Refreshment Policies for Web Content Caches

Edith Cohen
AT&T Labs–Research
180 Park Avenue
Florham Park, NJ 07932 USA
edith@research.att.com

Haim Kaplan
Tel-Aviv University
Tel-Aviv 69978
Israel
haimk@math.tau.ac.il

Abstract—
Web content caches are often placed between end-users and origin servers as a mean to reduce server load, network usage, and ultimately, user-perceived latency. Cached objects typically have associated expiration times, after which they are considered stale and must be validated with a remote server (origin or another cache) before they can be sent to a client. A considerable fraction of cache hits involve stale copies that turned out to be current. These validations of current objects have small message size, but nonetheless, often induce latency comparable to full-fledged cache misses. Thus, the functionality of caches as a latency-reducing mechanism highly depends not only on content availability but also on its freshness. We propose policies for caches to proactively validate selected objects as they become stale, and thus allow for more client requests to be processed locally. Our policies operate within the existing protocols and exploit natural properties of request patterns such as frequency and recency. We evaluated and compared different policies using trace-based simulations.

I. Introduction

Caches are often placed between end-users and origin servers as a mean to reduce user-perceived latency, server load, and network usage (see Figure 1). Among these different performance objectives of caches, improving end-user Web experience is gradually becoming the most pronounced. Many organizations are deploying caching servers in front of their LANs, mainly as a way to speed up users Web access. Generally, available bandwidth between end-users and their Internet Service Providers (ISPs) is increasing and is complemented by short round trip times. Thus, the latency bottleneck is shifting from being between end-users and cache to being between cache and origin servers. From the viewpoint of Web sites and Content Distribution Networks (like Akamai [1]), decreasing costs of server-machines and backbone connectivity bandwidth along with increasing use of the Web for commercial purposes imply that server and network load are gradually becoming a lesser issue relative to end-user quality of service. At the limit, these trends indicate that communication time between local caches and remote servers increasingly dominates cache service-times and user-perceived latency, and that technologies which provide tradeoffs between traffic-increase and latency-decrease would become increasingly worthwhile for both Web sites and ISPs.

Servicing of a request by a cache involves remote communication if the requested object is not cached (in which case the request constitutes a *content miss*). Remote communication is also required if the cache contains a copy of the object, but the copy is *stale*, that is, its freshness lifetime had expired and it must be validated (by the origin server or a cache with a *fresh* copy) prior to being served. If the cached copy turns out to be modified, the request constitutes a content miss. Otherwise, the cached copy is *valid* and we refer to the request as a *freshness miss*. Validation requests that turn out as freshness misses typically have small-size responses but due to communication overhead with remote servers, often their contribution to user-perceived latency is comparable to that of full-fledged content misses.

Thus, cache service times can be improved by reducing both content and freshness misses. The content hit rate is measured per object or per byte and sometimes weighted by estimated object fetching cost. It is dictated by the available cache storage and the *replacement policy* used. Replacement policies for Web caches were extensively studied (e.g. [2], [3], [4], [5], [6], [7], [8], [9]). Policies that seem to perform well are Least Recently Used (LRU, which evicts the least recently requested object when the cache is full), Least Frequently Used (LFU, which evicts the least-frequently requested object), and Greedy-Dual-Size (which accounts for varying object sizes and fetching costs). Squid [10], a popular caching server software, implements the LRU policy. Under these replacement policies, however, and due to decreasing storage cost, cache hit rate is already at a level where it would not significantly improve even if unbounded storage is made available. Content availability can be improved by prefetching [11], [12], but prefetching of content involves more involved predictions and induces significant bandwidth overhead. The freshness hit rate of a cache is not directly addressed by replacement policies or captured by the content hit rate metric.

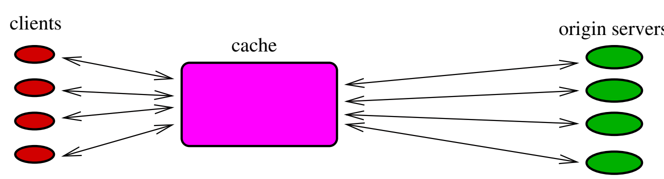

Fig. 1. Schematic configuration of Cache, clients, and origin servers

The expiration time of each object is determined when it is brought into the cache, according to attached HTTP response headers provided by the origin server. Expired content must be validated before being served. Most current caching platforms validate their content *passively* i.e. only when a client request arrives and the cached copy of the object is stale. They perform validation via a conditional `GET` request (typically this is an `If-Modified-Since (IMS) Get` request). This means that validation requests are always performed "online," while the end-user is waiting. Here we promote proactive refreshment where the cache initiates unsolicited validation requests for selected content. Such "offline" validations extend freshness time of cached objects and more client requests can be served directly from the cache. Our motivation is that the most significant cost issue associated with freshness misses is their direct effect on user-perceived latency rather than their effect on server and network load, and thus it is worth performing more than one "offline" validation in order to avoid one performed "online." We formalize a cost model for proactive refreshment, where overhead cost of additional validation requests to origin servers is balanced against the increase in freshness hits.

We propose and evaluate *refreshment policies*, which extend (*renew*) freshness periods of selected cached objects. The decision of which objects to renew upon expiration varies between policies and is guided by natural properties of the request history of each object such as time-to-live (TTL) values, popularity, and recency of previous requests. Refreshment policies can also be viewed as *prefetching freshness*. Their methodology and implementation, however, is closer to replacement policies than object prefetching algorithms. Our refreshment policies resemble common replacement policies (such as LRU and LFU) in the way objects are prioritized. First, policies prefer renewing recently- or frequently-requested objects. Second, implementation is similar since object value is determined only by the request-history of the object rather than by considering request history of related objects. Another difference of refreshment and document prefetching is that validations typically have considerably smaller response-sizes than complete documents, but due to communication overhead, the latency gap is not nearly as pronounced. Hence, refreshment potentially provides considerably better tradeoffs of bandwidth vs. reduced latency compared to object prefetching.

Our experimental study indicates that the best among the refreshment policies we have studied can eliminate about half of the freshness misses at a cost of 2 additional validation requests per eliminated freshness miss. Freshness misses themselves constitute a large fraction (30%-50%) of cache hits in a typical cache today. Therefore we conclude that one can considerably improve cache performance by incorporating a refreshment policy in it.

