The average response size was 12279 bytes in Digital log and 8822 in AT&T. In the Digital log, the top 1% of the servers were responsible for over 59% of the resources accessed, and 3.4% of the servers accounted for over half the unique resources accessed. In the AT&T log the top 1% of the servers were responsible for over 55% of the resources accessed, and 5.6% of the servers accounted for over half the unique resources accessed [9] . For resources that were accessed at least twice, about 15% of the responses in the AT&T log reflected that a response had changed [21]

Log (days)	Number of Requests	Number of Clients	Requests per Source	Unique Resources Considered
AIUSA (28)	180,324	7,627	23.64	1,102
Marimba (21)	222,393	24,103	9.23	94
Apache (49)	2,916,549	271,687	10.73	788
Sun (9)	13,037,895	218,518	59.66	29,436

Table 2: Server log characteristics

. Since response messages do not always include a Last-Modified time, and since resources may be modified without a size change, this estimate is necessarily conservative. In addition, it is impossible to determine if a resource changed multiple times between accesses.

The server logs represent a range of Web sites in terms of the number of resources and accesses, as shown in Table 2. Across the server logs, a very small percentage of clients were responsible for a majority of the accesses (often 10% of clients were responsible for over 50% of all accesses). Most of the requests were also for a small number of resources (around 85% of the requests were for less than 10% of the unique resources). These trends are consistent with studies of other Web server logs [22] . The Marimba server log was obtained from a site that served small amounts of data with practically all requests using the POST method rather than GET. This gives the Marimba logs a different traffic mix than more traditional server logs. We deleted apparent uncachables (resources with the string "cgi" or query URLs with the "?" character), ensured time entries were within the log dates range, and combined identical resources (e.g., `http://www.foo.com/` and `http://www.foo.com`). To avoid piggybacking information about unpopular resources, we only considered resources that were accessed at least ten times; these resources account for $98 - 99\%$ of requests.

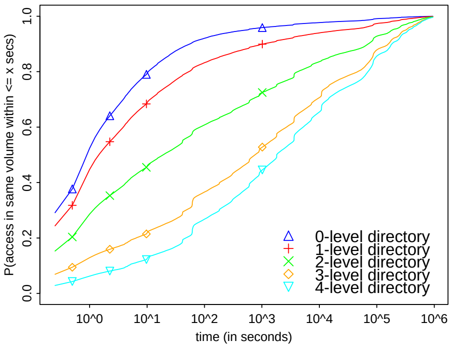
4.2 Directory-Based Volumes

Figure 2 quantifies the degree of reference locality at several levels of directory prefixes, using the AT&T proxy trace. The statistics in Figure 2(a) report the proportion of client requests that have a directory prefix that occurred earlier in the trace, as well as the average time between successive accesses. For example, 98.5% of requests access a server (level-0 directory) that has been accessed before, perhaps by a different client; on average, such accesses are just over an hour apart. Despite the large average spacing, many accesses occur in a small period of time, as shown by the cumulative distribution in Figure 2(b). In fact, many accesses are less than ten seconds apart, even within volumes based on the first and second levels of the directory structure. A server with directory-based volumes could predict and/or refresh these resources by piggybacking information on the earlier accesses.

Some of these accesses can be predicted easily, since they stem from embedded images that are referenced within a Web page. These embedded images are typically requested within a few seconds of the enclosing page, unless the client disables the transfer of images. Since the AT&T client log includes the full content of the resources, we were able to determine which URLs correspond to embedded references. Even with these references removed, the trace still exhibits significant temporal locality; the average interarrival times increase by just 10% and the probability distributions retain their shape. With or without embedded images, the probability distributions show that over 55% of accesses occur less than fifty seconds after another request in the same 2-level volume. A succession of accesses on this time scale are likely to be captured in a *single* piggyback message by using the RPV (recently piggybacked volume) list, discussed in Section 2.2. More than 82% of requests would follow a piggyback message that occurred within the previous two hours.

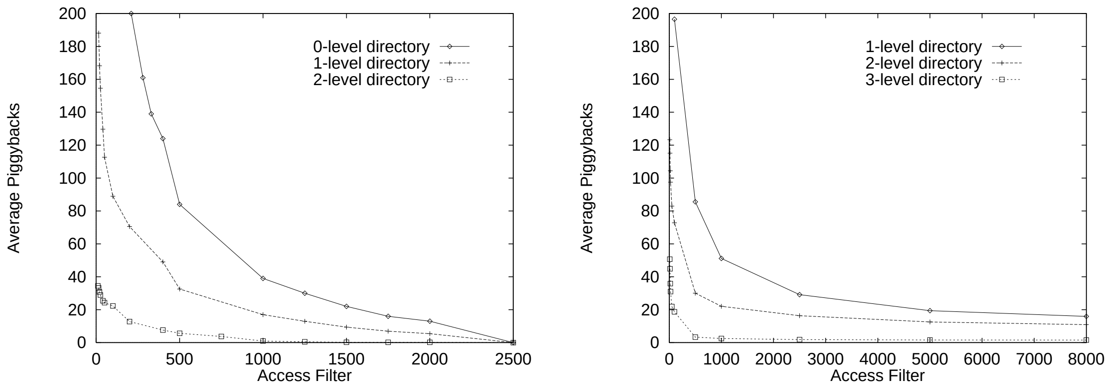
Directory Level	% Seen Before	Average Interarrival
0	98.5%	3637 sec
1	91.8%	10976 sec
2	78.0%	29919 sec
3	66.3%	50301 sec
4	61.6%	59526 sec

(a) Directory prefix statistics



(b) Distribution of interarrival times

Figure 2: Spacing of requests within directory-based volumes for AT&T proxy trace



