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

Edith Cohen, Balachander Krishnamurthy, and Jennifer Rexford

AT&T Labs–Research; 180 Park Avenue
Florham Park, NJ 07932 USA
{edith,bala,jrex}@research.att.com

Abstract

The rapid growth of the World Wide Web has caused serious performance degradation on the Internet. This paper offers an end-to-end approach to improving Web performance 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 a server, and a very small amount of transient 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 [1–3]. New cache allocation and replacement schemes improve cache performance by extending the conventional LRU policy to incorporate resource size and other notions of cost [4–6]. 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 may have 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 [7, 8]. 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 [9, 10]. 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 [11, 12]. For example, a persistent connection permits embedded images in an HTML document to be downloaded without new

TCP connections. To reduce the number of TCP connections, a proxy can allow multiple clients to share a single persistent connection to a server. Similarly, a client via a persistent connection to the proxy can access multiple servers [3]. 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 additional 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, 24]. Similarly, using access patterns to predict future requests, the server can speculatively disseminate resources to the proxy [14] or guide proxy prefetching decisions [15–18], 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 [19, 20] or validating a list of cached resources at the proxy [10]. In this paper, we focus on an *end-to-end* information exchange that involves both the proxy and the server, with an emphasis on determining what additional information should be exchanged, how this information can be maintained and exchanged efficiently, and how the proxy can 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. 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 1.1 protocol. Our mechanisms do not require new TCP connections or per-proxy state at the servers. A small amount of transient state is maintained on a per-server basis at the proxies. Our changes are thus scalable.

- *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 resources that are 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 [21], we introduce heuristics for thinning volumes to improve accuracy and reduce redundancy in the piggyback information.

- *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 propose that volume maintenance and piggyback generation be performed transparently at a router or gateway along the path between the proxy and server. This volume center can construct volumes, apply filters, and generate piggyback messages on behalf of several servers, allowing piggyback messages to include information about resources at multiple sites. Transparent volume center is explained in [21].

In this paper, we focus on one-level caching, though our techniques are applicable to the general case of hierarchical caching.

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 the server's ability to generate accurate predictions with minimal overhead. 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 set of client logs (from Digital Equipment Corporation and AT&T) and server logs (from Amnesty International USA [AIUSA], 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 constructs filters to customize the piggyback messages from the server. We also describe how piggybacking of filters and volume information can be incorporated in HTTP 1.1. Section 3 presents and evaluates several techniques for the server to construct volumes, based on the filesystem's directory structure and the likelihood that pairs of resources are accessed together. The performance evaluation focuses on 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 4. Section 5 concludes the paper with a discussion of future research directions. Appendix A describes our collection of client and server logs.

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. We then show how to piggyback filter and volume information in the context of HTTP 1.1.

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. The proxy also 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 content types (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 resource at the server, from the same volume as the requested resource. The proxy filter controls the number of piggybacked elements and the kind of resources included in them. The proxy stores the 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 a regular 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.

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 p is not in the cache, it could be prefetched. If p is in the cache and is fresh, its expiration time is updated; otherwise the stale copy is deleted and a fresh copy could be prefetched. The prefetching decision or the order of prefetching could be guided by the size attributes in the piggyback message.

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 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 any additional per-resource or per-volume state beyond the information that is typically in the cache. The simple frequency control techniques are thus particularly efficient for servers with a large number of volumes, as would occur with probability-based volumes discussed in Section 3.3.

Alternatively, when the number of volumes is small, the proxy can maintain transient information about recent piggyback communication from the servers. In particular, the proxy stores a list of recently piggybacked volumes (RPVs) for each server, or for a small subset of servers that are visited frequently. 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. Based on this information, the proxy could conceivably disable piggyback messages when it requests a resource that is in one of the volumes in the RPV list. However, this would require the proxy to store the volume identifier for each resource in the cache. More importantly, the proxy does not know the volume identifier for resources that are not yet in the cache, and the server may change volume memberships across time.

Instead, the proxy request message includes the RPV list as a filter, allowing the server to decide whether or not to piggyback volume information in the reply message. 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, for example, 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.

