When does a hit = a miss?

Brian D. Davison[†] Chandrasekar Krishnan[‡] Baoning Wu[†]

[†]*Computer Science & Engineering*
Lehigh University
19 Memorial Dr. W., Bethlehem, PA 18015
`davison,baw4@cse.lehigh.edu`

[‡]*Fatline Corporation*
4865 Sterling Drive, Suite 200
Boulder, CO 80301
`chandri@fatline.com`

Abstract

The Simultaneous Proxy Evaluation (SPE) architecture allows one to evaluate implemented proxies in a live network environment and be able to make performance comparisons between black-box systems. When using a real workload on a live network with real content, SPE is additionally able to evaluate prefetching proxies.

The SPE architecture requires, as a component, a proxy cache that serves all cache hits with the same response time as misses. The incorporation of a proxy cache that does not reduce end-user retrieval latencies hides the presence of the cache. By slowing down the response times of cache hits to that of cache misses, a client will not be able to distinguish between the two. Squid is modified to perform in this fashion. This paper describes the relevant issues, the implementation, and the successful evaluation of this modified Squid proxy cache.

1 Introduction

Sometimes, Web caching researchers endeavor to optimize cache hit metrics without regard to the metrics that matter to the customer or end user. The customer might care deeply about whether the resulting proxy cache will save the organization money. In that case, byte hit rates may indeed be a strong factor in the analysis. For end users, however, the benefits of a proxy cache are likely to be more qualitative than quantitative — does the Web feel faster with or without the cache?

In contrast, this work recounts progress toward building a proxy cache that does *not* make the Web any faster for its clients. In fact, it is specifically designed to provide a response time for a cache hit that is indistinguishable from a cache miss. Thus, a client of such a cache should not be able to tell (or care) whether the response was served from cache or whether it was fetched from an origin server. Performance from the client's perspective is effectively equivalent.

While one can imagine devious Internet Service Providers or organizational IT staff wanting to reap the bandwidth benefits of a cache without providing user-perceived response time benefits, this is not our motivation. A proxy cache that serves cached content as slowly as a miss is a necessary component of the Simultaneous Proxy Evaluation (SPE) architecture [10], which we summarize in Section 2.2.

In our work toward implementing a fully functional version of the SPE architecture called LEAP (Lehigh Evaluation Architecture for Proxies), we have modified a version (2.5.PRE7) of the publicly available Squid proxy server [31] to have this property.[1] Our goal for this portion of our project was to make the overall response time of a cache hit to be indistinguishable from that of a miss.

In this paper we describe our work, the issues involved, and experimental tests that evaluate the progress we have made toward our goal. In the next section we provide relevant background for the reader, giving our motivation in further detail and discussing related work. In Section 3 we go on to describe the changes we had to make to Squid. The performance of our system is evaluated in Section 4, and shows that the difference in response times of a hit and a miss of the same object in our system is considerably smaller

[1]Source code for our modifications as well as project information can be obtained from the LEAP home page: `http://wume.cse.lehigh.edu/projects/LEAP/`.

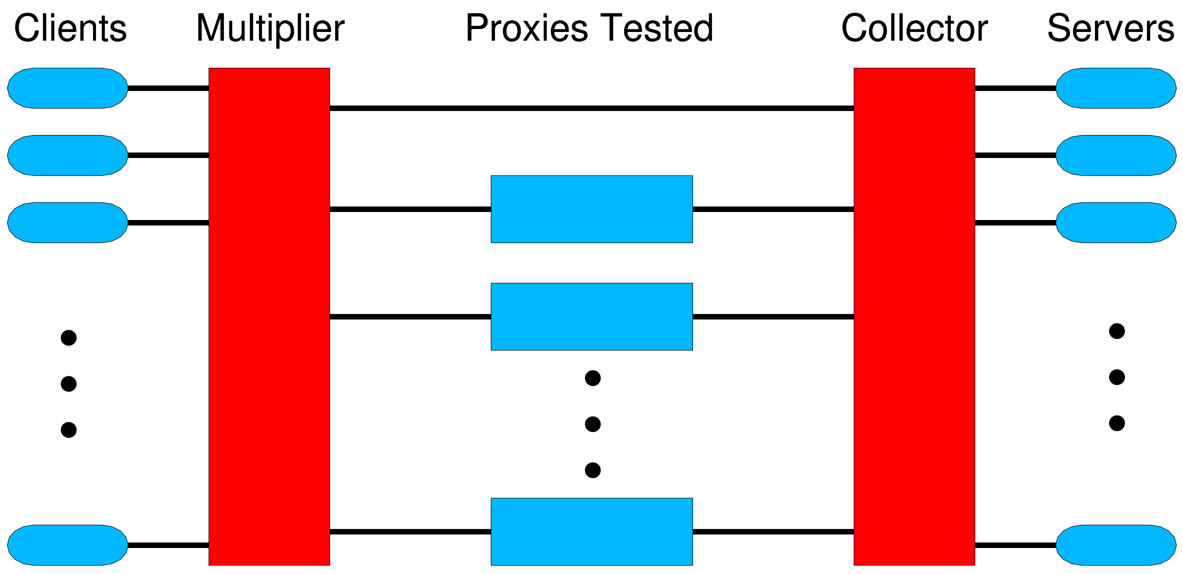


Figure 1: The Simultaneous Proxy Evaluation architecture, highlighting communication between modules. All client requests are sent to the proxy Multiplier, which sends a copy to each proxy cache being evaluated.

than the difference in response times of two misses for the same object. We summarize our contributions in Section 5.

2 Background

In this section we provide background material to help the reader understand the issues involved in implementing our system. We motivate our development by describing the difficulty in evaluating prefetching proxy caches and the implementation of the SPE architecture. We position our work in comparison to others by examining related work, and focus on the relevant issues using transaction timelines.