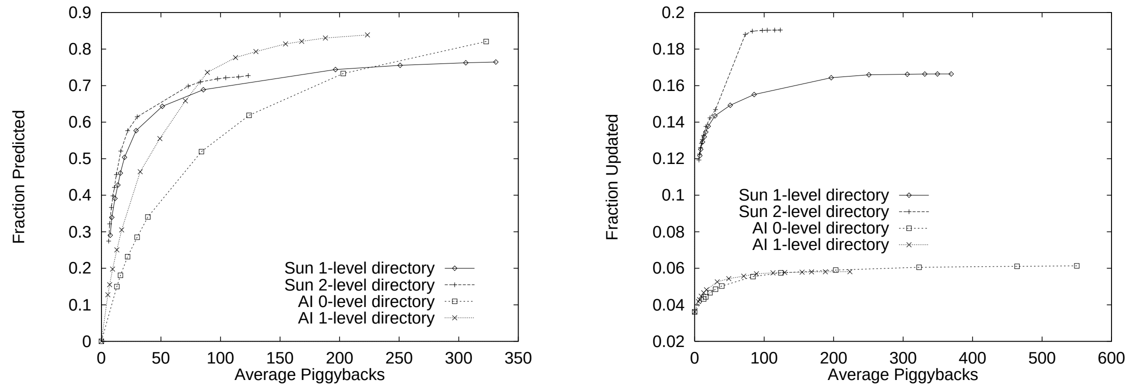
(a) AIUSA

(b) Sun

Figure 3: Average piggyback size vs. access filter for directory-based volumes

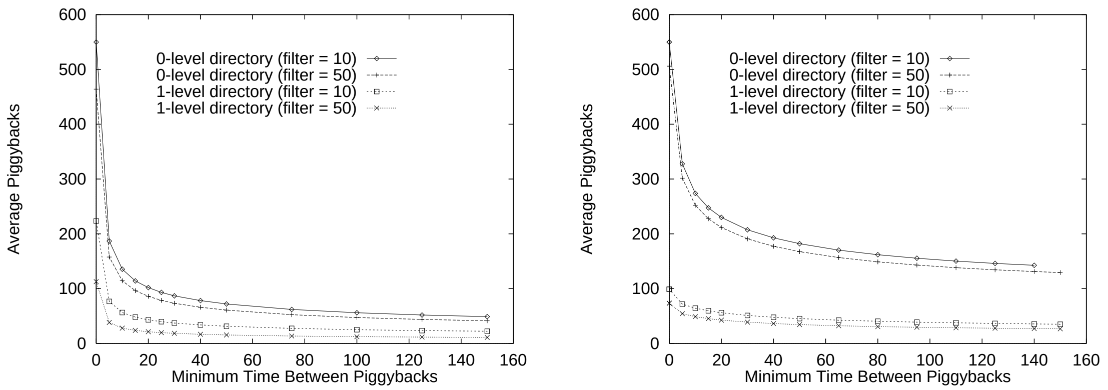
The experiments with the proxy trace suggest that directory-based volumes are effective in projecting future access. However, we cannot use the proxy trace to determine how much excess information the server would send in piggyback messages. To quantify these cost-performance trade-offs, we measured the piggyback overhead and predictive power of directory-based volumes for the four server logs. Figure 3 plots the average size of piggyback messages for the Apache and Sun server logs across a range of filters, where a filter of 100 indicates that piggyback messages do not include resources that are accessed less than 100 times in the trace. For efficient post-processing of the server logs, we imposed a maximum piggyback size and graphed the region with an average piggyback size of less than 200. The experiment evaluates three different levels of directory-based volumes; we do not evaluate a 0-level directory for the Sun log, since this would result in a single 29436-element volume.

The piggyback size drops dramatically when volumes are constructed based on longer prefixes in the resource pathnames. In addition, the piggyback size decreases dramatically as a function of the filter; even for 1-level volumes in the Sun logs, the average size is less than 20 elements when the piggyback messages omit information about resources with fewer than 5000 accesses. Fortunately, this aggressive filtering does not substantially reduce the effectiveness of the piggyback messages, as shown in Figure 4(a), which plots the proportion of accesses by a proxy that were predicted in a piggyback message to the same proxy within the last five minutes. The 1- and 2-level Sun volumes can predict approximately 60% of the future accesses with an average piggyback size of just 30



(a) Fraction predicted vs. average piggyback size (b) Fraction updated vs. average piggyback size

Figure 4: Accuracy of directory-based volumes for Sun and AIUSA logs



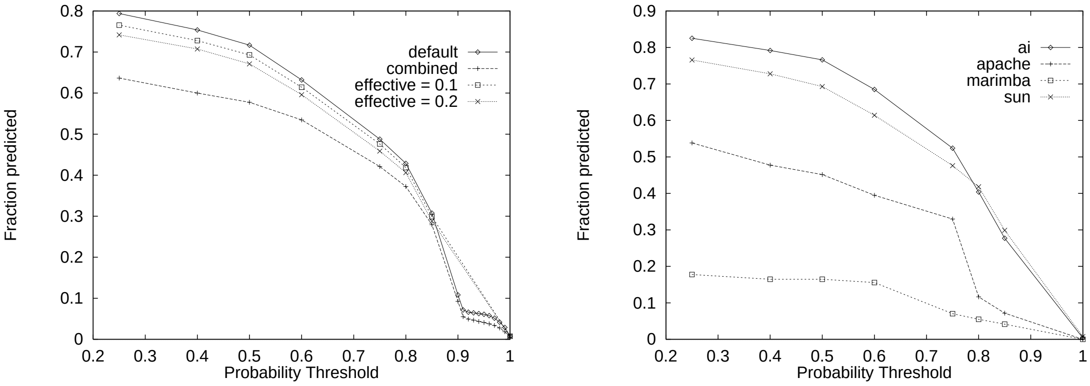
(a) AIUSA (b) Apache

Figure 5: Average piggybacks vs. minimum time between piggybacks

elements, with the 2-level volumes achieving slightly better predictions. Larger piggyback messages offer diminishing returns. Experiments with the Apache (not shown) and AIUSA logs show similar trends, though these logs have higher peak prediction rates of 80% and have lower piggyback sizes, due to the smaller number of resources at these sites. In contrast, the Marimba logs have very low prediction probabilities, consistently below 30%.