Overview

The paper proceeds as follows. We discuss related work in Section II. We then provide a brief overview of HTTP freshness control in Section III. In Section IV we discuss and analyze our log data. In Section V we present the different refreshment policies. In Section VI we describe our methodology for trace-based simulations and the simulation results for the performance of the different policies. We conclude in Section VII with a summary and future research directions.

II. RELATED WORK

Recent work addressed validation latency incurred on freshness misses, including transferring stale cached data from the cache to the client's browser while the data validity is being verified [13] or while the modified portion (the "delta") is being computed [14]. These schemes, however, may require browser support and are effective only when there is limited bandwidth between end-user and cache (such as with modem users).

Some or all freshness misses can be eliminated via stronger cache consistency. Cache consistency architectures include: server-driven mechanisms where clients are notified by the server when objects are modified (e.g. [15], [16]); client-driven mechanisms, where the cache validates with the server objects with a stale cached copy; and hybrid approaches where validations are initiated either at the server or the client. Hybrid approaches include the server piggybacking validations on responses to requests for related objects [17], [18], [19], which are used by the cache to update the freshness status of its content. Another hybrid approach is *leases* where the server commits to notify a cache of modification, but only for a limited pre-agreed period [20], [21], [22], [23]. Server-driven mechanisms provide strong consistency and can eliminate all freshness misses. Hybrid approaches can provide a good balance of validation overhead and reduced validation traffic. None of these mechanisms, however, is deployed or even standardized. Implementation requires protocol enhancements and software changes not only at the cache, but at all participating Web servers, and may also require Web servers to maintain per-client state. Hence, it is unlikely they become widely deployed in the near future. Meanwhile, except for some proprietary coherence mechanisms deployed for hosted or mirrored content [1], [24] (which require control of both endpoints), the only coherence mechanism currently widely deployed and supported by HTTP/1.1 is *client-driven* based on TTL (time-to-live). Our refreshment approach utilizes this mechanism.

We considered refreshment policies similar to the ones proposed here in [25] for caches of Domain Name System (DNS) servers. Each resource record (RR) in the domain name system has a TTL value initially set by its authoritative server. A cached RR becomes stale when its TTL expires. Client queries can be answered much faster when the information is available in the cache, and hence, refreshment policies which renew cached RRs offline increase the cache hit rate and decrease user-perceived latency.

Although not yet supported by some widely deployed Web caching platforms (e.g., Squid [10]), proactive refreshment is offered by some cache vendors [26], [27]. Such products allow refreshment selections to be configured manually by the administrator or integrate policies based on object popularity. Our work formalizes the issues and policies and systematically evaluates them.

III. FRESHNESS CONTROL

We provide a simplified overview of the freshness control mechanism specified by HTTP and supported by compliant caches. For further details see [28], [29], [10], [30], [31]. Caches compute for each object a *time-to-live (TTL)* value during which it is considered *fresh* and beyond which it becomes *stale*. When a request arrives for a stale object, the cache must validate it before serving it, by communication either with an entity with a fresh copy (such as another cache) or with the origin server. The cacheability and TTL computation is performed using directives and values found in the object's HTTP response headers.

When the cache receives a client request for an object then it acts as follows:

- If the object is cached and fresh, the request constitutes *content and freshness hit* (*chit* and *fhit* respectively) and the cached copy is immediately returned to the client.

• If the object is cached, but stale, the cache issues a conditional HTTP `GET` request to the origin server (or another appropriately-selected cache). The conditional `GET` uses the entity tag of the cached copy (with HTTP/1.1 and if an E-TAG value was provided with the response) or it issues an `If-Modified-Since (IMS)` request with the LAST-MODIFIED response header value (indicating last modification time of the object). If the source response is `Not-Modified` then the request constitutes a *content hit* (*chit*) but a *freshness miss* (*fmiss*). If the object was modified, the request is a *content miss* (*cmiss-r*).
• If the item is not found in the cache, then it is fetched from the origin (or another cache) and the request constitutes a *content miss* (*cmiss-d*).
• If the request arrives from the client with a `no-cache` header then the cache forwards it to the origin server. The cache must forward the request even if it has a fresh copy. The cache uses the response to replace or refresh its older copy of the object. We refer to such requests as *no-cache* requests.

The TTL calculation for a cacheable object as specified by HTTP/1.1 compares the *age* of the object with its *freshness lifetime*. If the age is smaller than the freshness lifetime the object is considered fresh and otherwise it is considered stale. The TTL is the freshness lifetime minus the age (or zero if negative).

The age of an object is the difference between the current time (according to the cache's own clock) and the timestamp specified by the object's DATE response header (which is supposed to indicate when the response was generated at the origin). If an AGE header is present, the age is taken to be the maximum of the above and what is implied by the AGE header.

Freshness lifetime calculation then proceeds as follows. First, if a MAX-AGE directive is present, the value is taken to be the freshness lifetime. Otherwise, if EXPIRES header (indicating absolute expiration time) is present, the freshness lifetime is the difference between the time specified by the EXPIRES header and the time specified by the DATE header (zero if this difference is negative). Thus, the TTL is the difference between the value of the EXPIRES header and the current time (as specified by the cache's clock). Otherwise, no explicit freshness lifetime is provided by the origin server and a heuristic is used: The freshness lifetime is assigned to be a fraction (HTTP/1.1 mentions 10% as an example) of the time difference between the timestamp at the DATE header and the time specified by the LAST-MODIFIED header, subject to a maximum allowed value (usually 24 hours, since HTTP/1.1 requires that the cache must attach a warning if heuristic expiration is used and the object's age exceeds 24 hours).

Before concluding this overview, we point on two qualitative issues with the actual use of freshness control and their relation to our refreshment approach. In [32] we analyze the distribution of different freshness control mechanisms for objects in the traces we used and it shows that the large majority of cacheable objects do not have explicit directives. For these objects, the heuristic calculation is used to determine the freshness lifetime, and thus, tradeoffs between freshness and coherence can be controlled by tuning parameter values and URL filters [30]. Our refreshment policies are applied on top of this heuristic, take it as a given, and attempt to reduce freshness misses without further compromising coherence. Another important issue suggested by recent studies is that cache control directives and response header timestamp values are often not set carefully or accurately. These practices may skew freshness calculations away from the original intent [31], [33], [32]. This issue is also orthogonal to our approach since our policies, like caches, take these settings at face value.

IV. DATA ANALYSIS

We used two 6 days NLANR cache traces [34] collected from the UC and SD caches from January 20th till January 25th, 2000. The NLANR caches run Squid [10] which logs and labels each request with several attributes such as the request time, service time, cache action taken, the response code returned to the client, and the response size.