2.1 Motivation

The primary reason to build a proxy cache that does not reduce end-user retrieval latencies is to hide the ability of the proxy to cache content. By slowing down the response times of cache hits to that of cache misses, the client will not be able to distinguish between the two. One can also imagine a cautious proxy tester wanting to evaluate the change in performance attributable just to bandwidth reductions and not to changes in human behavior attributable to improvements in retrieval times as perceived by the end user.

Our particular motivation is considerably more grounded. Previously, we have examined evaluation methodologies for proxy caches [11] and identified the difficulty of evaluating prefetching proxies under current schemes. A preloading proxy cache typically chooses what to preload based on either user retrieval history (e.g., [27, 20]) or the content of the current or recently retrieved pages (e.g., [7, 6, 17, 29, 14, 13]). Neither are modeled well (if at all) in artificial workloads and captured traces which are typically used in evaluation. In the case where the workload is a captured trace from real users, the content is typically unavailable (at least not in the form that the users saw). Additionally, since a preloading proxy cache may choose to preload content that is never actually used, a captured trace will not be able to provide resource characteristics such as size or retrieval time. To address the difficulty of prefetching proxy evaluation, we proposed the Simultaneous Proxy Evaluation architecture [10], which we briefly describe below.

2.2 Simultaneous Proxy Evaluation

The Simultaneous Proxy Evaluation (SPE) architecture allows one to evaluate implemented proxies in a live network environment and be able to make performance comparisons between black-box systems. By using a live network and real content, SPE is additionally able to evaluate prefetching proxies.

Figure 1 depicts the SPE architecture. Clients are configured to connect to a non-caching proxy, the

Multiplier. Requests received by the Multiplier are sent to each of the proxies which are being evaluated. Instead of letting the test proxies retrieve the objects directly from the origin server, which increases the traffic in the network, they are configured to use a parent proxy cache, the *Collector*, which interacts with the origin servers. This prevents two types of problems that might otherwise arise: 1) an increase in traffic loads on the network or at the origin server, and 2) side-effects from multiple copies of non-idempotent requests [15, 12]. The Multiplier also sends the requests directly to the Collector to use the responses to service the client requests and validate the responses from the test proxies.

Thus the Multiplier does not perform any caching or content transformation. It forwards copies of each request to the Collector and the test proxies, receives responses, performs validation and logs the characteristics of each request and response to evaluate the test proxies. The clients view the Multiplier as a standard HTTP [15] proxy.

In contrast, the Collector is a proxy cache with additional functionality:

- *It must cache normally uncacheable responses.* This prevents multiple copies of uncacheable responses from being generated when each tested proxy makes the same request. This aspect is not the focus of this paper, and so is not discussed further.

- *It will replicate the time cost of a miss for each hit.* By doing so we make it look like each proxy has to pay the same time cost of retrieving an object.

When SPE was first introduced, some significant concerns were raised regarding the difficulty of building such a system, and in particular a working Collector. This paper is intended to demonstrate the viability of the approach we have taken to replicate miss response times.

Finally, we note that the SPE architecture was not proposed for use in testing peak workload performance or to determine proxy failure modes. Thus, the capabilities provided by SPE complement existing proxy performance evaluation techniques, such as those we describe below.

2.3 Related work

A number of researchers have proposed proxy cache (or more generally just Web) benchmark architectures (e.g., the Wisconsin Proxy Benchmark [4, 3], Web-Monitor [1, 2], hbench:Web [23], httperf [25], Surge [5] and Web Polygraph [30]). Some use artificial traces; some base their workloads on data from real-world traces. They are not, however, principally designed to use a live workload or live network connection, and are generally incapable of handling prefetching proxies.

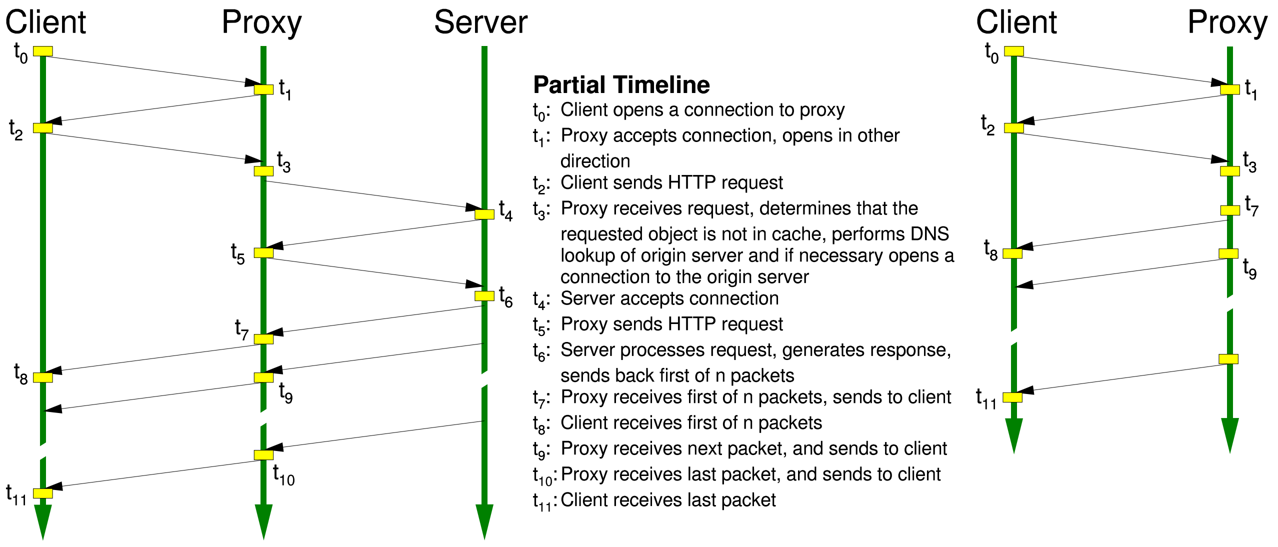
Koletsou and Voelker [19] built the Medusa Proxy, which is designed to measure user-perceived Web performance. It operates similarly to the Multiplier in SPE in that it duplicates requests to different Web delivery systems and compares results. It also can transform requests, e.g., from Akamaized URLs to URLs to the customer's origin server. The primary use of this system was to capture the usage of a single user and to evaluate (separately) the impact of using either: 1) the NLANR IRCache proxy cache hierarchy, or 2) the Akamai content delivery network.