Although the piggybacking messages can predict 70–80% of the accesses that will occur in the next five minutes, some of these future requests will access resources that are not in the proxy cache. Figure 4(b) plots the proportion of accesses that are predicted by a piggyback message (within five minutes) *and* have appeared in a previous request in the last two hours. This metric estimates what fraction of client requests have been updated recently at the proxy cache, to either freshen or invalidate the resource. For the Sun logs with 2-level volumes, nearly 20% of requests were updated in the last five minutes; the update fraction increases to just over 20% if the experiment considers a 15-minute time interval instead of a 5-minute interval. For the Apache (not shown) and AIUSA logs, the update fraction is consistently in the range of 5–10%. Given that cache hit rates are typically around 30-50%, these results suggest that the piggybacking protocol enables a significant portion of the cache hits to access a fresh version of the resource (without requiring an If-Modified-Since request to the server).

The combination of access filters and directory-based volumes is effective in achieving a high prediction fraction with a modest piggyback size. For further reductions in protocol overhead, the



(a) Sun logs (various techniques) (b) All logs (effective threshold 0.1)

Figure 6: Fraction predicted vs. probability threshold for probability-based volumes

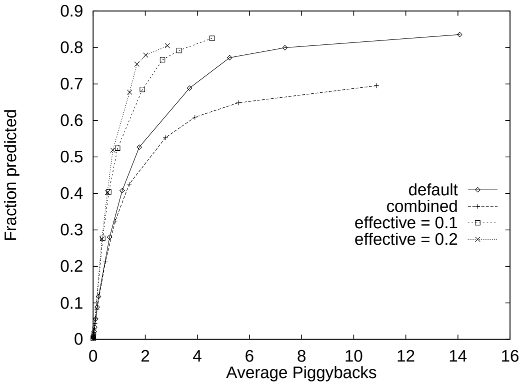
proxy can tune the frequency of piggyback message by maintaining a list of recently piggybacked volumes (RPV). The graphs in Figure 5 plot the average piggyback size as a function of the minimum time between successive piggyback messages, for 0- and 1-level volumes and two different access filters (10 and 50). The RPV list is extremely effective in reducing the amount of piggyback traffic with no significant loss in the fraction of resources that are predicted. A 30-second minimum time between piggyback messages achieves most of the necessary reduction; this permits the proxy to impose a tight limit on the amount of information stored in the RPV lists. Thinning the piggyback traffic, through both access filters and RPV lists, permits the server to construct larger volumes, with larger prediction fractions, without sending an excessive amount of piggyback traffic.

4.3 Probability-Based Volumes

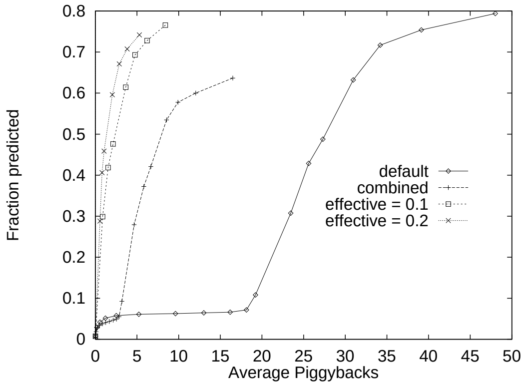
Directory-based volumes can achieve high prediction rates, at the expense of sending excess piggyback information. In contrast, estimating pairwise probabilities allows the server to construct a very accurate volume for each resource, at the expense of additional computational complexity and the possibility that some popular resources occur in multiple volumes. Across all four server logs, probability-based volumes rarely had symmetric volume assignments, whereby a pair of resources occur in each other's volumes. For a time interval of $T = 300$ seconds and a probability threshold of $p_t = 0.2$, only 1% of resources belong to their own volumes, and only 3%–18% of volume contents are symmetric. In addition, most resources occur in a small number of volumes and most volumes have a small number of resources. These statistics support the use of probability-based volumes as an efficient alternative to partitioning resources based on their directory prefixes.

To evaluate the predictive power of probability-based volumes, Figure 6(a) plots the fraction of accesses that were predicted by a piggyback message to the same proxy in the last five minutes ($T = 300$ seconds). The top curve corresponds to the base case which computes probabilities $p_{s|r}$ for each pair of resources, and includes in volumes all pairs where $p_{s|r} > p_t$. The next two curves in Figure 6(a) plot the prediction rate after removing implications with *effective* probability below 0.1 and 0.2, respectively. Removing these implications does not have a significant impact on the prediction rate. The bottom curve in Figure 6(a) corresponds to *combined* volumes that remove implications for resource pairs that do not have the same 1-level directory prefix. For very small threshold probabilities, these combined volumes are virtually identical to the 1-level directory-based volumes described in Section 4.2.

Each of the logs has resource pairs with a range of implication probabilities, as shown in Figure 6(b), though the Marimba log has relatively low prediction rates, as discussed in Section 4.2.