The data analysis below considered all HTTP `GET` requests such that a 200 or 304 response codes (`OK` or `Not-Modified`) were returned to the client. We classified each of these requests as *fhit*, *fmiss*, *cmiss-r*, cmiss-d, or *no-cache* using the Squid logging labels as follows.

• Content hits:
– *fhit:* (freshness hit) the cache had a fresh cached copy. Squid labels TCP_HIT, TCP_MEM_HIT, TCP_IMS_HIT.
– *fmiss:* (freshness miss) the cache had a stale cached copy and validated it. Squid label TCP_REFRESH_HIT.
• Content misses:
– *cmiss-r:* the cache had a stale cached copy, issued an IMS request, and got a new copy with a Modified response. Squid label TCP_REFRESH_MISS.
– *cmiss-d:* there was no cached copy of the object. Squid label TCP_MISS.
• *no-cache:* the request arrived with a `no-cache` request header. Squid label TCP_CLIENT_REFRESH_MISS.

The table of Figure 2 shows the fraction of requests of each type.

Requests classified as fmisses, cmisses, or no-cache involve communication with the origin server or another cache. Freshness misses, targeted by refreshment policies, constitute 13% (UC) and 19% (SD) of all such requests. Moreover, it is evident that freshness misses constitute 31% (UC) and 43% (SD) of all content hits (requests classified as fmisses or fhits). These NLANR caches directed most validation requests (fmisses and cmisses-r) to the origin server (100% in the UC cache and 99.3% in the SD cache). It is also apparent that the vast majority (90% for UC and 95% for SD) of validation requests return `Not-Modified` (are classified as fmisses).

The NLANR traces also recorded the service time of each request. That is, the time from when the HTTP request is received to when the last byte of the response is written to the client's socket. Note that this is usually one Round Trip Time (RTT) less than from the client's viewpoint. Figure 3 plots the Cumulative Distribution Function (CDF) of service time of requests, broken down by the cache action. The gap between freshness misses to freshness hits indicates the potential benefit, in terms of latency, of eliminating fmisses. The gap between freshness misses and content misses is in part due to the additional time required to transfer a larger-size response. Another explaining factor is that content misses exhibit less locality of reference in

trace	fhits	fmisses	cmisses-d	cmisses-r	no-cache
UC	23%	10%	56%	1%	10%
SD	19%	15%	56%	3%	7%

Fig. 2. Classification of the requests in the UC and SD traces.

the sense that the elapsed time since the preceding request to the server is more likely to be longer. The decreased locality implies longer DNS resolutions of the server's hostname, since the required DNS records are less likely to be cached, and longer response time for the HTTP request, since the origin server is more likely to be "cold" with respect to the cache[1]. The similar service time distribution for no-cache and freshness misses suggests that most no-cache requests are made to popular cached content.

Figure 4 plots the CDF of service times on freshness misses and freshness hits, further broken down by the response code that the cache returned to the client. HTTP response code 200 indicates that content was returned whereas response code 304 (`Not-Modified`) indicates that the client issued an `IMS GET` request and that its copy was validated by the cache. Responses with code 304 are typically smaller-size than responses with code 200. We can see that freshness hits with a 304 response to the client had minimal service time whereas freshness hits with 200 responses reflect RTTs between the cache and its clients and additional processing. The gap for freshness misses between 200 and 304 responses is also similar and reflects the additional communication between cache and clients due to the larger transmitted response size.

V. Refreshment Policies

Refreshment policies associate with every cached object a *renewal credit* (a nonnegative integer). When a cached copy is about to expire (according to its respective TTL interval), and it has nonzero renewal credit, a renewal request is sent to the respective authoritative server, and the renewal credit is decremented.

The association of renewal credits to objects is governed by the particular policy. The policies we consider may increment the renewal credit only upon a client request. Renewal scheduling can be implemented via a priority queue, grouping together objects with close-together expiration, or possibly by an event-triggered process.

We discuss two types of renewal requests:

1. **Conditional fetch:** The cache notifies the server of last modification times or entity tag(s) of cached version(s) of the object, and requests either a validation of its current version or a new valid version. (This is supported by HTTP/1.0 by an `If-Modified-Since GET` request)
2. **Pure validation request:** Test whether the cached version of the object is valid, without requesting a valid copy if it is no longer valid. (Under HTTP/1.0 this is performed by issuing an `If-Modified-Since HEAD` request. HTTP/1.1 provides additional mechanisms, e.g., range requests.) Policies that use pure validation requests stop renewing a copy once it is invalidated, even if its renewal credit is positive.

Pure validation renewals generally use less bandwidth than conditional fetches, since if the object was modified, only the object header is transmitted. On the other hand, conditional fetches result in a fresh cached copy even when the object was modified. Thus, policies that only perform pure validation renewals target only freshness misses whereas policies that allow conditional fetches also address some content misses (cmisses-r). For small-size objects that can fit on a single packet, however, the overhead of pure validations is similar to that of conditional fetches so conditional fetches would be more effective. Ultimately, the type of renewal request can be determined by the policy on a per-request or per-object basis, according to (previous) content length and modification patterns. The data analysis in Section IV shows that only a small fraction (5%-10%) of `IMS` requests result in invalidations and content transmission. This suggests that the additional overhead of performing conditional fetches over pure validations is typically amortized over 10-20 renewals. Therefore, for moderate size objects, the total bandwidth usage is similar under both choices of renewal actions. For the sake of simplicity, we evaluated only policies with pure validation renewals. Since the likelihood of modification is low, pure validation renewals capture most of the potential of refreshments and provide a good indication for the full potential. We believe, however, that the incorporation of conditional fetches in the policies deserves further study.

Another design issue that may require explaining is that we chose not to consider predictive policies which renew or prefetch long-expired objects. Predictive renewals and document prefetching are typically effective if activity is traced at a per-user basis, where future requests are predicted according to current requests made by the same user to related objects. Our refreshment approach differs from predictive-renewal in that we consider the aggregate behavior across users on each object. Our policies use minimal book-keeping, simple implementation, and do not require Web server support. Ultimately, it may be effective for refreshment policies to co-exist with predictive renewals and content prefetching, but we believe that the basic differences between these techniques call for separate initial evaluations.