Liu *et al.* [21] describe experiments measuring connect time and elapsed time for a number of workloads by replaying traces using Webjamma [18]. The Webjamma tool plays back HTTP accesses read from a log file using the GET method. It maintains a configurable number of parallel requests, and keeps them busy continuously. While Webjamma is capable of sending the same request to multiple servers so that the server behaviors can be compared, it is designed to push the servers as hard as possible.

While the work cited above is concerned with performance, and may indeed be focused on user perceived latencies (as we are), there are some significant differences. For example, the SPE approach is designed carefully to minimize the possibility of unpleasant side effects — we explicitly attempt to prevent multiple copies of a request instance to be issued to the general Internet (unlike Koletsou and Voelker). Similarly, SPE minimizes any additional bandwidth resource usage (since only one response is needed).

2.4 Transaction timelines

In order to reproduce miss timings as hits, we need to be careful about some of the details. In Figure 2a, we show a transaction timeline depicting the sequence of events when using a non-caching proxy (or equiv-



Partial Timeline

t_0: Client opens a connection to proxy
t_1: Proxy accepts connection, opens in other direction
t_2: Client sends HTTP request
t_3: Proxy receives request, determines that the requested object is not in cache, performs DNS lookup of origin server and if necessary opens a connection to the origin server
t_4: Server accepts connection
t_5: Proxy sends HTTP request
t_6: Server processes request, generates response, sends back first of n packets
t_7: Proxy receives first of n packets, sends to client
t_8: Client receives first of n packets
t_9: Proxy receives next packet, and sends to client
t_{10}: Proxy receives last packet, and sends to client
t_{11}: Client receives last packet

(a) Sequence for proxy cache miss.

(b) Sequence for proxy cache hit.

Figure 2: Transaction timelines showing the sequence of events to satisfy a client need. The diagrams do not show TCP acknowledgments, nor the packet exchange to close the TCP connection.

alently, when the proxy does not have a valid copy of the content requested). In contrast, the transaction timeline in Figure 2b illustrates the typical sequence of events when a caching proxy has a valid copy of the requested content and returns it to the client.

In the case that the client has an idle persistent connection to the proxy, the transaction would start with t_2. If the proxy had an idle connection to the desired origin server, a new connection would not be necessary (merging the actions at t_3 and t_5 and eliminating the activity between). From the client's perspective, the time between t_2 and t_0 constitutes the *connection setup time*. The time between t_8 and t_2 constitutes the *initial response time*. The time between t_{11} and t_8 is the *transfer time*. The complete response time would be the time elapsed between t_{11} and t_0.

3 Issues and Implementation

In this work, we have made changes only to the source code of Squid to implement our partial SPE tool. We have not modified the underlying operating system.

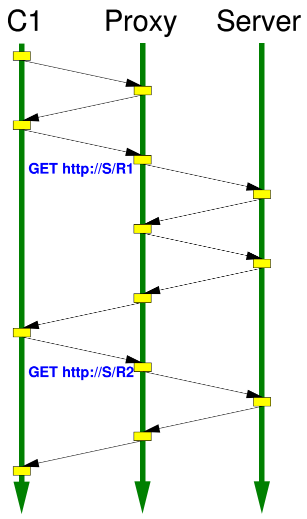
The basic approach used in our system is as follows: if the object requested by a client is not present in the cache, we fetch it from the origin server. We record the connection setup time, response time and transfer time from the server for this object. In subsequent requests from clients for the same object in which the object is in the cache, we delay some time before sending the first chunk of the cached object, and we also delay some time before sending subsequent chunks, so that even though this request is a cache hit, the total response time experienced is the same as a cache miss.

The function *clientSendMoreData* in *client_side.c* takes care of sending a chunk of data to the client. We have modified it so that for cached objects, the appropriate delays are introduced by using Squid's built-in event scheduling apparatus. Instead of sending the chunk right away, we schedule for the desired time the execution of a new function to send the chunk instead.

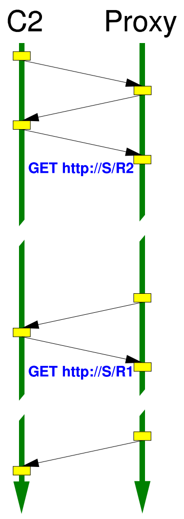
Given our goal of replicating HTTP response timings, we realize we will have to address a number of issues. Here we describe some of the most significant issues and how we have addressed them.

3.1 Persistent connections

Issue. Most Web clients, proxies and servers support persistent connections (which are optional in HTTP/1.0 and the default in HTTP/1.1). Persistent connections allow subsequent requests to the same server to re-use an existing connection to that server, obviating the TCP connection establishment delay that would otherwise occur. Squid supports persistent connections between client and proxy and between proxy and server. This process is sometimes called *connection caching* [9], and is a source of difficulty



(a) Client re-uses persistent connection.



(b) Second client finds cached data.

Figure 3: Transaction timelines showing how persistent connections complicate timing replication.

for our task.

Consider the case in which client $C1$ requests resource $R1$ from origin server S via a proxy cache (see the transaction timeline in Figure 3a). Assuming that the proxy did not have a valid copy of $R1$ in cache, it would establish a connection to S and request the object. When the request is complete, the proxy and server maintain the connection between them. A short time later, $C1$ requests resource $R2$ from the same server. Again assuming the proxy does not have the resource, the connection would be re-used to deliver the request to and response from S. This arrangement has saved the time it would take to establish a new connection between the proxy and server from the total response time for $R2$.