(a) AIUSA



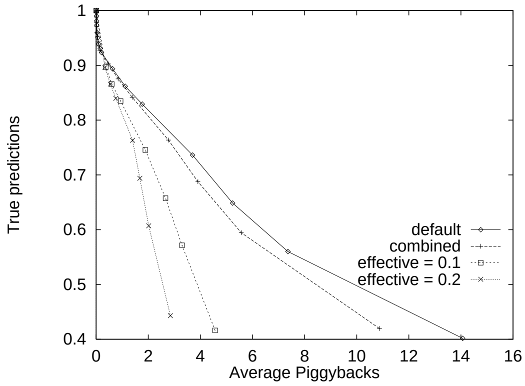
(b) Sun

Figure 7: Fraction predicted vs. average piggyback size for probability-based volumes

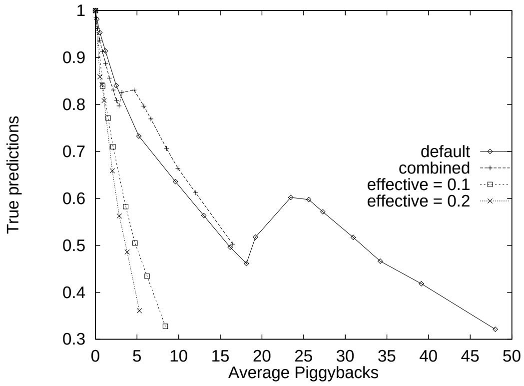
High probabilities often stem from embedded images and popular HREF links in a Web page. High prediction rates can always be achieved using very large volumes. Large volumes, however, incur large average piggyback sizes and a high rate of false predictions. Figure 7 plots the prediction rate as a function of the average piggyback size, computed across a range of probability threshold values, for the AIUSA and Sun logs. As expected, the prediction rate grows with the piggyback size, with diminishing returns for larger piggyback messages. Compared to the experiments for directory-based volumes in Figure 4(a), the probability-based volumes achieve a high prediction rate with a lower piggyback size. Thinning the probability-based volumes by removing ineffective implications or pairs in different 1-level directories offers a significant reduction in the piggyback size, particularly for the Sun logs, as shown in Figure 7(b). The amount of redundancy present in piggybacks can be gauged by the ratio between average piggyback size and the average number of distinct predictions (per request). This ratio was only a factor of 2–4 for volumes with effective probability above 0.2.

By focusing on effective predictions, the server can significantly reduce the size of piggyback messages without reducing the prediction rate. This substantially increases the fraction of true predictions, as shown in Figure 8 for both the AIUSA and Sun logs. In general, for well-constructed volumes, the ratio of true predictions should increase under smaller piggyback sizes, since we expect higher values of p_t to yield more accurate predictions. As evidenced by the non-monotonic curve in Figure 8(b), however, this is not always the case. This nonmonotonicity stems from many resource pairs with high implication probability and low effective probability. Such elements contribute to the piggyback size without increasing the number of true predictions. They rarely generate new *true* predictions, though they sometimes generate *false* predictions. Such pairs occur from repeated sequences of requests for resources in the same 5-minute interval, due to popular HREF links and downloading of embedded images. Removing these pairs not only reduces the average size of piggyback messages, but also has desirable monotonic dependence where smaller piggyback sizes yield more accurate predictions. Although experiments with the AIUSA and Apache logs show these effects, the results are most dramatic for the larger and more popular Sun site.

Applications where false predictions incur large costs (such as prefetching and cache replacement) must strike a careful balance between *precision* (the prediction fraction in Figure 7) and *recall* (the true predictions in Figure 8). Removal of ineffective implications resulted in better tradeoffs between these two metrics, and considerable improvement was obtained for the Sun logs. Figure 9 summarizes the relationship between these two metrics for volumes constructed with an effective probability threshold of 0.2, which consistently produced the best volumes for a given piggyback size. The graph does not plot the average piggyback size, which increases with the prediction fraction on the x-axis; since effective implications result in such small piggyback sizes, this overhead is less of a factor compared to the cost of making incorrect prefetching and cache replacement decisions.



(a) AIUSA



(b) Sun

Figure 8: True prediction vs. average piggyback size for probability-based volumes

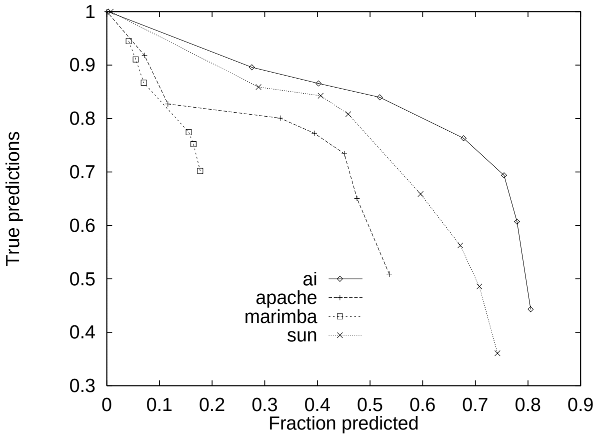


Figure 9: True prediction vs. fraction predicted for effective threshold 0.2 for all logs

By comparison, combined volumes exhibited worse tradeoffs, and the directory-based volumes (not shown), consistently generate a much larger number of false predictions, often in the range of 70–90%, even with the aid of filtering techniques.

4.4 Estimating Cache Performance

Since our traces do not permit a complete cache simulation, we estimate the impact of our piggybacking scheme on cache performance through the fraction updated metric. We also consider a simplified replacement model where resources are removed from the proxy cache if they are not accessed (or appear in a piggyback message) within a certain time.