2.3 Piggybacking in HTTP 1.1

Piggybacking of filter and volume information can be incorporated into HTTP 1.1 [22]. A proxy can specify a filter as an extra field in the HTTP request message. 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 the `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 look like:

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

The proxy specifies a filter with a `maxpiggy` of 10 indicating that a maximum of ten elements to 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 p_t requiring piggybacked volume elements to occur together with the requested resource with a probability greater or equal to this threshold. The server response might look like:

```
HTTP/1.1 200 OK
Trailer: P-volume
Transfer-encoding: chunked
 < Size-of-chunk >
      <data>
   ...
 0
P-volume: vol=7; pe="u4,895527629,5465";
    pe="u3,891527021,1290";pe="u7,821993421,1290"
CRLF
```

The piggybacked information appears in the response header field in the trailer of the response. The server must include a `Trailer` header field indicating the later appearance of the `P-volume` response header field. The server's chunked response ends with the mandatory zero-length chunk. The `P-volume` response header line includes a volume id and resources in the volume. Note that the resources are not from volumes 3 or 4 and the number of elements piggybacked is less than the `maxpiggy` of 10, in keeping with the proxy's filter requirement.

A piggybacked message consists of a 2 byte volume identifier (allowing up to 32767 volumes per server) and a sequence of piggyback elements. Each piggyback element consists of a URL, Last-Modified time, and a resource size. In our logs the length of a typical URL is about 50 bytes, after omitting the redundant server name portion. The Last-Modified time and the resource size can be represented by 8 byte integers. This results in an average of 66 bytes for each piggyback element.

For example, using probability based volumes with the Sun logs, an average of 6 piggyback elements (other logs had even smaller piggyback sizes) were necessary to predict 75% of the future accesses in the next five-minute interval (as can be seen later in Figure 6(b)). This would result in an overall addition of 398 bytes for the entire piggybacked message, which might often fit in the same packet as the response or at most require one additional packet. The piggyback messsage is small relative to the mean response size of 13900 bytes (with a median of 1530 bytes). Note that filtering will ensure that not all messages include a piggyback response. Also, since every future TCP connection obviated saves at least two packets, our scheme should reduce the total number of packets.

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 generates a list of other resources that are likely to be requested in the near future. After formulating the volume construction problem, we describe and evaluate two effective heuristics based on the file system's directory structure and the probability that resources are accessed together. The performance evaluation draws on the client and server logs discussed in Appendix A.

3.1 Optimization Criteria

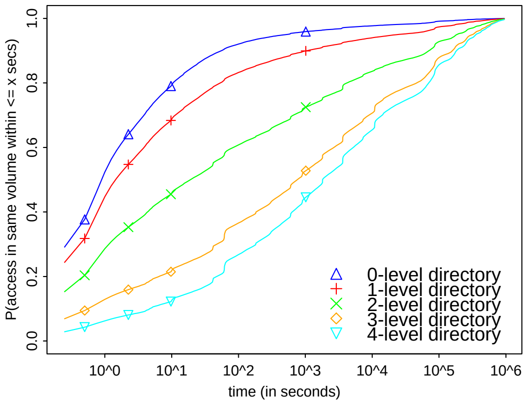
In evaluating volume construction techniques, we consider three key metrics that relate to how various applications would use the piggybacked information:

- *Fraction predicted:* 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 prediction,

Directory Level	% Seen Before	Median Interarrival
0	98.5%	0.9 sec
1	91.8%	1.5 sec
2	78.0%	19.7 sec
3	66.3%	766.2 sec
4	61.6%	1812.0 sec

(a) Directory prefix statistics



(b) Distribution of interarrival times

Figure 1: Spacing of requests within directory-based volumes for AT&T proxy trace

and is important for prefetching policies. Note that recall (prediction fraction) can be high without a corresponding increase in precision (true prediction fraction) and vice-versa.

- *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 update fraction subject to a constraint on either the average piggyback size or the fraction of true predictions. However, constructing optimal volumes is an NP-complete problem [21]. Instead, we present heuristics for grouping resources into volumes, and use these metrics as the basis of our performance evaluation.

3.2 Directory-Based Volumes

We first consider a simple static heuristic that groups resources based on the fields in the URL. Our evaluation of this heuristic draws on both the client and server logs, and considers filters based on access frequency and the list of recently piggybacked volumes.

3.2.1 Volume Construction and Maintenance

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 [20]. To reduce the volume size, the server can group resources with the same directory prefix in their pathnames, up to some number of levels. 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.