Assume, however, a short time later, client $C2$ also requests resource $R2$, which is now cached (Figure 3b). Under our goals, we would want the proxy to delay serving $R2$ as if it were a miss. But a miss under what circumstances? If it were served as the miss had been served to $C1$, then the total response time would be the sum of the server's initial response time and transfer time when retrieving $R2$. But if $C1$ had never made these requests, then a persistent connection would not exist, and so the proxy would have indeed experienced the connection establishment delay. Our most recent measurement of that time was from $R1$, so it could be re-used. On the other hand, if $C2$ had earlier made a request to S, a persistent connection might be warranted. Similarly, if $C2$ were then to request $R1$, we would not want to replicate the delay that was experienced the first time $R1$ was retrieved, as it included the connection establishment time.

In general it will be impossible for the Collector to determine whether a new request from a tested proxy would have traveled on an existing, idle connection to the origin server. The existence of a persistent connection is a function of the policies and resources on either end. The Collector does not know the idle timeouts of either the tested proxy nor the origin server. It also does not know what restrictions might be in place for the number of idle or, more generally, simultaneous connections permitted.

Implementation. While an ideal SPE implementation would record connection establishment time separately from initial response times and transfer times, and apply connection establishment time when persistent connections would be unavailable, such an approach is not possible (as explained above). Two simplifications were possible — to simulate some policies and resources on the proxy and a fixed maximum idle connection time for the server side, or to serve every response as if it were a connection miss. For this implementation, we chose the latter, as the former would require the maintenance of new data structures on a per-proxy-and-origin-server basis, as well as the design and simulation of policies and resources.

The remaining decision is whether to internally use persistent connections to the origin servers from the Collector. While a Collector built on Squid (as ours is) has the mechanisms to use persistent connections, idle connections to the origin server consume resources (at both ends), and persistent connections may skew the transfer performance of subsequent responses as they will benefit from an established transfer rate and reduced effects of TCP slow-start.

Therefore, in our implementation, we modified Squid to never re-use a connection to an origin server. This effectively allows us to serve every response as a connection miss, since the timings we log correspond to connection misses, and accurately represent data transfer times as only one transfer occurs per connection.

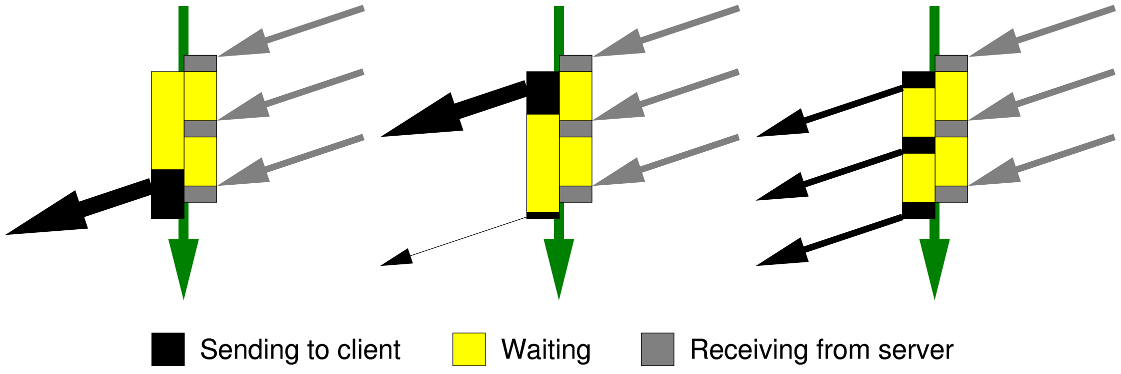


Figure 4: Possible inter-chunk delay schedules for sending to client.

3.2 DNS resolution

Issue. The first request to an origin server requires a DNS lookup to determine the server's IP address. In some cases, this process can take a noticeable amount of time [8]. For subsequent HTTP requests to the same server, no lookup will be required, as the IP address will have been saved in cache. It is important that even though one proxy may have issued a request that encountered a delay because of DNS resolution times, requests by other proxies should still experience the same delay.

Implementation. In order to provide no advantage to slower proxies that make a request later than a faster one, we incorporate the DNS resolution actions as part of the server connection establishment time. This helps to more accurately replicate the delay seen by the first request.

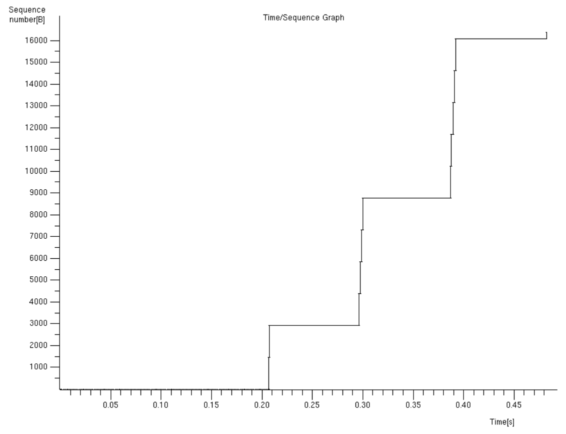
3.3 System scheduling granularity

Issue. Most operating systems have a limit on the granularity of task scheduling that is possible. Unless it is modified within the OS, this limit will provide a lower-bound on the ability to match previously recorded timings.