The update fraction metric captures the effectiveness of using piggybacked volumes to update the cache contents. The second column of Table 3 shows the fraction of requests that were preceded by a request for the same resource by the same proxy within the last $C = 2$ hours. This can be thought of as "cache hits". The third column is the fraction of requests that were preceded by a request for the same resources by the same proxy within the last $T = 5$ minutes (cache contains a fresh copy without exchanging piggybacks). The fourth column is the fraction of requests made to resources that were accessed within the last two hours but more than five minutes ago and updated by a pigggyback in the last five minutes (i.e., sending volumes resulted in *new* updates to cached

Server Log	prev. occ < 2hr.	prev. occ < 5min	updated by piggybacks and 5min < prev. occ. < 2hr.	average piggyback
AIUSA	6.5%	3.6% (55%)	2.0% (37%)	2.9
Apache	11.5%	5.4% (47%)	2.2% (22%)	1.6
Sun	23.7%	9.6% (41%)	11.0% (46%)	5.0

Table 3: Update fraction for probability-based volumes

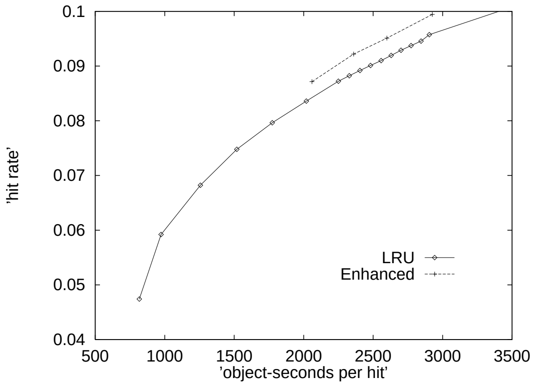


Figure 10: Cache hits vs. cost per hit for two replacement policies for Sun logs

resources). The parenthetical figure shown in columns 3 and 4 is the fraction of the "cache hits". The last column is the average piggyback size. The numbers reported in the fourth and fifth columns are for volumes with probability threshold $p_t = 0.25$, effective probability 0.2, and time interval $T = 300$. The fraction updated metric is the sum of the third and fourth columns; for example, the Sun logs have an update rate of 20.6% (9.6%+11.0%) with an average piggyback size of 5. As seen from the plots in Figure 4(b), directory-based volumes exhibit considerably worse performance for comparable piggyback sizes. Large average piggyback sizes (50-100) allow for higher fraction of updates, but still, the fraction updated peaks lower than for probability-based volumes. This inherent gap stems from requests that are not preceded by a recent request for a resource with same 1- or 2-level directory prefix.

Piggybacks can be used to enhance cache replacement decisions by increasing the priority of predicted resources. We evaluate the benefit of piggybacks by considering the tradeoffs between the number of cache hits and the "cost" of each cache hit. We measure the cost of a caching session by the total number of object-seconds spent (that is, the sum over resources of the number of seconds each resource spent in the cache). This is an appealing measure since it is independent of the exchanges of any given proxy with other servers. For each of the server logs, we evaluated the average cost per hit versus the total number of cache hits, averaged among all participating clients. Figure 10 plots these tradeoffs for two cache replacement policies. The first replacement policy utilizes only the last request time for an object to predict the next one. Hence, all resources remain in the cache for the same, fixed, amount of time. We plot the tradeoffs cost per hit versus the number of hits obtained by varying the time period from 40 seconds to 10 minutes. The second, piggyback-enhanced, replacement policy utilizes volumes obtained with time interval $T = 300$, probability threshold $p_t = 0.25$ and effective probability 0.2. Under this policy, all resources are first scheduled to remain in the cache for $t = 120 \ldots 300$ seconds. However, if a resource appears in a piggyback while in the cache, its "removal time" is updated to be 300 seconds beyond the current time. The enhanced policy that utilizes piggybacks exhibits 3-15% improvement in hit rate for the same cost.

Ultimately, the appropriate volume construction and filtering techniques depend on how the proxies use the piggybacked information and how much computational load can be borne by the server. Probability-based volumes, thinned to include only effective implications or resources with the same directory prefix, can be used to guide prefetching decisions at the proxy site. Although probability-based volumes introduce additional computational load at the server (or the transparent volume center), volume construction does not have to occur in an online fashion; in fact, in our experiments, we applied a single set of volumes for the duration of each log. Directory-based volumes are easier to create, though they generate larger piggyback messages. Filtering recently-piggybacked-volumes substantially reduces the size of the piggyback messages, without decreasing the fraction of accesses that are predicted in advance. The directory-based volumes are well-suited to applications like cache coherency and, to a lesser extent, cache replacement, where extra piggyback information does not incur a significant cost at the proxy.

5 Web Proxy Applications

The information in server piggyback messages can be used to improve the effectiveness of a variety of proxy policies, with different cost-performance trade-offs:

Cache coherency: Using the Last-Modified time information in the piggyback message, the proxy can remove stale items from the cache and freshen valid entries [9,19] . This lowers the likelihood of returning out-of-date resources to clients, and avoids the latency and TCP overheads of generating If-Modified-Since requests to the server on future client requests. Instead of simply removing stale resources from the cache, the proxy could construct an updated version by requesting that the server transmit the *difference* between the old and new versions; this proposed enhancement to HTTP [20] should be very effective in reducing the amount of data transfer, since most changes are small, relative to the size of the resource [20] . The update fraction metric allows us to estimate the impact of volumes. About 40%–50% of requests to cached objects are made to resources previously requested within 5 minutes. Assuming the cache has fresh copies of these resources, our best volumes (Section 4.3) enabled a priori refreshment for additional 22%–46% of requests made to cached resources, using average piggyback sizes of only 1–5. reading the plot and only the ai and sun logs volumes updated of 50-100.