3.2.2 Performance Evaluation

Figure 1 shows the locality of reference at several levels of directory prefixes, using the AT&T proxy trace. The statistics in Figure 1(a) report the proportion of client requests that have a directory prefix that occurred earlier in the trace, as well as the median 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, with a median of 0.9 seconds. Many accesses occur in a small period of time, as shown by the cumulative distribution in Figure 1(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 within other Web pages. 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 median interarrival times increase by $10-20\%$ 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 for this trace. 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 to disable redundant piggyback messages, as discussed in Section 2.2. More than 82% of requests would follow a piggyback message that occurred within the previous two hours.

These experiments 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 2 plots the average piggyback size (the number of elements in a piggyback message) for the AIUSA 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 entire 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 number of elements in piggyback messages 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 3(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 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, with higher peak prediction rates of 80% and lower piggyback sizes, due to the smaller number of resources at these sites. In contrast, the Marimba logs have very low prediction probabilities (as discussed in the Appendix A).

Although the piggyback 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 3(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 fraction predicted with a modest piggyback size. For further reductions in protocol overhead, the proxy can tune the frequency of piggyback message by maintaining a list of recently piggybacked volumes (RPV). Figure 4(a) plots 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; i.e., limiting to resources that have been accessed at least this many times) for the Apache logs. 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, as shown in Figure 4(b). Experiments with the other server logs show the same trends. 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 fraction predicted, without sending an excessive amount of piggyback traffic.

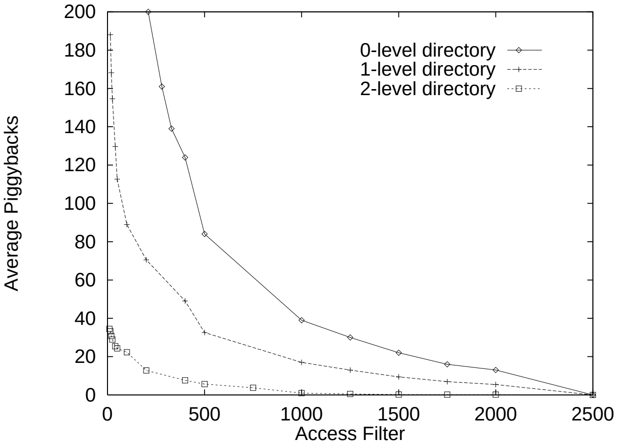
3.3 Probability-Based Volumes

The server can provide more accurate predictions by estimating the likelihood that resources are accessed together. After describing how these access probabilities are computed, we introduce heuristics to improve accuracy and reduce volume size by determining which predictions are most effective. Our evaluation of probability-based volumes draws on the server logs, and considers the effectiveness of a probability threshold for defining volume membership and various heuristics for removing ineffective predictions.

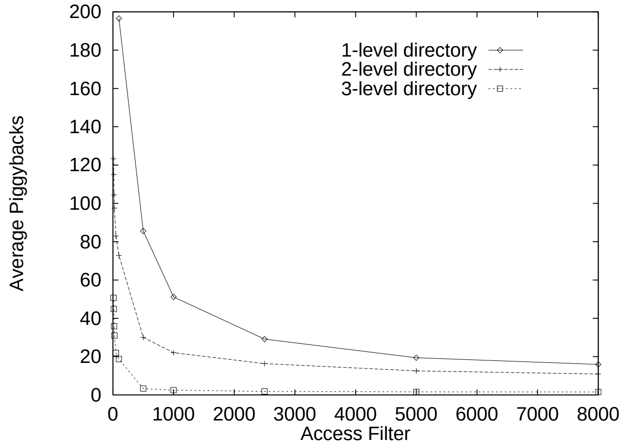
3.3.1 Volume Construction and Maintenance

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. 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 [14–16]. 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_r