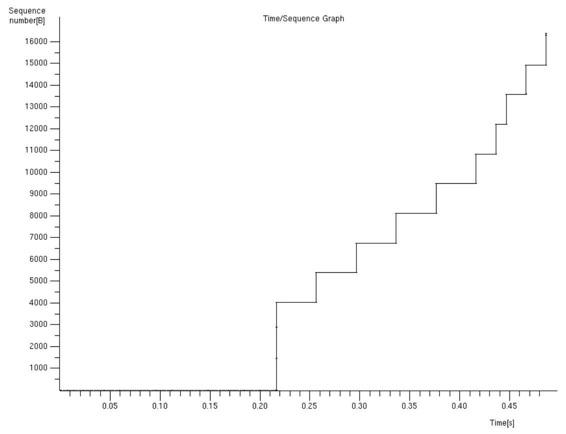
Implementation. In Linux as well as other UNIX-like systems on the Intel architecture, the `select(2)` system call has granularity of 10ms. Thus, if we tell `select` to wait at most 15ms, and no other monitored events occur, it will likely return after approximately 20ms. Thus, in our initial implementation a large difference would accumulate between what we wanted to delay and the actual amount of time delayed. In order to solve this problem, we use the following method: given a granularity of 10ms, we must decide to round up or down to an even 10ms. That is, if we need to delay 15ms, do we tune it to 10ms or to 20ms? In our implementation, we decide this probabilistically. We first extract the largest multiple of 10ms from our desired delay. The fraction of 10ms remaining is then used as a probability to delay an additional 10ms, and thus we choose stochastically to request a multiple of 10ms that is at most 10ms larger or smaller than our original need.

3.4 Inter-chunk delay times

Issue. When Squid receives a non-trivial response from an origin server, it groups the data received in a single call into an object it calls a chunk, which it then stores into cache. Chunks then are the content of at minimum one packet, and possibly many packets, depending on the rate at which data is being received. In a cache miss, those chunks are sent on to the client as they are received. Since our goal is to replicate the miss experience as much as possible, it would be undesirable to wait and then send all data at the end, even though that might be a simplistic approach to replicate overall response time. Likewise, sending all data but the last byte until the desired time had passed would also be unreasonable, as it does not come close to replicating reality. Ideally, our system would send data at the same rate as it was received, and with the same inter-chunk delays. These three scenarios are depicted graphically in Figure 4. In each portion, the timings of when chunks were received are re-drawn on the right, while the various schedules for inter-chunk delays in the case of a cache miss are



(a) SPE Collector miss



(b) SPE Collector hit

Figure 5: Time/sequence graphs for one object.

drawn on the left of the timeline.

The differences between these conditions are real — modern browsers will parse the HTML page as it is received (i.e., in chunks) to find embedded resources such as images to fetch, and when found, issue new requests for them. Prefetching proxies may in fact do something similar (e.g., [7, 6]). Likewise, images with a progressive encoding may be rendered by a browser with low-resolution when the initial data is received and then be refined as the remaining data arrive.

Implementation. As described above, it may be insufficient to merely replicate the total response time — when data arrives within the response time period may be important. The overhead of recording the per-chunk delay was deemed too high for implementation. Instead, we replicate the average delay between chunks, as this only requires a fixed per-object storage overhead.

In order to accurately replicate the overall response time, we record the time difference between the time at which we receive the client request (t_3) and the time at which we begin to send the response (t_7) thus summing the server connection establishment time and the initial response time. In addition, we record the time difference between receiving the first chunk of response from the origin server (t_7) and the time when we receive the last chunk of response from the origin server (t_{10}) as the transfer time. As shown in Figure 2, this also corresponds to the time difference between just before sending the first chunk of response to the client and just before sending the last chunk of response to the client. So for subsequent requests for the same object, we intentionally add delays (in particular, before each chunk) so that the total time experienced between events t_3 and t_{10} are identical to the miss case.

We insert delays in two situations: before the first chunk (i.e., before event t_7) we delay the amount of time it took for the connection establishment and initial response time from the origin server. Before sending subsequent chunks, we delay a portion of the transfer time. In that situation the delay portion is calculated by the following formula:

$$delay = \frac{chunksize}{totalsize} * transfertime$$

In our experiments we found that the above formula is not accurate enough as there is a non-negligible chance that the randomly chosen errors (because of select call granularity) will accumulate. Therefore, we use a global optimization approach and change the formula to be:

$$delay = \frac{chunksize}{restofsize} * restoftime$$

where $restofsize$ is defined as the total remaining bytes of the cached object that has yet to be sent, and the $restoftime$ is defined as the delay time remaining that is to be distributed to the remaining chunks of the object.

Figures 5a and b demonstrate the approach. In 5a, the Collector passes packets to the client as they are received from the origin, in effectively four blocks. In

5b, the Collector closely replicates the start and end of the transmission, but uses ten blocks to do so.

3.5 Persistent state in an event-driven system

Issue. Our implementation requires the recording of various time values, some of which are object-based, and some of which are request-based.

Implementation. Since Squid uses a single process, we cannot easily use global storage for persistent state. Thus, we have extended the structures of `_connStateData`, `_StoreEntry`, and `_clientHttpRequest` in order to record the information for later access.

3.6 HTTP If-Modified-Since request

Issue. Not all HTTP requests return content. Instead, some respond only with header information along with response codes — telling the client that the object has moved (e.g., response code 302), or cannot be found (404), or that there is no newer version (304). The latter is of particular interest because it forces the consideration of many situations. Previously we have considered only the case in which a client makes a request and either the proxy has a valid copy of the requested object or it does not. In truth, it can be more complicated. If the client requests object R, and the cache has an old version of R, it must check to see if a modified version is available from the origin server. If a newer one exists, then we have the same case as a standard cache miss. If the cached version is still valid, then the proxy has spent time communicating with the origin server that must be accounted for in the delays seen by the client. Likewise, the client may have a copy and need to verify its freshness with the proxy. If the proxy has a valid copy, it may wish to respond directly to the client, in which case the proxy needs to add delays as if the proxy had to confer with the origin server.