One last important point that we would like to make explicit is that our policies were designed (and evaluated) for caches that forward requests to origin servers. For example, top-level caches in a hierarchy and caches that are not attached to a hierarchy. Directing renewal requests to an authoritative source assures a maximum freshness time for the response. Caches that are configured to forward requests to a parent cache may also benefit from deploying a refreshment policy. The potential gain, however, is limited since renewals would often obtain aged responses (i.e. objects that have already been cached for a time period at the higher level cache). We studied age effects on

[1] Our study in [35] indicates that a first HTTP request to a server in a time period is more likely to take longer than subsequent ones.

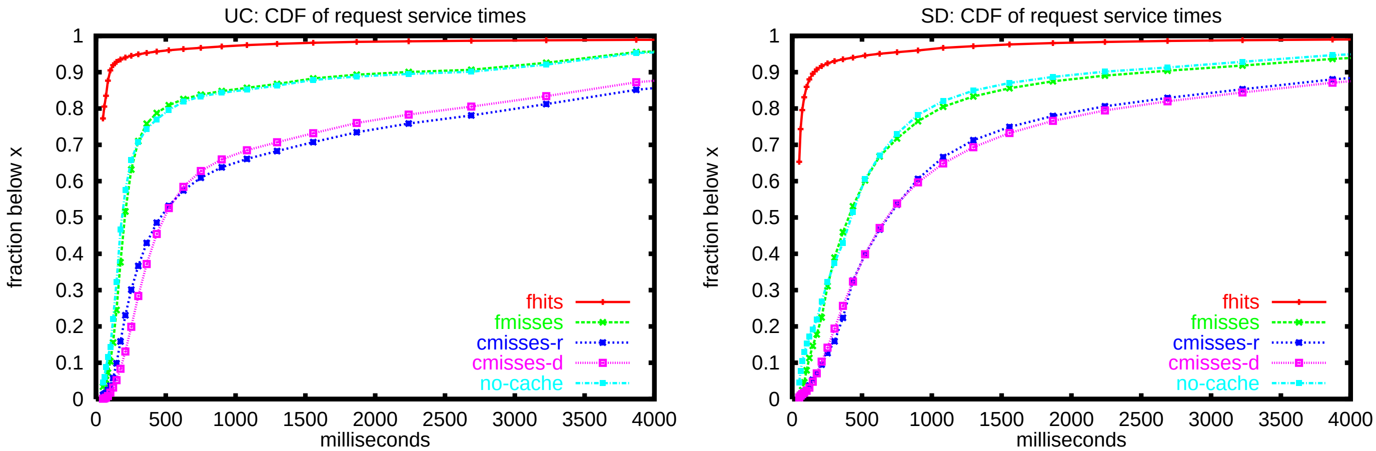

Fig. 3. CDF of the service times, broken down by cache action

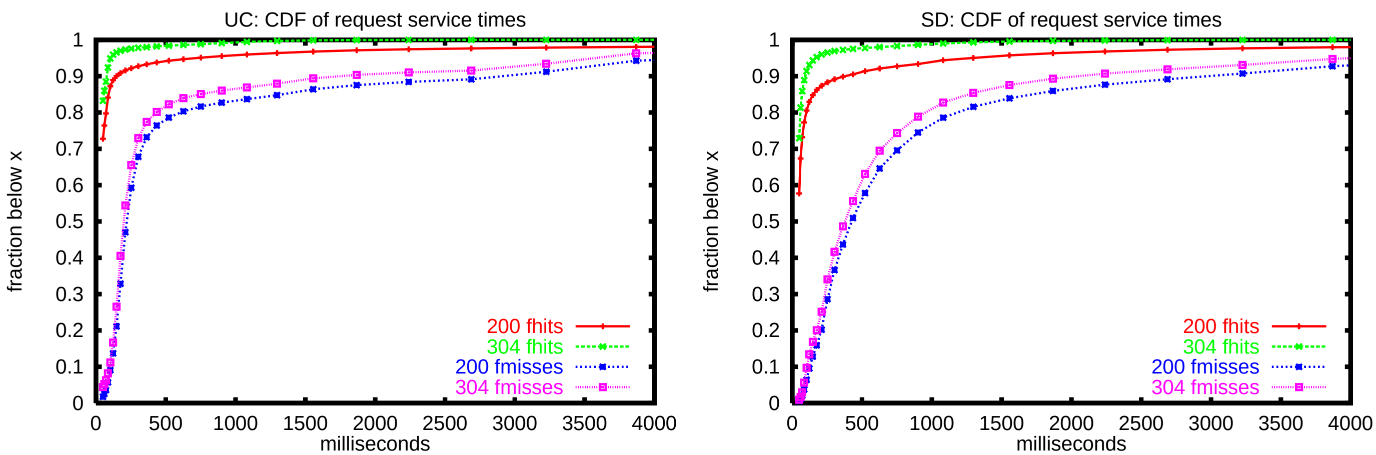

Fig. 4. CDF of the service times for content-hits, broken down by response code to the client

performance of cascaded caches in [36].

We proceed by describing the different policies. An illustrated example is provided in Figure 5. In the next section we evaluate and compare their performance using trace-based simulations.

Policies

- PASSIVE: passive validation, objects are validated only as a result of a freshness miss i.e. when a request arrives and there is a stale cached copy. This is the way most caches work today.
- OPT(i): An approximation to the optimal omniscient offline policy.[2] This policy assumes knowledge of the time of the subsequent request for the object and whether the object copy would still be valid then. If the subsequent request is such that the current copy remains valid and it is issued within $c \leq i$ freshness-lifetime-durations after the expiration of the current copy, then the renewal credit is set to c. Otherwise, no renewals are performed.
- RECENCY(k): The renewal credit is reset to k following any request for the object, including no-cache requests. RECENCY(k), similarly to the cache replacement policy LRU, exploits the *recency property* of request sequences, which states that future requests are more likely to be issued to recently-requested objects.
- RECENCY$^*(k)$: A variant of RECENCY(k) that resets the renewal credit to be k following any request for the object, *except for* no-cache requests.
- FREQ(j, m): increment the renewal credit by j for any request that would have been a freshness miss under PASSIVE. In other words, we add j to the renewal credit upon any request which is issued more than freshness-lifetime-duration units of time after a previous request that caused the passive policy to contact the origin server. In addition, upon any request (except for no-cache requests) the renewal credit is set to m if it is less than m. With $m = 0$, this policy FREQ$(j, 0)$ purely exploits the *frequency property* that states that objects that were frequently requested in the past are more likely to be requested soon. (A cache replacement policy that exploits the frequency property is LFU.) For $m > 1$, the policy FREQ(j, m) is a hybrid of FREQ$(j, 0)$ and RECENCY(m).
- TH-FREQ(th, m) keep renewing objects until the ratio of would-have been PASSIVE freshness misses to number of

[2] We specify optimal offline algorithm in [25].

