Study of Piggyback Cache Validation for Proxy Caches in the World Wide Web

Balachander Krishnamurthy

AT&T Labs–Research
180 Park Ave
Florham Park, NJ 07932 USA
bala@research.att.com

Craig E. Wills

Worcester Polytechnic Institute
100 Institute Road
Worcester, MA 01609 USA
cew@cs.wpi.edu

Abstract

This paper presents work on piggyback cache validation (PCV), which addresses the problem of maintaining cache coherency for proxy caches. The novel aspect of our approach is to capitalize on requests sent from the proxy cache to the server to improve coherency. In the simplest case, whenever a proxy cache has a reason to communicate with a server it piggybacks a list of cached, but potentially stale, resources from that server for validation.

Trace-driven simulation of this mechanism on two large, independent data sets shows that PCV both provides stronger cache coherency and reduces the request traffic in comparison to the time-to-live (TTL) based techniques currently used. Specifically, in comparison to the best TTL-based policy, the best PCV-based policy reduces the number of request messages from a proxy cache to a server by 16-17% and the average cost (considering response latency, request messages and bandwidth) by 6-8%. Moreover, the best PCV policy reduces the staleness ratio by 57-65% in comparison to the best TTL-based policy. Additionally, the PCV policies can easily be implemented within the HTTP 1.1 protocol.

1 Introduction

A proxy cache acts as an intermediary between potentially hundreds of clients and remote web servers by funneling requests from clients to various servers. In the process, the proxy caches frequently requested resources to avoid contacting the server repeatedly for the same resource if it knows, or heuristically decides, that the information on the page has not changed on the server.

A problem with caching resources at a proxy or within a browser cache, is the issue of cache coherency—how does the proxy know that the cached resource is still current [6]? If the server knows how long a resource is valid (e.g., a newspaper generated at 5:00am daily), the server can provide a precise expiration time. Cached copies are always fresh until the expiration time. More commonly, the resource that is made available has no clear expiration time. It may change in five minutes or remain unchanged for a long time.

Our work addresses the problem of maintaining cache coherency for proxy caches. The novel aspect of our approach is using requests sent from the proxy cache to the server to obtain additional coherency information. In the simplest approach, whenever a proxy cache has a reason to communicate with a server it piggybacks a list of cached, but potentially stale, resources from that server for which the expiration time is unknown. Compared to other techniques, this piggyback cache validation (PCV) approach has the potential of both ensuring stronger cache coherency and reducing costs.

The paper is organized as follows: Section 2 discusses related work in the area of cache coherency in the Web context. Section 3 describes piggyback cache validation and presents two possible implementations on top of the Hypertext Transport Protocol (HTTP). Section 4 presents a study of two variants of PCV and contrasts them with four other cache coherency approaches. The study is based on trace-driven simulation of two large logs using evaluation criteria discussed in Section 5. Results from these simulations are presented in Section 6. Section 7 summarizes the work with a discussion of ongoing work and future directions.

2 Related Work

Caching and the problem of cache coherency on the World Wide Web are similar to the problems of caching in distributed file systems [10, 18]. However, as pointed out in [8], the Web is different than a distributed file system in its access patterns, its larger scale, and its single point of updates for Web objects.

Cache coherency schemes providing two types of consistency have been proposed and investigated for caches on the World Wide Web. One type—strong cache consistency, is maintained via one of two approaches. In the first approach, *client validation*, the proxy treats cached resources as potentially out-of-date on each access and sends a If-Modified-Since header with each access of the resource. This approach provides strong cache consistency, but can lead to many 304 responses (HTTP response code for "Not Modified") by the server if the resource does not actually change. The second approach is *server invalidation*, where upon detecting a resource change, the server sends invalidation messages to all clients that have recently accessed and potentially cached the resource [14]. This approach requires a server to keep track of lists of clients to use for invalidating cached copies of changed resources and can become unwieldy for a server when the number of clients is large. In addition, the lists can become out-of-date causing the server to send invalidation messages to clients who are no longer caching the resource.

In contrast, weak consistency approaches seek to minimize the proxy validation and server invalidation messages by using a heuristic or a pre-defined value as an artificial expiration time on a cached resource. One such approach, based on the last modified time [8], is for the proxy to adopt an adaptive time-to-live (TTL) expiration time (also called the Alex protocol [4]). The older the resource, the longer the time period between validations. This adaptive TTL heuristic is reasonable, but there can be periods where the cached resource is potentially stale. Related work in the WebExpress project for mobile environments allows users to set a fixed coherency interval for objects with the capability to change this interval for specific objects [9].

In terms of prior work using piggybacking to improve cache coherency, Mogul [15] has proposed piggybacking server invalidations for cached resources as part of replies to client requests. This idea was a motivation for our work on piggyback cache validation, but not directly investigated in the results reported here.