Implementation. We have considered IMS requests, and the general principle we use is to reproduce any communication delays that would have occurred when communicating with the origin server. Thus, if the proxy does communicate with the origin server, then no additional connection establishment and initial response time delay needs to be added. If the proxy is serving content that was cached, then as always, it needs to add transfer time delays (as experienced when the content was received). If the response served to the client does not contain content, then no transfer time is involved.

3.7 Network stability

Issue. An object may be cacheable for an arbitrary period of time. Since there exists significant variability in end-to-end network performance on the Internet [28], network characteristics may have changed since the object was first retrieved.

Implementation. We use the times from the most recent origin server access of an object to be our gold standard. In reality, the origin server may be more or less busy and network routes to it may be better or worse. Thus, we have chosen consistent performance reproduction over technically accurate performance reproduction (which would preclude caching). Note that to minimize any adverse effects, we can limit the amount of time that we allow objects to be cached (since a new retrieval will get a new measurement of communication performance).

4 Experiments

4.1 Method

To claim success, we must argue that clients using our modified proxy would be unlikely to tell whether the proxy utilizes a cache. To provide evidence supporting this argument, we experimentally measure response times of requests for data on real origin servers. For each run of our experiments, we iterate through a list of URLs and give one at a time to *httperf* [25] to send the request through our modified proxy (as shown in Figure 6. We collect the per-request timing results captured by httperf to calculate statistics on the overall response times as seen by the client.

We repeat the run for each of the four combinations of: original Squid vs. modified Squid, and Squid with empty cache (i.e., generating all cache misses) vs. Squid with cached results of previous run (making cache hits possible) on separate identical machines (1Ghz Pentium III systems running stock RedHat 7.3 on switched full-duplex 100 Mbit/sec Fast Ethernet). Each run was repeated once per hour.

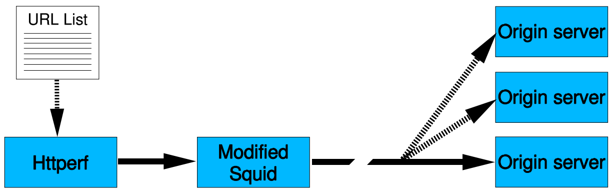


Figure 6: Experiment structure. httperf is used to send requests to our modified Squid and measure response times.

4.2 Data sets

In our experiments we use two representative data sets. To generate the first, we used the top fifty queries as reported by Lycos [22] on 24 April 2002. Those queries were fed automatically to Google [16], resulting in 10 URLs per query, for a total of 500 URLs. We eliminated three that caused problems for httperf in initial tests.

The second dataset was extracted from an NLANR IRCache proxy log (sv.ircache.net) on 18 April 2002. The intentions for this log of about half a million entries were to focus on cacheable responses,[2] so we removed all log entries that had non-200 HTTP response codes or Squid status codes other than `TCP_HIT`, `TCP_MEM_HIT`, `TCP_IMS_HIT`, `TCP_REFRESH_HIT`, and `TCP_REFRESH_MISS`. This eliminated more than 75% of the log entries, leaving only those entries corresponding to objects that had been cached by the proxy or a client of the proxy. We then arbitrarily selected the URLs that ended in `.html` and eliminated any duplicate URLs. We additionally removed those that generated HTTP errors in initial tests, resulting in a set of 1054 URLs.

4.3 Results

After running the tests hourly for more than half a day, we calculated relevant statistical properties using `strat` [24]. Examining first the IRCache data, we show the hit and miss response time distributions in Figure 7. Our primary concern, however, is in the paired differences in response times. That is, what is the typical difference in response time for a miss versus a replicated miss (i.e., a hit)? Since we are not concerned with whether the difference is positive or negative, we plot the the distribution of the absolute value of the differences in Figure 8.

In Table 1 we show various summary statistics for the two datasets. Tests on each dataset were repeated for some number of runs, and from each run we calculated the mean and median difference between response times of a cache miss versus a cache hit, and a cache miss versus a cache miss one hour later. In each row, we show the mean and standard error for the run means and the run medians. All values are in milliseconds.

For reference, the first row of Table 1 shows the results when calculating the real (not absolute value) difference between response times. The remaining rows calculate results using the absolute value of the differences. The data for the second row is otherwise identical, but now shows a significant difference in response times between the Miss-Hit and Miss-Miss cases. As expected, the typical difference for the Miss-Hit case is quite small (since our intention was to get this close to zero).

Recall from Section 4.2 that the IRCache data set was selected specifically to be cacheable. In fact, in a few instances, the results were not cacheable. If we remove these points, we get even better results, shown in the third row.

We now examine the second data set, generated from popular search engine queries. Unlike the IRCache data, this dataset contains many references (approximately two-thirds) to uncacheable objects. Under these conditions, we find the absolute difference in response times between miss and hit pairs of the same requests are much closer to the miss pairs separated by one hour, as shown in the last row.

For comparison, one might also ask what the typical difference is when an unmodified Squid is used (i.e., when hits are served as fast as possible). Results from both datasets are recorded in Table 2. In both cases, mean differences in Miss-Hit response times are quite high (similar to Miss-Miss), and the medians are in fact much higher. This confirms the fact that unmodified hits are served much faster than misses, generating as much if not more variation in response times than misses an hour apart.

The above tests compared hit performance with that of a miss. It is also important to be able to generate multiple hits to the same object with consistent response times. The modified Squid is indeed able to

[2]Proxy logs from NLANR's IRCache proxies [26] cannot be replayed in general because some entries are incomplete. To preserve privacy, all query terms (parts of the URL after a "?") are obscured.