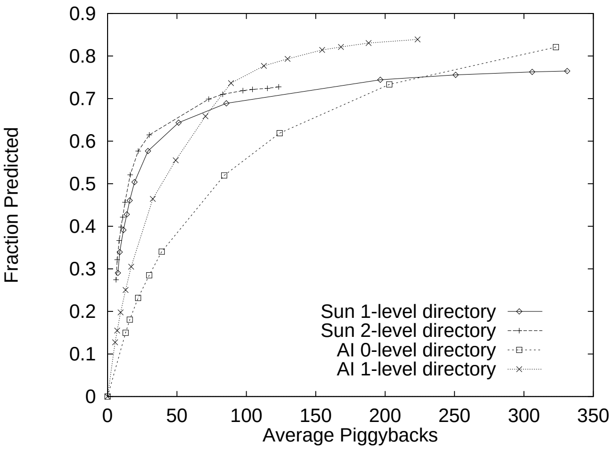


(a) AIUSA

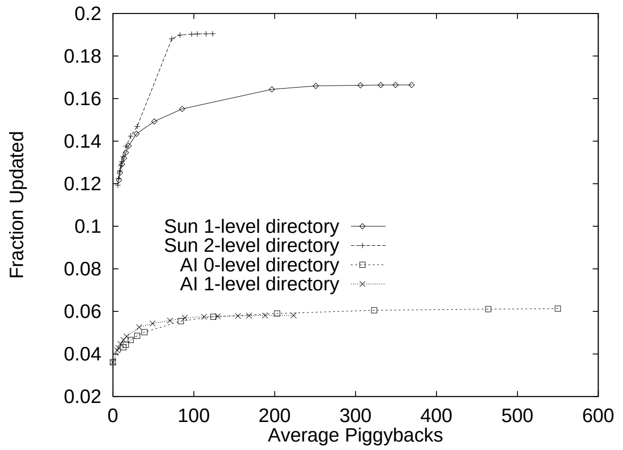


(b) Sun

Figure 2: Average piggyback size vs. access filter for directory-based volumes

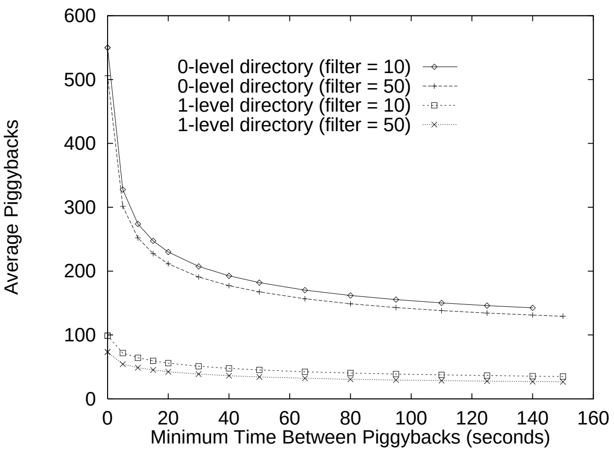


(a) Fraction predicted vs. average piggyback size

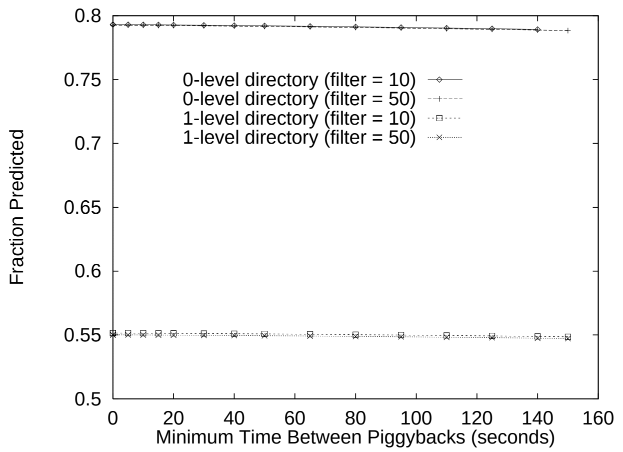


(b) Fraction updated vs. average piggyback size

Figure 3: Accuracy of directory-based volumes for Sun and AIUSA logs



(a) Average piggyback size



(b) Fraction predicted

Figure 4: Enforcing a minimum time between piggybacks for Apache logs

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; the details of processing a trace to compute these counters is described in more detail in [21]. 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 resource s. If the counter $c_{s|r}$ does not exist, we create a counter with probability inversely proportional to the product of the frequency of access 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 limiting the calculation of probability implications to pairs of resources that have the same *directory prefix*, at the expense of missing associations between resources in different directories. In addition, 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. Another way to reduce the number of counters is to create a counter $c_{s|r}$ only if resource s is reachable directly from resource r (e.g., s appears as an HREF in r), if such information is readily available [16].

To improve the accuracy and reduce the size of probability-based volumes, the volumes can be trimmed by focusing on *effective* predictions. Quite often, a request for resource s is preceded by accesses to several other resources, each of which is credited with generating a prediction for 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. With a small amount of additional processing, it is possible to measure how often an access to r generates a *new* prediction for s. If most of r's predictions are redundant (subject to an effectiveness threshold), then s is removed from r's volume, leaving only the effective predictions.