The concept of piggybacking additional information to a Web request or reply has been proposed in limited forms for other uses. Previous work suggests that server knowledge about access patterns for a requested resource could be returned along with the resource. This knowledge could be used by the client to control prefetching [2, 19]. [22] extends this approach by using both client and server knowledge. Mogul proposes "hit-metering" as a technique for a cache to report reference count information back to an origin server [16]. This information can be used in predicting reference hit information, which can be passed as hints to a cache whenever the server sends a response to a cache. Finally, the WebExpress [9] work proposes a batch approach to perform a single check at the beginning of a session for all cached objects older than the coherency interval. This batching is related to our idea of a validation list and could be carried out using a piggybacked approach.

3 Piggyback Cache Validation

Our approach for maintaining cache coherency while reducing the number of messages exchanged with a server is to piggyback cache state information onto HTTP requests to the server [12]. While individual browser clients could use our approach, it is most beneficial to proxy caches because the number of resources cached from a particular server is small and likely short-lived in an individual browser cache. Thus, we focus on the use of the piggyback mechanism for a proxy cache.

In the simplest approach, whenever a proxy cache has a reason to communicate with a server it piggybacks a list of cached resources from that server, for which the expiration time is unknown and the heuristically-determined TTL has expired. The server handles the request and indicates which cached resources on the list are now stale allowing the proxy to update its cache.

The proxy treats client requests for cached resources with a validated age of less than the time to live threshold as current. Requests for cached resources that have not recently been validated cause an If-Modified-Since (IMS) Get request to be sent to the server.

The performance of piggyback cache validation depends on the number of resources cached at a proxy for a particular server and the number of requests that are sent by the proxy to the server. If there are few such requests, meaning the cache contents do not get validated with piggybacking, then the approach performs similar to TTL-based policies in generating a validation check when the time-to-live expires. However, if there is traffic between the proxy and server, then the cache contents are validated at the granularity of the expiration time threshold without need for IMS requests. Results from prior studies indicate such traffic exists and best case results have yielded a 30-50% proxy cache hit rate [1]. We have found a similar hit rate in our studies. The introduction of piggyback validation allows relatively short expiration times to be used resulting in close to strong cache coherency while reducing the number of IMS requests sent to a server in comparison to existing TTL-based policies.

The added cost of our mechanism is mainly in the increased size of the regular request messages due to piggybacking. However, there are no new connections made between proxies and servers. The number of piggybacked validations appended to a single request can be controlled by the proxy cache. The cost for the proxy cache is also slightly increased as it must maintain a list of cached resources on a per server basis. The additional cost for the server is that it must validate the piggybacked resources in addition to processing the regular request. However, in the absence of piggybacking, such validations may have to be done in the future by the server in separate connections.

Implementation of piggyback cache validation can be done independent of a particular cache replacement policy [3, 21]. In our initial work we have used a standard LRU cache replacement policy. However, validation information provided by a server could be used by such a replacement policy. For example, if a proxy cache finds that a cached resource is frequently invalidated then this resource would be a good candidate for cache replacement.

Two approaches could be used to implement the PCV mechanism within HTTP. The first approach is to implement the mechanism via a new HTTP header type for validation list requests and replies. In a request, the header field consists of a list of resource and last modified time pairs. On a reply, the field would contain a list of invalid resources or a value indicating that all resources in the request list are valid. This approach is compact, but it could require the server to validate the entire piggybacked list before it replies to the request itself. Alternately, invalid resources could be return as part of a footer as allowed in HTTP/1.1 [11].

Another approach is to pipeline HEAD requests trailing the resource request. The approach requires more bandwidth, but it can be implemented in HTTP 1.1 with no changes to the protocol. It also separates the request from cache validation.

In either implementation, if a server does not implement the mechanism, the proxy cache works fine, albeit without the piggybacked validation information. In our testing we assume the first approach.

4 Testing

4.1 Proxy Cache Logs

We constructed a trace-driven simulation to test our ideas and used two sets of logs:

- Digital Equipment Corporation proxy logs [5] (Sept. 16–22, 1996) with 6.4 million Get requests for an average rate of 40031 requests/hour and 387.0 MByte/hour. 57832 distinct servers were contacted with the top 1% of the servers being responsible for over 59% of the resources accessed. 45% of the servers had fewer than 10 resources accessed and over 89% of the servers had fewer than 100 resources accessed. 3.4% of the servers (1943) accounted for over half the 2083491 unique resources accessed.
- AT&T Labs–Research packet level trace [17] (Nov. 8–25, 1996) with 1.1 million Get requests for an average rate of 2805.63 requests/hour and 18.4 MByte/hour. 18005 distinct servers were contacted with the top 1% of the servers being responsible for over 55% of the resources accessed. 48% of the servers had fewer than 10 resources accessed and over 92% of the servers had fewer than 100 resources accessed. 5.6% of the servers (1019) accounted for over half the 521330 unique resources accessed.