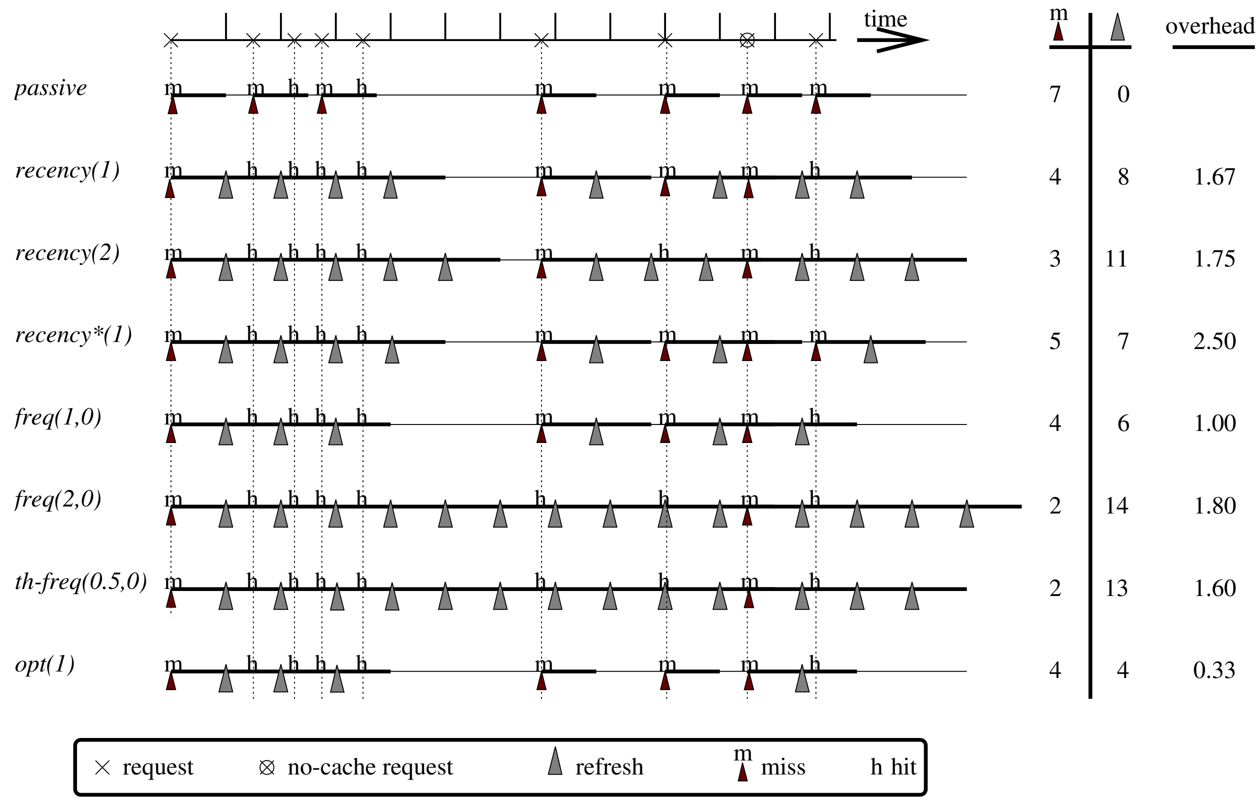

Fig. 5. Behavior of different refreshment policies on an example sequence of 9 requests. The time line is in units of freshness-lifetime durations. All policies incur at least two misses: The first request is a cold-start miss and the 8th request, which has a no-cache request header, is always a miss. The figure also summarizes the number of misses and renewals performed by each policy. PASSIVE, for example, incurs 7 misses and performs no renewals, while TH-FREQ$(0.5, 0)$ incurs only 2 misses and performs 13 renewals. The policy OPT(1) is the most efficient in the following sense: it performs the least renewals among all policies that incur at most 4 misses (e.g., RECENCY(1) and FREQ$(1, 0)$). This example illustrates how the *coverage* (fraction of misses eliminated) of the policies RECENCY(i) and FREQ(i) increases with i. The *overhead* (number of "extra" requests per eliminated miss), however, typically (but not always) increases as well. For example, RECENCY(1) eliminates 3 misses with respect to PASSIVE but performs 8 renewals. Thus, it issues $8 + 4 - 7 = 5$ more requests (renewals and misses) than PASSIVE and has overhead of $5/3 \approx 1.67$. Similarly, RECENCY(2) eliminates 4 misses while issuing $3 + 11 - 7 = 7$ more requests than PASSIVE, and thus, has overhead of $7/4 = 1.75$.

freshness-lifetime-durations since beginning of log drops below a threshold. In other words, upon each request which would have been a freshness miss for PASSIVE we increment the renewal credit such that we would keep renewing until the ratio drops below a threshold. In addition, upon any request (except for no-cache requests) the renewal credit is increased to m if it was less than m. This policy exploits the frequency property, and normalizes it by freshness-lifetime-duration. It also naturally provides a more continuous range of tradeoffs, since th is not necessarily an integer. With $m = 0$ the policy is purely frequency-based whereas higher values of m correspond to hybrids with RECENCY(m) policy.

VI. Experimental Evaluation

We conducted trace-based simulations in order to evaluate cache performance under the different refreshment policies. We outline our methodology and then proceed to present and discuss the simulation results.

A. Methodology

The traces included the cache action taken on each request for the currently-implemented passive refreshment. In order to simulate other policies, however, we had to obtain response header values or TTL values for requested objects. Unfortunately, this data is not available in the recorded trace. We therefore separately issued `GET` requests for the URLs in the trace shortly after downloading it. We processed the response headers and extracted cache directives and values of relevant header fields. For cacheable objects (objects without a `no-cache` directive in the response header), we applied the Squid object freshness model [10] (HTTP/1.1 compliant) described in Section III to calculate TTLs using the values extracted above. When an explicit freshness-lifetime duration was not provided by an EXPIRES or MAX-AGE header, we applied Squid's heuristic. We used 10% of the time difference between the time specified by the DATE header and the time specified by the LAST-MODIFIED header subject to a maximum value of one day.