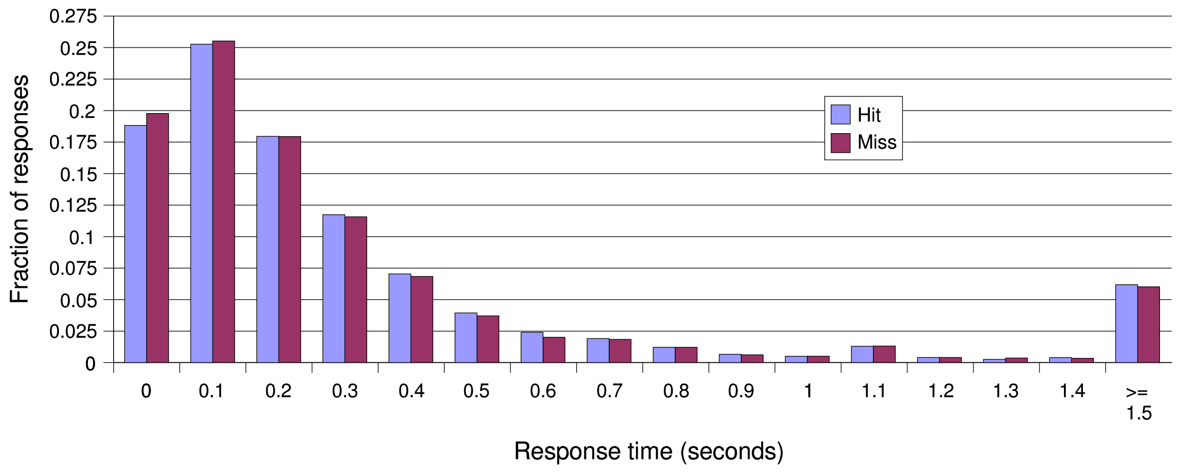


Figure 7: Distribution of cache hit and miss response times with NLANR IRCache data using our modified Squid.

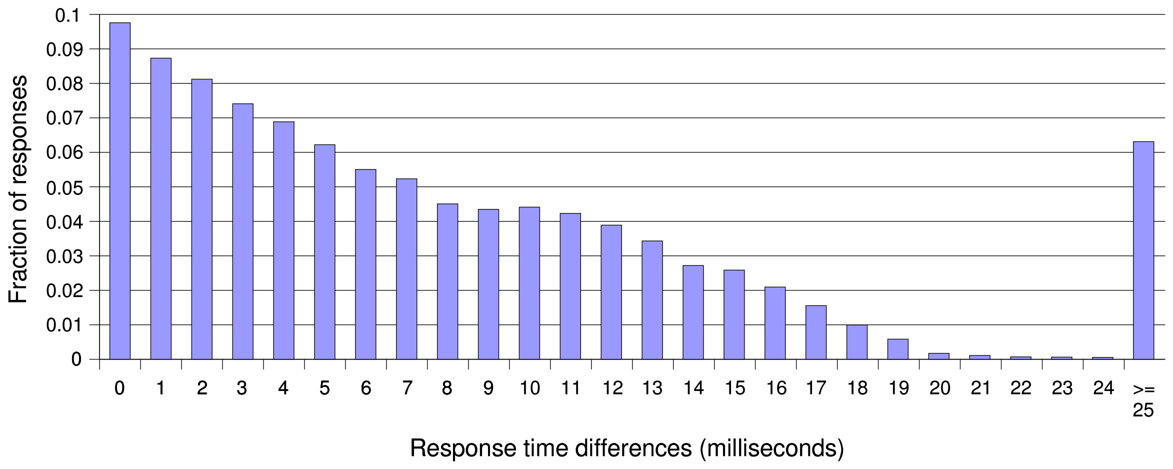


Figure 8: Distribution of absolute differences in paired response times between hits and misses with NLANR IRCache data using our modified Squid.

achieve very similar response times for repeated hits. The means of the run means and medians of the absolute difference between two hits for the same object is 6.60 ± 0.64ms and 2.03 ± 0.29, respectively.

From the above analysis, we can state that the typical difference in response times is significantly reduced for subsequent requests for the same object when our modified Squid is utilized. In fact, when only tested on fully cacheable data, we find an arguably useful mean miss-hit difference of just 7.4ms, and mean hit-hit difference of 6.6ms.

5 Summary

This paper has recounted progress toward building a proxy that serves cache hits with the same timings as cache misses. Our system attempts to give the client the same response times in either case, so that there is no advantage given to a client when the object is indeed present in cache. We have reviewed the SPE architecture, raised a number of issues relevant to this task, and described our approach in each case.

The primary contribution of this paper has been the implementation, description, and evaluation of a proxy cache modified to return cached results as if they were not cached (within a few multiples of

Data Set	# runs	Cache Miss vs. Cache Hit (ms) Mean $\pm$ StdErr of run means	of run medians	Cache Miss vs. Cache Miss (ms) Mean $\pm$ StdErr of run means	of run medians
Orig. IRCache	13	12.85 ± 4.41	5.440 ± 0.08	-18.03 ± 12.96	0.08 ± 0.16
Abs. Val. IRCache	13	54.20 ± 3.90	6.45 ± 0.07	347.95 ± 10.09	10.53 ± 0.04
Rev. Abs. IRCache	13	7.44 ± 0.82	6.34 ± 0.07	337.84 ± 11.23	10.31 ± 0.46
Popular Queries	17	366.12 ± 15.27	13.01 ± 0.19	670.71 ± 22.92	33.34 ± 1.39

Table 1: Mean and standard error of run means and medians using modified Squid.

Data Set	# runs	Cache Miss vs. Cache Hit (ms) Mean $\pm$ StdErr of run means	of run medians	Cache Miss vs. Cache Miss (ms) Mean $\pm$ StdErr of run means	of run medians
Abs. Val. IRCache	17	503.34 ± 7.55	245.62 ± 1.04	279.57 ± 8.30	10.26 ± 0.47
Popular Queries	17	600.23 ± 17.61	109.11 ± 2.50	578.07 ± 17.20	40.17 ± 1.63

Table 2: Mean and standard error of run means and medians using unmodified Squid.

the OS timing granularity). While we believe that even tighter results are possible (particularly when the implementation includes operating system modifications), this paper has shown substantial progress toward a complete SPE implementation.