Prefetching: The piggyback message can be used by the proxy to prefetch resources [15] , or by the server to automatically send resources that are likely to be accessed soon. In contrast to the cache-coherency application, piggyback messages that do not accurately predict future accesses incur a significant cost to the proxy, which may wrongly consume network bandwidth and cache space to fetch resources that are never requested. The proxy may decide not to prefetch items that have a recent Last-Modified time, since these resources may change again before they are accessed by a client. In addition, to avoid the cost of fetching and storing large resources, the proxy's filter can inform the server to generate a piggyback list that omits resources that exceed a target size. Even if the proxy does not prefetch the items in the piggyback message, the proxy and the server can both decide to maintain an open TCP connection if the piggyback information suggests that more proxy requests are likely to occur in the near future. The fraction predicted metric indicates how many requests are predicted by piggybacks and hence, can be prefetched. The true predictions metric measures the overhead of futile per useful prefetches. Our most accurate volumes (see Figure 9) exhibited the following tradeoffs. For the Apache log, for example, 40% of accesses can be prefetched with 20% futile fetches (10% total increase in bandwidth) or 55% of accesses can be prefetched with 50% futile fetches (27% increase in bandwidth). On the Sun logs, prefetching 30% of requests incurs 15% futile prefetches (5% bandwidth increase) and 70% prefetching incurs 50% futile prefetches (35% increase in bandwidth).

Cache replacement: The server piggyback information can help guide cache replacement decisions. Rather than the typical removal of the least-recently-used item, the proxy could continue to cache items that have appeared in recent piggyback messages, as long as the resource has not been modified at the server. More generally, the proxy could combine piggyback information with other "cost" metrics that impact the cache replacement decision, including resource size, content, or frequency of modification. The effectiveness of the cache replacement strategy depends on the accuracy of the piggyback information, though a suboptimal cache replacement decision may not be as detrimental as an incorrect prefetching decision. Promising evidence for the gain from such enhanced cache replacement decisions is the fraction updated which shows that volumes produce a considerable number of predictions for requests for cached resources, and also the tradeoffs between the prediction rate and the fraction of true predictions, which assert that these predictions are accurate.

Adaptive freshness interval: The proxy can use the piggyback message to tune its caching policies, even when the piggybacked items do not reside in the cache. Since the piggyback includes the Last-Modified time of each resource, the proxy can estimate and record how often the resource changes, or query the server explicitly for this information. If a client requests a resource not in the proxy's cache, the proxy can use the rate-of-change information to decide if it should cache it or select an appropriate freshness interval (Δ) for that resource (or for all the resources in its volume). This permits the proxy to balance the cost of resource validation with the risk of sending stale information to a client.

Informed fetching: The piggybacks contains meta attributes of resources that are likely to be requested soon. These attributes can be kept and used to prioritize the fetching queue when users do issue the requests (the scheduling is performed prior to contacting the servers). For example, shorter files can be fetched first (when there is not enough bandwidth for all outstanding requests). If bandwidth between proxy and server(s) is congested, such a scheduling strategy decreases average per-user latency (users requesting small files do not have to wait long and users with large requests wait a bit longer.) Single users utilizing a low-bandwidth connections would first view text and small images when requesting a page with embedded images. The relevant metric is the fraction predicted. Our best volumes (Section 4.3) inform the client with meta-attributes prior to issuing 55-80% of requests, while using very small average piggyback sizes.

6 Protocol Deployment

Our piggybacking protocol can be incorporated into HTTP 1.1 [23] . The performance improvements can be attained by deploying our changes at the popular proxies and servers, since a significant percentage of Web traffic stems from a small portion of proxies and servers (as discussed in Section 4.1). We present an example exchange and quantify the actual overheads. Then, we describe how to reduce server load, and avoid modification to existing server sites, by performing volume maintenance and piggyback generation inside the network.

6.1 Piggybacking in HTTP 1.1

A proxy can specify a filter as an extra field in the HTTP request header. A cooperating server parses the filter and sends relevant piggyback information appended to the response. Although the response could easily be piggybacked in a new response header, we avoid delaying the actual body of the response while the piggyback is being constructed. HTTP 1.1 permits *chunked transfer-coding* whereby additional information may be sent in the *trailer* of the response. The proxy uses a `TE` request header field indicating its willingness to accept such a chunked coding. A proxy's GET or HEAD request would thus include `TE: chunked` and `Piggy-filter` request headers, with the latter listing the filter attributes. A GET request enhanced by a proxy filter might like:

```
GET /foo.html HTTP/1.1
host: bar.com
TE: chunked
Piggy-filter: maxpiggy = 10; rpv = 3, 4;
```

The proxy specifies a filter with a `maxpiggy` of 10 indicating that no more than ten elements should be piggybacked. The `rpv` field identifies the most recently piggybacked volumes obviating the need for the server to send information about those volumes again. Alternately, the proxy could have specified a different filter: a probability threshold `pt` (say, 0.7) requiring piggybacked volume elements to occur together with the requested resource with a probability greater or equal to this threshold.

The piggybacked information will appear in the `Piggyback-volume` response header field in the trailer of the response. The server must include a `Trailer` header field indicating the later appearance of the `Piggyback-volume` response header field. The response from the server at the site **bar.com** might look like:

```
HTTP/1.1 200 OK
Trailer: Piggyback-volume
Content-encoding: chunked
< Size-of-chunk >
      <data>
   ...
0
Piggyback-volume: {7, r4 r3 r1 r6};
CRLF
```

The server's chunked response ends with the mandatory zero-length chunk. The `Piggyback-volume` response header line includes a volume id and resources in the volume. Note that none of the resources are from volumes 3 or 4 and the number of elements piggybacked is less than the `maxpiggy`, in keeping with the proxy's filter requirement.

Each piggyback element consists of a URL, Last-Modified time, and a resource size. A URL length of 50 bytes, a 4-byte Last-Modified time, and a 1-byte size field result in an average of 55 bytes for each element. In the case of the probability threshold filter of 0.7, our best scheme would generate less than 6 piggybacks (as shown in Figure 7(b)), and thus add less than 350 bytes, including the volume identifier. This might often fit in the same packet as the response or at most require one new packet, and is small relative to the average resource size of around 10 kilobytes. Since every future TCP connection obviated saves at least two packets, our scheme does not increase the number of packets and incurs an extremely small increase in the bandwidth consumption. In exchange for the piggyback traffic, the proxy has prediction information for nearly 75% of the accesses in the next 5-minute interval. Similarly, directory-based volumes can predict 52% of requests with an average piggyback size of 16 (as shown in Figure 4(a)), resulting in less than 900 bytes, which is just over half the size of a 1500-byte packet.