We issued more than a single `GET` for a sample of the objects and repeated the TTL calculation. We found that cache control directives and freshness-lifetime values do not change frequently. This indicated that our estimates reflected reasonably well values that would have been obtained from the origin server at the time requests were logged.

We then ran simulations using the original sequence of requests and the extracted TTL values. To put all policies on equal ground and eliminate the effect of boundary conditions we also simulated PASSIVE, so the performance figures provided later correspond to the simulated PASSIVE rather than the one reflected by the original labels (given in Section IV).

We first discarded all requests that were not labeled as HTTP GET. The simulation was only applied to requests for URLs on which we obtained a 200 (OK) response on our separately-issued requests. Note that 302 responses (HTTP redirect) are not cacheable and hence requests of URLs for which we obtained such a response were discarded. The simulation then utilized the logged information in the following way: (i) All logged requests for each considered URL were used along with the request time to determine the status of the object in the simulated cache and the resulting cache actions, (ii) the original cache-action label was used to identify requests which arrived with a no-cache header (our simulation accounted for such requests by resetting the TTL to the freshness-lifetime duration even if a fresh copy was present in the simulated cache), and (iii) the original labels were also used in a heuristic that estimated at which points objects were modified.

The modification heuristic considered various recorded labels of the requests. Clearly, when a successful request was classified by the labels in the trace as a *cmiss-r* (TCP_REFRESH_MISS, see Section IV) then contents had changed. For requests labeled *cmisses-d* we did not have an explicit indication whether content had changed so we used a heuristic based on the logged size of the response to the client. In order to simulate performance of refreshment policies, we also had to estimate at which point in the interval between requests the modification had actually occurred (since refreshment policies stop refreshing once the server invalidates the object). The simulation assumed that when a modification had occurred between two consecutive requests (and therefore incur a content miss on the later request), it happened at the midpoint of the time interval between the two requests. If more modifications had happened, it is likely that the first one occurred earlier in the interval, and hence our assumption means that more unproductive renewals occur in the simulation than in reality, and thus, the simulated policies would exhibit somewhat worse tradeoffs. Since the majority of validations return not-modified, however, this assumption could not have had a significant effect on our results.

Since we could not issue GET requests for all the URLs present in the trace in a reasonable time without adversely affecting our environment, we selected a subset. We used all URLs that were requested more than 12 times, and a weighted (by number of requests) random sample of the others. In total we fetched about 224K distinct URLs[3]. Our results were scaled up such that the effect of the sampling is factored out.

The simulations assumed infinite cache storage capacity. This is consistent both with current industry trends and with the actual traces we used, since objects requested twice or more in the 6 day period were not likely to be discarded by the LRU replacement policy used in Squid.

In the performance metric used in the simulations we counted all requests that constituted *content hits*. Content hits that occurred more than a freshness-lifetime duration past the previous (simulated) server contact were counted as *freshness misses* and requests occurring within the duration were counted as *freshness hits*. Content hits exclude requests to explicit noncacheable objects, requests with no-cache request headers, and requests when the content had changed. Since we had no information on such requests, we did not classify the first request of each object. Appropriate requests for objects with freshness-lifetime-duration of 0 were counted as content hits, but were all considered freshness misses. Note that renewals are not performed on such objects and hence, the number of freshness misses incurred on these objects is not reduced by refreshment policies.

B. Simulation results

Under the simulated baseline policy PASSIVE (where objects are refreshed only as a result of client requests), 48% of content hits constituted freshness misses on the UC trace, and 53% were freshness misses on the SD trace. We recall that the respective numbers according to labels on the recorded trace provided in Section IV are 31% and 43%. The gap is mainly due to simulating a shorter trace, different boundary conditions, and the conservative heuristic used to identify content hits. These factors should affect all policies in a similar manner. So in order to put PASSIVE on equal grounds with all other policies we chose to simulate it rather than using the labels of the trace.

We evaluated the performance of the different refreshment policies by the tradeoff of *overhead* vs. *coverage*. The coverage (reduction in freshness-misses) is calculated as the fraction of the number of freshness misses according to passive that are eliminated by the respective policy. More precisely, if x denotes the number of freshness misses of PASSIVE, and y the number of freshness misses of a policy P then the coverage of P is the fraction $(x-y)/x$.

We calculated the *overhead* of policy P as the difference between the number of validation requests issued by P and the number of validation requests issued by PASSIVE. Recall that the cache issues a validation request for each freshness miss. Hence, the request overhead is the total number of renewals performed minus the number of freshness misses eliminated (converted to freshness hits). We normalize the overhead by dividing it with the total number of freshness misses that were converted to freshness hits.

To obtain the coverage/overhead tradeoff for each type of policy, we swept the value of its respective parameter. For example, the points on the curve of RECENCY correspond to runs with RECENCY(1), RECENCY(2), RECENCY$(3), \ldots$ and the points for FREQ were obtained on runs with FREQ$(1,0)$, FREQ$(2,0), \ldots$. Note that OPT(0), RECENCY(0), RECENCY$^*(0)$, FREQ$(0,0)$, and TH-FREQ$(\infty,0)$ are in fact PASSIVE.[4] These tradeoffs are shown in Figure 6.

The performance of RECENCY and RECENCY* policies was almost identical, hence we omitted the RECENCY* curve. This similarity shows that requests without a no-cache header are about as likely to follow a request with a no-cache header as one without a no-cache header.

[3]The original logs had about 5 million different URLs, most of them requested only once. However, most of the requests are for frequently requested URLs, so reducing the miss rate of those is the goal.

[4]We remark that these policies are able to achieve "continuous" tradeoffs by mixing two consecutive integral values of the respective parameter, for example, applying RECENCY(1) on some URLs and RECENCY(2) on others.