Acknowledgments

This work has been supported by NSF grant ANI 9903052. We are indebted to Terrance Boult and Ross Micheals for their helpful discussions and contributions to the statistical analysis, and to Haym Hirsh, Vincenzo Liberatore, and anonymous reviewers for useful suggestions.

References

[1] J. Almeida, V. A. F. Almeida, and D. J. Yates. Measuring the behavior of a World Wide Web server. In *Proceedings of the Seventh Conference on High Performance Networking (HPN)*, pages 57–72. IFIP, Apr. 1997.

[2] J. Almeida, V. A. F. Almeida, and D. J. Yates. Web-Monitor: A tool for measuring World Wide Web server performance. *first monday*, 2(7), July 1997.

[3] J. Almeida and P. Cao. Measuring proxy performance with the Wisconsin Proxy Benchmark. *Computer Networks And ISDN Systems*, 30(22-23):2179–2192, Nov. 1998.

[4] J. Almeida and P. Cao. Wisconsin Proxy Benchmark 1.0. Available from `http://www.cs.wisc.edu/~cao/wpb1.0.html`, 1998.

[5] P. Barford and M. Crovella. Generating representative Web workloads for network and server performance evaluation. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pages 151–160, 1998.

[6] CacheFlow Inc. Active caching technology. `http://www.cacheflow.com/technology/whitepapers/active.cfm`, 2002.

[7] K. Chinen and S. Yamaguchi. An interactive prefetching proxy server for improvement of WWW latency. In *Proceedings of the Seventh Annual Conference of the Internet Society (INET'97)*, Kuala Lumpur, June 1997.

[8] E. Cohen and H. Kaplan. Prefetching the means for document transfer: A new approach for reducing Web latency. In *Proceedings of IEEE INFOCOM*, Tel Aviv, Israel, Mar. 2000.

[9] E. Cohen, H. Kaplan, and U. Zwick. Connection caching. In *Proceedings of the 31st Annual ACM Symposium on Theory of Computing*. ACM, 1999.

[10] B. D. Davison. Simultaneous proxy evaluation. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, pages 170–178, San Diego, CA, Mar. 1999.

[11] B. D. Davison. A survey of proxy cache evaluation techniques. In *Proceedings of the Fourth International Web Caching Workshop (WCW99)*, pages 67–77, San Diego, CA, Mar. 1999.

[12] B. D. Davison. Assertion: Prefetching with GET is not good. In A. Bestavros and M. Rabinovich, editors, *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, pages 203–215, Boston, MA, June 2001. Elsevier.

[13] B. D. Davison. Predicting Web actions from HTML content. In *Proceedings of the The Thirteenth ACM Conference on Hypertext and Hypermedia (HT'02)*, pages 159–168, College Park, MD, June 2002.

[14] Deerfield Corporation. InterQuick for WinGate. `http://interquick.deerfield.com/`, 2002.

[15] R. T. Fielding, J. Gettys, J. C. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol — HTTP/1.1. RFC 2616, `http://ftp.isi.edu/in-notes/rfc2616.txt`, June 1999.

[16] Google Inc. Google home page. `http://www.google.com/`, 2002.

[17] T. I. Ibrahim and C.-Z. Xu. Neural net based pre-fetching to tolerate WWW latency. In *Proceedings of the 20th International Conference on Distributed Computing Systems (ICDCS2000)*, Apr. 2000.

[18] T. Johnson. Webjamma. Available from `http://www.cs.vt.edu/˜chitra /web-jamma.html`, 1998.

[19] M. Koletsou and G. M. Voelker. The Medusa proxy: A tool for exploring user-perceived Web performance. In *Web Caching and Content Delivery: Proceedings of the Sixth International Web Content Caching and Distribution Workshop (WCW'01)*, Boston, MA, June 2001.

[20] 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 (USITS '97)*, Dec. 1997.

[21] B. Lui, G. Abdulla, T. Johnson, and E. A. Fox. Web response time and proxy caching. In *Proceedings of WebNet98*, Orlando, FL, Nov. 1998.

[22] Lycos, Inc. Lycos home page. `http://www.lycos.com/`, 2002.

[23] S. Manley, M. Seltzer, and M. Courage. A self-scaling and self-configuring benchmark for Web servers. In *Proceedings of the Joint International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '98/PERFORMANCE '98)*, pages 270–271, Madison, WI, June 1998.

[24] R. J. Micheals and T. E. Boult. Efficient evaluation of classification and recognition systems. In *Proc. of the IEEE Conf. on Computer Vision and Pattern Recognition*, Dec. 2001.

[25] D. Mosberger and T. Jin. httperf—A tool for measuring Web server performance. *Performance Evaluation Review*, 26(3):31–37, Dec. 1998.

[26] National Laboratory for Applied Network Research. A distributed testbed for national information provisioning. Home page: `http://www.ircache.net/`, 2002.

[27] V. N. Padmanabhan and J. C. Mogul. Using predictive prefetching to improve World Wide Web latency. *Computer Communication Review*, 26(3):22–36, July 1996. Proceedings of SIGCOMM '96.

[28] V. Paxson. End-to-end Internet packet dynamics. *IEEE/ACM Transactions on Networking*, 7(3):277–292, June 1999.

[29] J. E. Pitkow and P. L. Pirolli. Life, death, and lawfulness on the electronic frontier. In *ACM Conference on Human Factors in Computing Systems*, Atlanta, GA, Mar. 1997.

[30] A. Rousskov. Web Polygraph: Proxy performance benchmark. Available at `http://www.web-polygraph.org/`, 2002.

[31] D. Wessels. Squid Web proxy cache. Available at `http://www.squid-cache.org/`, 2002.