3.3.2 Performance Evaluation

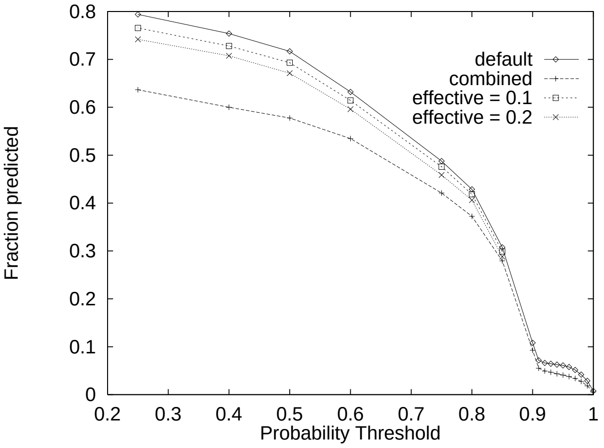
Directory-based volumes can achieve high prediction rates, at the expense of sending excess piggyback information. In contrast, estimating pairwise probabilities allows the server to construct a very accurate volume for each resource, at the expense of additional computational complexity and the possibility that some popular resources occur in multiple volumes (leading to duplication of elements in piggyback messages). 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 5(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) (shown for Sun, other logs are similar). 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 5(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 5(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.

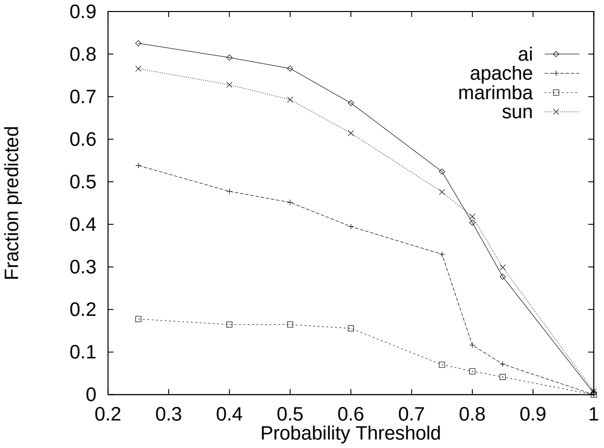
Each of the logs has resource pairs with a range of implication probabilities, as shown in Figure 5(b). High probabilities often stem from embedded images and popular HREF links in a Web page. High prediction rates can always be achieved using very large volumes. Large volumes, however, incur large average piggyback sizes and a high rate of false predictions. Figure 6 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 3(a), the probability-based volumes achieve a high prediction rate with a lower piggyback size. Thinning the probability-based volumes by removing implications that were not effective or pairs in different 1-level directories offers a significant reduction in the piggyback size, particularly for the Sun logs, as shown in Figure 6(b).

Focusing on effective predictions significantly reduces the size of piggyback messages without reducing the prediction rate. This substantially increases the fraction of true predictions, as shown in Figure 7 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 7(b), however, this is not always the case. This non-monotonicity 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 pairs that fall below the effectiveness threshold (as discussed in Section 3.3.1) not only reduces the average size of piggyback messages, but also has the 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 overheads (e.g., prefetching and cache replacement) must strike a careful balance between *recall* (the fraction predicted in Figure 6) and *precision* (the true predictions in Figure 7). These two metrics are orthogonal but we would like to increase recall if precision drops, for a particular volume construction scheme. Removal of implications that were not effective resulted in better tradeoffs between these two metrics, and considerable improvement was obtained for the Sun logs. Figure 8 summarizes the relationship between these

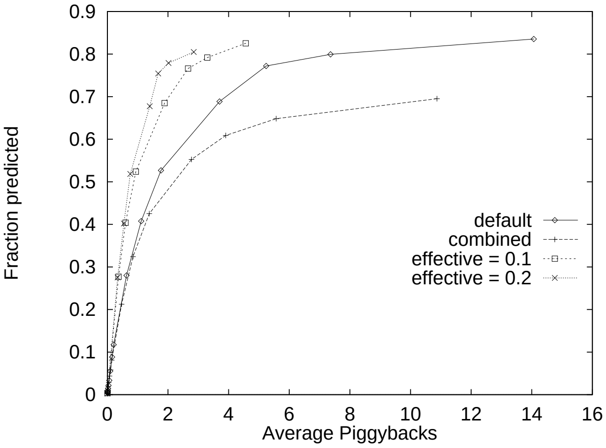


(a) Sun logs (various techniques)