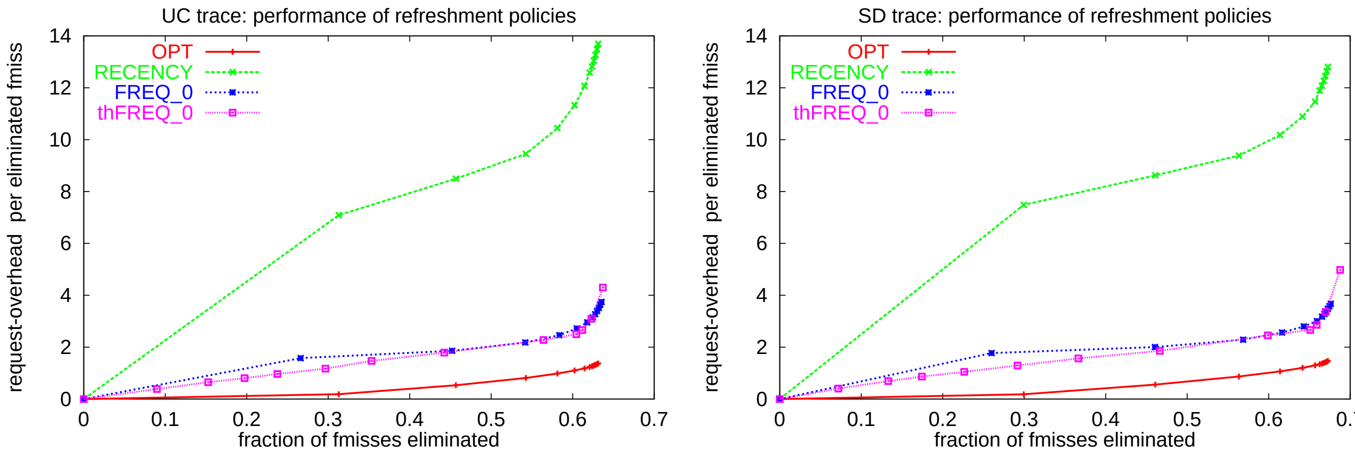

Fig. 6. Performance of the different refreshment policies when simulated on the UC and SD traces

Under all types of policies, the coverage peaks at about 63%-67%. The remaining 33%-37% of freshness misses mostly occur on objects with freshness-lifetime-duration of 0, on which refreshment is not effective. The OPT policies eliminates all "addressable" freshness misses with overhead of about 1.3 requests per miss, and eliminates the bulk of these misses with an overhead of 0.5. These numbers bound the potential of refreshment policies. They also indicate on the performance loss by the restriction to the refreshment framework.[5]

The simulation results for OPT and RECENCY show the locality effect, where most freshness misses that can be eliminated occur within a small number of freshness-lifetime-durations after a previous request. The left-most point on the curves of RECENCY and OPT, which correspond to RECENCY(1) and OPT(1), show that about 30% of freshness misses occur within one freshness-lifetime-duration after the expiration of the cached copy. The second points on the left correspond to RECENCY(2) and OPT(2) and indicate that about additional 15% of freshness misses occur between one and two freshness-lifetime durations passed the expiration. We note that the observed fact that a very small number of freshness misses occur more than 10 freshness-lifetime-durations passed the expiration is not only due to locality but also reflects the interaction of the log duration of 6 days and the most common freshness-lifetime-duration of 24 hours [32].

The fact that coverage of RECENCY and the frequency-based policies peaks at about the same place indicates that a very small fraction of freshness misses are incurred on very infrequently requested objects (since the frequency-based policies do not perform any renewals on the first request and thus can not eliminate misses incurred on the second request.). The correspondence in peak coverage of OPT and other policy-types is due to a "threshold phenomenon" where most freshness misses occur on objects with a freshness-lifetime-duration of 0 or occur within a small number of freshness-lifetime-durations following a previous request.

The frequency-based policies FREQ$(j, 0)$ and TH-FREQ$(th, 0)$ significantly outperformed RECENCY(k). This reflects the fact that the vast majority of freshness misses which can be eliminated occur on the more popular URLs. The gap is caused by the very large number of cacheable URLs that were requested only once. The RECENCY(k) policy performed up to k unproductive renewals[6] on each such request. The hybrid policies FREQ and TH-FREQ with $m > 0$ performed considerably worse than the pure frequency-based policies (that correspond to $m = 0$) and hence only results for $m = 0$ are shown. This behavior is not surprising given that RECENCY yielded much worse tradeoffs. Overall, the results indicate that frequency-based object prioritization is more effective than recency-based prioritization.

The domination of frequency-based policies is also consistent with studies of cache replacement policies for Web contents [7], [6], since different URLs tend to have a wide range of characteristic popularities, a property that is captured better by frequency-based policies. It is interesting to contrast these results with a related study of refreshment policies that we performed for DNS records. In contrast with our findings here, for DNS caches the recency and frequency-based policies exhibited similar performance [25]. Our explanation is that at the hostname level, there is significantly smaller fraction of "objects" that are resolved only once.

The performance of FREQ and TH-FREQ is similar, although TH-FREQ, which normalizes the frequency by freshness-lifetime-duration, performed somewhat better. The similarity is mostly explained by the fact that the large majority of freshness-lifetime-durations are of the same length (one day) and also because for shorter durations, frequency of requests is correlated with the duration. The policy TH-FREQ provides a spectrum of tradeoffs and better performance, particularly in the low-overhead range. TH-FREQ, however, may require more bookkeeping than FREQ. The particular tradeoffs obtained by the frequency-based policies shows that significant improvements can be obtained with fairly low overhead. About 10% of freshness misses can be eliminated with the overhead of half of a validation request per eliminated freshness miss; 25% of freshness

[5] Recall that under the refreshment framework, objects must be kept fresh continuously till the following "hit." The optimal *unrestricted* policy, which validates objects just before they are requested, incurs no overhead.

[6] It could be less than k since we did not perform renewals passed the termination time of the log.

misses can be eliminated with overhead of a single request per eliminated miss; 50% can be eliminated with overhead of two; and 65% of freshness misses can be eliminated with overhead of three.

VII. Conclusion

A large fraction (30%-50%) of cache hits constitute freshness misses, that is, the cached copy was not fresh, but turned out to be valid after communication with the origin server. Validations are performed prior to sending responses to users, and significantly extend cache service time. Therefore, freshness misses impede the cache ability to speed-up Web access.

An emerging challenge for Web content caches is to reduce the number of freshness misses by proactively maintaining fresher content. It seems that some cache vendors had already implemented ad-hoc refreshment mechanisms. Our proposed refreshment policies are a relatively low-overhead systematic solution. Refreshment policies extend freshness lifetime by selectively validating cached objects upon their expiration. Since cache freshness is increased, requested objects are more likely to be fresh and thereby are serviced faster. We demonstrated that a good refreshment policy can eliminate about 25% of freshness misses with an overhead of a single validation requests per eliminated miss, that is, two "offline" validation requests replace one "online" request.