6.2 Transparent Volume Centers

Despite the relatively simple strategies for maintaining resource volumes and applying proxy-generated filters, the piggybacking protocol consumes server resources. By maintaining volumes in (say) a router or gateway that connects one or more servers to the network, we can reduce server overhead. Although this "volume center" may have limited knowledge about recent changes to server resources, it can observe traffic between the proxies and the server, glean Last-Modified time information, and update volumes accordingly. Depending on where it is placed, a volume center could observe accesses across multiple replicas of a single Web site, or across related sites from the same institution. For transparent operation, the volume center should lie along all paths between the proxy and the server. Otherwise, some packets may travel directly to the server, even though other packets have been intercepted by the volume center. Placing the volume center at the router that connects to a web-hosting platform would ensure that all packets travel through the center en route to/from the servers.

In many cases, the volume center can obviate the need for direct communication between the proxy and server sites. For example, suppose a proxy transmits an If-Modified-Since GET request for resource r at the server. After intercepting the TCP establishment message and the GET request, the volume center can generate a Not Modified response if it has recent Last-Modified time information that implies that r has not changed. If the volume center cannot satisfy the GET request, because the resource has changed or the volume has out-of-date Last-Modified time information for r, the GET request is simply forwarded to the server for normal handling. The volume center awaits the server's reply before updating r's volume information and forwarding the reply to the proxy client. In addition, the volume center can transparently piggyback validation information to the reply message, whether the GET request is satisfied at the server or the volume center. To avoid piggybacking out-of-date Last-Modified times, the volume center can issue requests to the server to refresh the status of its resources.

We are investigating the cost-performance trade-offs of deploying volume centers without software modifications at the server sites. The volume center model has similarities to recent proxy architectures that integrate caching with routing, transparent to the client sites [24] , although the volume center maintains information about resources instead of the actual contents, and can generate piggyback information. If the volume center is integrated with a network router, the architecture should avoid imposing a performance penalty on the general router functions. This can be achieved by offloading the volume maintenance and piggyback operations to a separate processor. Based on packet headers, the router can forward request traffic to the volume server for consideration. On the reverse path from the server to the network, the router can immediately forward the response message to the proxy cache and forward a copy of the response header to the volume center. This permits the volume center to update its resource and volume information without delaying the reply to the proxy client.

7 Conclusions and Future Work

We have presented an end-to-end framework that examines the problem of network overload caused by the exponential increase in Web related traffic. Our holistic approach permits piggybacked exchange of the unused information present in the various Web components. We group resources into volumes to maximize information available at the server and use filters to tailor the piggybacked information to the various proxies. No per-proxy or per-server state needs to be maintained. Based on efficient data structures and algorithms, we presented an in-depth evaluation of various volume construction and thinning techniques on a collection of large server logs. Our proposed protocol changes can be used in a variety of applications such as prefetching, cache coherence, and validation. A way to deploy our protocol changes in HTTP 1.1 was examined along with the use of transparent volume centers to maintain volumes in routers to lower server load.

As future work, we are formalizing a filter language that can be the basis for a range of proxy specific tailoring of piggybacks. We are also developing a variety of additional volume thinning techniques. The deployment of transparent volume centers to obviate server modifications is also being examined in greater depth. At the application level, we plan to examine the actual impact of our piggybacks on several cache related issues such as replacement, validation, and coherency.

8 Acknowledgments

We would like to thank Sandy Irani, Jeff Mogul, Pawan Goyal for their detailed comments on draft version of this memorandum. We would also like to thank Dan Duchamp, Craig Wills for their suggestions.

Appendix

The performance measures allow us to define natural optimization problems of producing *optimal sets of volumes*. There are various optimization problems that result from bounding one measure such as average piggy size, prediction rate, or true prediction, while optimizing another.

> Given a server log (sequence of triples) and an upper bound on average piggy size, find a set of volumes which maximizes the prediction-rate.

Another problem is:

> Given a server log and a limit on the false-prediction rate, find a set of volumes which maximizes the prediction rate.

Lemma: The two optimization problems defined above are NP-complete.

Proof: We sketch a reduction from 3SAT to the first problem. A similar reduction works for the second problem. Given an instance of 3-SAT, construct a log as follows: Create a resource for each variable and its negation. Create another "special" resource R. We create many "subsequences" that are assumed to each be either by different clients or far enough in time from others so that predictions made in one subsequence will not affect items in others. Let M be the number of clauses and N the number of variables in the 3SAT instance. We use the same symbol for each variable and its corresponding resource. For each variable x and its negation $\overline{x}$ we create $3M$ subsequences of the form: "$x, \overline{x}$ at time t and R at time $t+1$". For each clause of the form $(x \vee y \vee z)$ we create a subsequence of the form "x, y, z at time t, R at time $t+1$". We restrict the average piggy size to be at most $3MN + 3M$. We claim that there exist a volumes set with prediction rate of $M(3N+1)/(9MN+4M)$ if and only if the 3SAT instance is satisfiable.