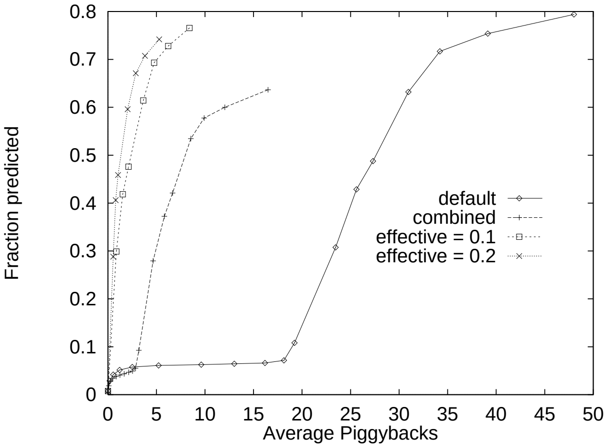


(b) All logs (effective threshold 0.1)

Figure 5: Fraction predicted vs. probability threshold for probability-based volumes

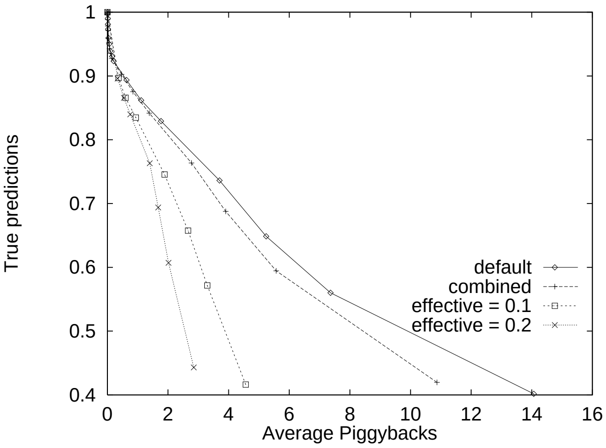


(a) AIUSA

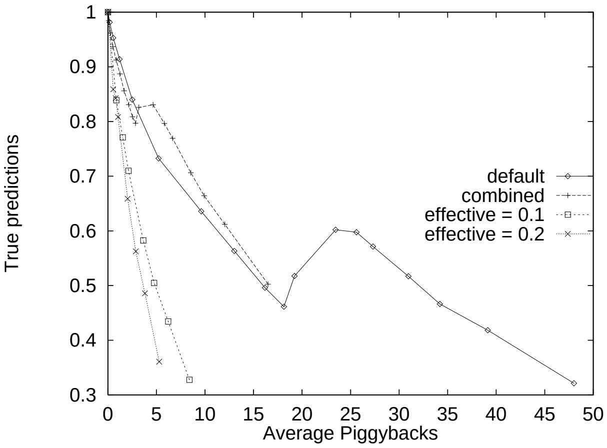


(b) Sun

Figure 6: Fraction predicted vs. average piggyback size for probability-based volumes



(a) AIUSA



(b) Sun

Figure 7: True prediction vs. average piggyback size for probability-based volumes

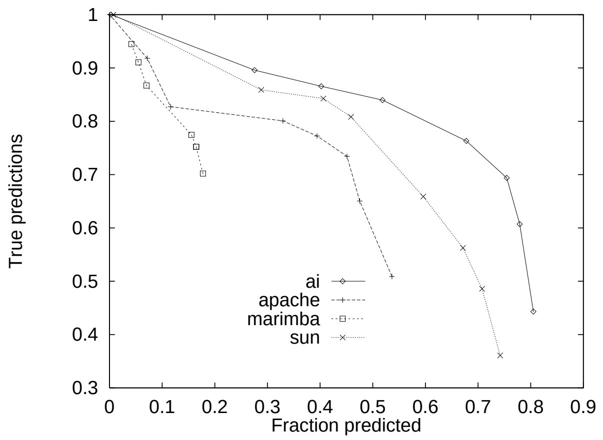


Figure 8: Precision vs. recall

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 fraction predicted 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. 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.

The update fraction metric captures the effectiveness of using piggybacked volumes to update the cache contents. The second column of Table 1 shows the fraction of requests that were preceded by a request for the same resource by the same proxy within the last 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 resource by the same proxy within the last 5 minutes. This could be viewed as if the cache already had a fresh copy. 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 piggyback in the last five minutes (i.e., sending volumes resulted in *new* updates to cached resources). The parenthetical figures in columns 3 and 4 are 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 update fraction 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 3(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 update fraction 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.