For future work, we propose ways to further reduce renewal overhead. We first propose that renewals of objects located at the same host are batched together, and thus, decrease overhead by sharing the same persistent connection. Batching can be natural as co-located objects often share the same cache-control mechanism and subsets of such objects (that are embedded on the same page or related pages) are often requested together. A second proposal is to perform renewals at off-peak hours. The most common freshness-lifetime duration of 24 hours provides sufficient scheduling flexibility to do so (e.g., performing renewals due at 10am EST at the significantly less-busy time of 7am EST).

As a next step in the evaluation of the effectiveness of a refreshment policy we hope to incorporate one such policy in a popular caching server software such as Squid. Our results indicate that the integration of refreshment would not impose a significant computational overhead, and would boost performance in terms of user-perceived latency.

Acknowledgment. Our experiments would not have been possible without the collection and timely availability of the NLANR cache traces. We thank Duane Wessels for answering questions with regard to a Squid logging bug.

References

[1] "Akamai." `http://www.akamai.com`.

[2] S. Williams, M. Abrams, C. R. Standbridge, G. Abdulla, and E. A. Fox, "Removal policies in network caches for world-wide web documents," in *Proceedings of the ACM SIGCOMM Conference*, pp. 293–305, August 1996.

[3] P. Cao and S. Irani, "Cost-aware www proxy caching algorithms," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), December 1997.

[4] A. Bestavros *et al.*, "Application-level document caching in the internet," in *Proceedings of the Second Intl. Workshop on Services in Distributed and Networked Environments (SDNE '95)*, 1995.

[5] E. Cohen, B. Krishnamurthy, and J. Rexford, "Evaluating server-assisted cache replacement in the Web," in *Proceedings of the 6th European Symposium on Algorithms*, pp. 307–319, Springer-Verlag, Lecture Notes in Computer Science Vol. 1461, Aug. 1998.

[6] E. Cohen and H. Kaplan, "Exploiting regularities in Web traffic patterns for cache replacement," in *Proc. 31st Annual ACM Symposium on Theory of Computing*, ACM, 1999.

[7] L. Breslau, P. Cao, L. Fan, G. Phillips, and S. Shenker, "Web caching and zipf-like distributions: Evidence and implications," in *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.

[8] A. Feldmann, R. Cáceres, F. Douglis, G. Glass, and M. Rabinovich, "Performance of Web proxy caching in heterogeneous bandwidth environments," in *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.

[9] N. Young, "On line file caching," in *Proc. 9th ACM-SIAM Symposium on Discrete Algorithms*, ACM-SIAM, 1998.

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

[11] 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*, pp. 13–22, Dec. 1997.

[12] D. D., "Prefetching hyperlinks," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[13] A. Dingle and T. Partl, "Web cache coherence," in *Proceedings of the Fifth International World Wide Web Conference*, May 1996.

[14] G. Banga, F. Douglis, and M. Rabinovich, "Optimistic deltas for WWW latency reduction," in *Proceedings of the USENIX Annual Technical Conference*, USENIX Association, 1997.

[15] J. Howard, M. Kazar, S. Menees, D. Nichols, S. M., R. Sidebotham, and M. West, "Scale and performance in a distributed file system," *ACM Transactions on Computer Systems*, vol. 6, no. 1, pp. 51–81, 1988.

[16] D. Li and D. R. Cheriton, "Scalable web caching of frequently updated objects using reliable multicast," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, 1999.

[17] B. Krishnamurthy and C. E. Wills, "Study of piggyback cache validation for proxy caches in the world wide web," in *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, (Monterey, California), December 1997.

[18] E. Cohen, B. Krishnamurthy, and J. Rexford, "Improving end-to-end performance of the Web using server volumes and proxy filters," in *Proceedings of the ACM SIGCOMM'98 Conference*, Sept. 1998.

[19] E. Cohen, B. Krishnamurthy, and J. Rexford, "Efficient algorithms for predicting requests to Web servers," in *Proceedings of the IEEE INFOCOM'99 Conference*, 1999.

[20] C. Gray and D. Cheriton, "Leases: an efficient fault tolerant mecahnism for distributed file cache consistency," in *Proceedings of the 12th ACM symposium on operating systems principles*, pp. 202–210, 1989.

[21] C. Liu and P. Cao, "Maintaining strong cache consistency in the world-wide web," in *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[22] J. Yin, L. Alvisi, M. Dahlin, and C. Lin, "Volume leases for consistency in large-scale systems," *IEEE transactions on knowledge and data engineering*, 1999.

[23] V. Duvvuri, P. Shenoy, and R. Tewari, "Adaptive leases: a strong consistency mechanism for the World Wide Web," in *Proceedings of the 19th IEEE INFOCOM Conference*, 2000.

[24] "Inktomi Content Delivery Suite." `http://www.inktomi.com`.

[25] E. Cohen and H. Kaplan, "Proactive caching of DNS records: addressing a performance bottleneck," in *Proceedings of the Symposium on Applications and the Internet*, IEEE, 2001.

[26] "IBM WebSphere Cache Manager." `http://www.software.ibm.com/webservers/cacheman`.

[27] "Lucent IPWorX." `http://www.lucentipworx.com`.

[28] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, and T. Leach, P. Berners-Lee, "Hypertext Transfer Protocol — HTTP/1.1," RFC 2616, ISI, June 1999.

[29] T. Berners-Lee, R. Fielding, and H. Frystyk, "Hypertext Transfer Protocol — HTTP/1.0," RFC 1945, MIT/LCS, May 1996.

[30] M. Nottingham, "Optimizing object freshness controls in Web caches," in *The 4th International Web Caching Workshop*, 1999.

[31] J. C. Mogul, "Errors in timestamp-based HTTP header values," Tech. Rep. 99/3, Compaq Western Research Lab, December 1999.

[32] E. Cohen and H. Kaplan, "The age penalty and its effect on cache performance." Submitted, 2000.

[33] B. Krishnamurthy and M. Arlitt, "PRO-COW: protocol compliance on the web," Tech. Rep. 990803-05-TM, AT&T Labs-Research, 1999.

[34] "A Distributed Testbed for National Information Provisioning." `http://www.ircache.net`.

[35] E. Cohen and H. Kaplan, "Prefetching the means for document transfer: A new approach for reducing web latency," in *Proceedings of the IEEE INFOCOM'00 Conference*, 2000.

[36] E. Cohen, E. Halperin, and H. Kaplan, "Performance aspects of distributed caches using TTL-based consistency." Manuscript, 2000.