Given a volume set we produce a satisfying assignment as follows: For each variable x we set x to be true if and only if the volume of x is equal to $\{R\}$. Given a satisfying assignment, we can obtain a good volume set as follows. The volume of R is empty, for each true variable (or negation) the corresponding volume is $\{R\}$, and for each false variable the corresponding volume is empty. Note that only requests for R can be predicted. The prediction rate above is equal to the fraction of requests that are made to the resource R. Hence, any volume set with such prediction rate implies that all occurances are correctly predicted. Therefore, for each variable x, at least one of the volumes of x and the volume of $\overline{x}$ must predict R. The limit on average piggy imply that for each variable x, at most one of the two volumes of x and $\overline{x}$ is nonempty and can be of size at most 1. It follows that any good volume set is such that for each variable x, one of the two volumes of x and $\overline{x}$ is empty and the other volume equals $\{R\}$. For each clause $(x \vee y \vee z)$, the request for R in the subsequence which corresponds to this clause is predicted if and only if at least one of the volumes of $\{x, y, z\}$ equals $\{R\}$. ∎

References

[1] B. M. Duska, D. Marwood, and M. J. Feeley, "The measured access characteristics of World-Wide-Web client proxy caches," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), pp. 23–35, December 1997. `http://www.usenix.org/events/usits97`.

[2] "Squid internet object cache." `http://squid.nlanr.net/Squid`.

[3] A. Bestavros, R. L. Carter, and M. E. Crovella, "Application-level document caching in the Internet," in *Proceedings of the International Workshop on Services in Distributed and Networked Environments*, June 1995. `http://www.cs.bu.edu/faculty/best/res/papers/Home.html`.

[4] S. Williams, M. Abrams, C. R. Standbridge, G. Abdulla, and E. A. Fox, "Removal policies in network caches for World Wide Web documents," in *Proceedings of ACM SIGCOMM*, pp. 293–305, August 1996. `http://www.acm.org/sigcomm/sigcomm96/program.html`.

[5] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), pp. 193–206, December 1997. `http://www.usenix.org/events/usits97`.

[6] V. Cate, "Alex – A global filesystem," in *Proceedings of the USENIX File System Workshop*, pp. 1–12, USENIX Association, May 1992.

[7] A. Dingle and T. Partl, "Web cache coherence," in *Proceedings of the International World Wide Web Conference*, May 1996. `http://www5conf.inria.fr/fich_html/papers/P2/Overview.html`.

[8] S. D. Gribble and E. A. Brewer, "System design issues for Internet middleware services: Deductions from a large client trace," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), December 1997. `http://www.usenix.org/events/usits97`.

[9] B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the World Wide Web," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), pp. 1–12, December 1997. `http://www.usenix.org/events/usits97`.

[10] V. N. Padmanabhan and J. C. Mogul, "Improving HTTP latency," *Computer Networks and ISDN Systems*, vol. 28, pp. 25–35, December 1995.

[11] H. F. Nielsen, J. Gettys, A. Baird-Smith, E. Prud'hommeaux, H. W. Lie, and C. Lilley, "Network performance effects of HTTP/1.1, CSS1, and PNG," in *Proceedings of ACM SIGCOMM*, (Cannes, France), pp. 155–166, August 1997. `http://www.inria.fr/rodeo/sigcomm97/program.html`.

[12] S. Manley and M. Seltzer, "Web facts and fantasy," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), pp. 125–133, December 1997. `http://www.eecs.harvard.edu/~vino/web/sits.97.html`.

[13] J. C. Mogul, "Hinted caching in the Web," in *Proceedings of the ACM SIGOPS European Workshop*, 1996. `http://mosquitonet.stanford.edu/sigops96/papers/mogul.ps`.

[14] A. Bestavros, "Using speculation to reduce server load and service time on the WWW," in *Proceedings of the ACM International Conference on Information and Knowledge Management*, 1995.
`http://www.cs.bu.edu/faculty/best/res/papers/Home.html`.

[15] V. N. Padmanabhan and J. C. Mogul, "Using predictive prefetching to improve World Wide Web latency," *Computer Communication Review*, vol. 26, no. 3, pp. 22–36, 1996. `http://daedalus.cs.berkeley.edu/publications/ccr-july96.ps.gz`.

[16] T. M. Kroeger, D. D. E. Long, and J. C. Mogul, "Exploring the bounds of Web latency reduction from caching and prefetching," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), pp. 13–22, December 1997.
`http://www.usenix.org/events/usits97`.

[17] M. Crovella and P. Barford, "The network effects of prefetching." To appear in *Proceedings of IEEE INFOCOM*, April 1998.
`http://www.cs.bu.edu/faculty/crovella/papers.html`.

[18] C. Liu and P. Cao, "Maintaining strong cache consistency in the World Wide Web," in *Proceedings of the IEEE International Conference on Distributed Computing Systems*, pp. 326–334, May 1997.
`http://www.cs.wisc.edu/~cao/papers/icache.html`.

[19] B. Krishnamurthy and C. E. Wills, "Piggyback server invalidation for proxy cache coherency," in *Proceedings of the World Wide Web-7 Conference*, (Brisbane, Australia), April 1998.

[20] J. C. Mogul, F. Douglis, A. Feldmann, and B. Krishnamurthy, "Potential benefits of delta encoding and data compression for HTTP," in *Proceedings of ACM SIGCOMM*, (Cannes, France), pp. 181–194, August 1997.
`http://www.inria.fr/rodeo/sigcomm97/program.html`.

[21] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul, "Rate of change and other metrics: a live study of the World Wide Web," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), pp. 147–158, December 1997.
`http://www.usenix.org/events/usits97`.

[22] M. F. Arlitt and C. L. Williamson, "Internet Web servers: Workload characterization and implications," *IEEE/ACM Transactions on Networking*, vol. 5, pp. 631–644, October 1997.
`http://www.cs.usask.ca/homepages/faculty/carey/research.html`.

[23] R. Fielding, J. Gettys, J. C. Mogul, L. Masinter, P. Leach, H. Frystyk, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1 (draft-ietf-http-v11-spec-rev-01.txt)," November 1997.
`ftp://ftp.ietf.org/internet-drafts`.

[24] "Cisco cache engine."
`http://www.cisco.com/warp/public/751/cache`.