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. 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 proxy applications like cache coherency and, to a lesser extent, cache replacement, where extra piggyback information does not incur a significant cost at the proxy.

4 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 [10, 20]. 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 [23] should be very effective in reducing the amount of data transfer, since most changes are small, relative to the size of the resource. 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 3.3.2) enabled a priori refreshment for an additional 22%–46% of requests made to cached resources, using average piggyback sizes of only 1–5.

Prefetching: The piggyback message can be used by the proxy to prefetch resources, 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. 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 ask 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 prediction fraction metric measures the overhead of futile versus useful prefetches. Our most accurate volumes (see Figure 8) 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

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% (31%)	2.9
Apache	11.5%	5.4% (47%)	2.2% (19%)	1.6
Sun	23.7%	9.6% (41%)	11.0% (46%)	5.0

Table 1: Update fraction for probability-based volumes

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 removing 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 impose as much load on the network as an incorrect prefetching decision. The fraction updated metric shows that the piggyback messages generate a considerable number of predictions for requests for cached resources. In addition, these predictions are accurate. Drawing on these promising initial results, we have performed an extensive comparison of server-assisted cache replacement policies [24].

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 contain 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 the path between the proxy and the server 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 connection 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 3.3.2) inform the client with meta-attributes prior to issuing 55-80% of requests, while using very small average piggyback sizes.

5 Conclusions and Future Work

We have presented an end-to-end approach that examines the problem of network overload caused by the exponential increase in Web traffic. Our holistic approach permits piggybacked exchange of the 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 piggyback information to the various proxies. 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 (see Appendix A). Our proposed protocol changes can be used in a variety of applications such as prefetching, cache coherence, and cache validation. We also showed how the required changes can be expressed in HTTP 1.1.

As future work, we are formalizing a filter language that can be the basis for proxy-specific tailoring of piggybacks. Additional information that could be piggybacked includes information about popular resources gathered in a separate volume. 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, coherency, and the use of multi-level caches. In addition, we are studying ways for the proxy to piggyback information to the server about accesses that are satisfied at the cache. The combination of proxy-supplied information and server-generated hints provides a effective framework for improving end-to-end performance.

6 Acknowledgments

We thank Sandy Irani, Jeff Mogul, Pawan Goyal, and Craig Wills for their detailed comments on an earlier version of this paper. We also thank Dan Duchamp for his suggestions. Dave Kristol and Henrik Frystyk Nielsen kindly answered questions regarding HTTP 1.1 syntax. We are indebted to all the organizations who shared log data; many thanks to Sun Microsystems, Marimba Inc., Apache, Amnesty International USA, and Digital Equipment Corporation. We also thank the anonymous referees for their thorough comments and suggestions.

A Client and Server Logs

The set of available client, proxy, and server traces does 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-processed 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,

Client Log (days)	Requests (millions)	Distinct Servers	Unique Resources
Digital (7)	6.41	57,832	2,083,491
AT&T (18)	1.11	18,005	521,330

Table 2: Client log characteristics

Server Log (days)	Number of Requests	Number of Clients	Requests per Source	Unique Resources
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 3: Server log characteristics

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. Hence, 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 evaluated directory-based and probability-based volumes, and estimated the usefulness of the piggyback information for various proxy policies.

We used client logs from AT&T and Digital [10] and server logs from Amnesty International USA, Marimba Inc., Apache Group, and Sun Microsystems. Table 2 summarizes the key information about the client logs, which have been described in detail in previous work [20, 23]. 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 2083491 unique resources accessed. In the AT&T log just 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 521330 unique resources accessed [10]. For resources that were accessed at least twice, about 15% of the responses in the AT&T log reflected that a response had changed [25]. 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 3. 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 [26]. The Marimba server log was obtained from a site that served small amounts of data with practically all requests using the POST method (to transmit data from the client to the server) rather than GET.

We deleted apparent uncachable responses (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`). In addition, our analysis focused on resources that were accessed at least ten times. These resources account for $98 - 99\%$ of requests. Removing the requests for unpopular resources reduces the complexity of generating probability-based volumes. By ignoring unpopular resources during volume construction, we avoid generating unnecessary piggyback information.

References

[1] B. M. Duska, D. Marwood, and M. J. Feeley, "The measured access characteristics of World-Wide-Web client proxy caches," in *Proc. USENIX Symp. on Internet Technologies and Systems*, pp. 23–35, December 1997. `http://www.cs.ubc.ca/spider/marwood/Projects/SPA/wwwap`.

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

[3] R. Caceres, F. Douglis, A. Feldmann, G. Glass, and M. Rabinovich, "Web proxy caching: The devil is in the details," in *Proc. Workshop on Internet Server Performance*, June 1998. `http://www.cs.wisc.edu/~cao/WISP98.html`.

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

[5] P. Cao and S. Irani, "Cost-aware WWW proxy caching algorithms," in *Proc. USENIX Symp. on Internet Tech-*

nologies and Systems, pp. 193–206, December 1997.
http://www.cs.wisc.edu/~cao/papers/gd-size.html.

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

[7] V. Cate, "Alex – A global filesystem," in *Proc. USENIX File System Workshop*, pp. 1–12, May 1992.

[8] A. Dingle and T. Partl, "Web cache coherence," in *Proc. World Wide Web Conference*, May 1996.
http://www5conf.inria.fr/fich_html/papers/P2/Overview.html.

[9] S. D. Gribble and E. A. Brewer, "System design issues for Internet middleware services: Deductions from a large client trace," in *Proc. USENIX Symp. on Internet Technologies and Systems*, December 1997.
http://www.usenix.org/events/usits97.

[10] B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the World Wide Web," in *Proc. USENIX Symp. on Internet Technologies and Systems*, pp. 1–12, December 1997.
http://www.research.att.com/~bala/papers/pcv-usits97.ps.gz

[11] V. N. Padmanabhan and J. C. Mogul, "Improving HTTP latency," *Computer Networks and ISDN Systems*, vol. 28, pp. 25–35, December 1995.
http://www.ncsa.uiuc.edu/SDG/IT94/Proceedings/DDay/mogul/HTTPLatency.html.

[12] 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 *Proc. ACM SIGCOMM*, pp. 155–166, August 1997.
http://www.inria.fr/rodeo/sigcomm97/program.html.

[13] J. C. Mogul, "Hinted caching in the Web," in *Proc. ACM SIGOPS European Workshop*, pp. 129–140, 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 *Proc. ACM Inter. Conf. 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] Z. Jiang and L. Kleinrock, "Prefetching links on the WWW," in *Proc. IEEE Inter. Conf. on Communications*, pp. 483–489, June 1997.
http://millennium.cs.ucla.edu/~jiang/Research/Publication/prefetch.ps.

[17] M. Crovella and P. Barford, "The network effects of prefetching," in *Proc. IEEE INFOCOM*, pp. 1232–1240, April 1998.
http://www.cs.bu.edu/faculty/crovella/papers.html.

[18] T. M. Kroeger, 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*, pp. 13–22, December 1997.
http://www.cse.ucsc.edu/~tmk/ideal.ps.

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

[20] B. Krishnamurthy and C. E. Wills, "Piggyback server invalidation for proxy cache coherency," in *Proc. World Wide Web Conference*, pp. 185–194, April 1998.
http://www.research.att.com/~bala/papers/psi-www7.ps.gz

[21] E. Cohen, B. Krishnamurthy, and J. Rexford, "Improving end-to-end performance of the web using server volumes and proxy filters," Tech. Rep. 980206-01, AT&T Labs – Research, February 1998.
http://www.research.att.com/~bala/papers/mafia-tm.ps.gz.

[22] R. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, and T. Berners-Lee, "Hypertext transfer protocol – HTTP/1.1,", Internet Draft, March 13 1998. This is a work in progress.
http://www.w3.org/Protocols/HTTP/1.1/draft-ietf-http-v11-spec-rev-03.txt.

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

[24] E. Cohen, B. Krishnamurthy, and J. Rexford, "Evaluating server-assisted cache replacement in the web," in *Proc. European Symposium on Algorithms*, August 1998.
http://www.research.att.com/~bala/papers/esa98.ps.gz

[25] F. Douglis, A. Feldmann, B. Krishnamurthy, and J. Mogul, "Rate of change and other metrics: A live study of the World Wide Web," in *Proc. USENIX Symp. on Internet Technologies and Systems*, pp. 147–158, December 1997.
http://www.research.att.com/~bala/papers/roc-usits97.ps.gz

[26] M. F. Arlitt and C. L. Williamson, "Internet Web servers: Workload characterization and implications," *IEEE/ACM Trans. on Networking*, vol. 5, pp. 631–644, October 1997.
ftp://ftp.cs.usask.ca/pub/discus/paper.96-3.ps.Z.