4.2 Cache Coherency Policies

We tested six cache coherency policies with these logs:

1. pcvfix—This policy implements the basic piggyback cache idea with a fixed TTL (time to live) expiration period. By default, a cached resource is considered stale once a period of one hour has elapsed. When the expiration time is reached for this resource, a validation check is piggybacked on a subsequent request to its server. If the resource is accessed after its expiration, but before validation, then a If-Modified-Since Get request is sent to the server for this resource.

2. pcvadapt—This policy implements the basic piggyback cache idea, but with an adaptive TTL expiration time based on a fraction (adaptive threshold) of the age of the resource. As in the Alex FTP protocol [4, 8], the motivation is that newer resources change more frequently, while older resources change less often. However, because we believe piggybacked validations are relatively inexpensive, the pcvadapt policy uses a maximum expiration time equal to the fixed TTL in effect. This policy allows a relatively tight limit for all resources with an even shorter expiration for newer resources.

3. ttlfix—This policy uses a fixed TTL expiration period for all resources and does no piggybacking. If a cached resource is accessed after its expiration then a If-Modified-Since Get request is sent to the server for this resource.

4. ttladapt—This policy uses an adaptive TTL expiration time based on resource age with no piggybacking. The upper bound for a resource expiration time is fixed at one day. The pcvadapt policy has a tighter bound because piggyback validation checks are less expensive than If-Modified-Since Get requests, which are the only means of validation for the ttladapt approach. The ttladapt policy is used in the Squid Internet Object Cache [20], which allows the adaptive threshold and maximum age to be configurable.

5. alwaysvalidate—This policy generates a If-Modified-Since Get request for every access of a cached resource. The policy ensures strong coherency, but causes a request to the server for every resource access. It is used to measure other policies against.

6. nevervalidate—This policy has an infinite expiration time so that cached resources are never validated. The policy minimizes costs by always using the cached copy of a resource, but results in the most use of stale copies. It is another policy against which other policies are measured.

4.3 Parameters Used

The cache coherency policies were studied by varying four parameters: cache size, PCV size (the maximum size of a single piggyback list), the TTL value, and the adaptive threshold. To focus the presentation of results while still being able to vary all parameters, we established a base set of parameters from which we varied one parameter at a time. The base values were established based on other published work and after testing the policies under different conditions. The base parameter values (and the range studied) were:

- cache size of 1GB (range 1MB to 8GB),
- maximum PCV size of 50 (range 10 to 1000),
- TTL of one hour (range 0.5 to 24 hours), and
- adaptive threshold of 0.1 (range 0.05 to 1.0).

We did not vary the cache replacement policy for this study, but used LRU for all our tests. Variation of replacement policy and its interaction with the cache coherency policies is an area for future work.

5 Evaluation Criteria

There are three types of costs traditionally considered when evaluating the performance of Web resource retrieval:

- response latency—how long it takes to retrieve the requested resource,
- bandwidth—how many bytes must be served by the server and transmitted over the network, and
- requests—how many requests must be handled by the server and transmitted over the network.

In this context, the goal of a good cache coherency policy, when combined with a cache replacement policy, is to provide up-to-date (non-stale) resources to clients while minimizing the above costs. However, translating the simulation results for a policy on a set of data to their relative "goodness" can be done in many ways. In the following, we define and justify how we determine the cost, staleness and overall goodness metrics used in comparing different cache coherency policies.

5.1 Cost Evaluation

The typical measure for cache replacement policies is the "hit rate," or the percentage of time that a resource request can be provided from cache. Because this measure does not account for resource size, the use of byte hit rate is also common in recent literature [3, 21]. These measures can be used to derive cost savings for bandwidth and requests, but do not reflect on savings in response latency. Mogul also points out that measuring cache hits does not measure the effect of caching mechanisms on the cost of a cache miss [16].

In addition, the cache replacement policy studies have ignored the cost for cache coherency and ratio of stale resources. These studies do not distinguish between cache hits for resources that can be used directly and hits for resources that are validated with the server. Previous studies on cache coherency [8, 14] report statistics on stale cache hits along with information on network, server, and invalidation costs. However, these studies do not report these values in the context of other cache activity, specifically cache replacement.

Our approach is to use a comprehensive cost model that accounts for the combined costs of cache replacement and coherency. The model incorporates the costs for the three possible actions that can occur when a resource is requested by a client from a proxy cache:

1. Serve from cache—the resource is currently cached and returned to the client without contacting the server. We define the costs of such action to be zero.
2. Validate—the resource is currently cached and is returned to the client after the proxy validates the cached copy is current by contacting the server. This action involves a request to the server, along with a latency cost corresponding to the distance to the server and a small bandwidth cost.
3. Get—the resource is not in cache (or the cached copy is invalid). The resource is returned to the client after retrieval from the server. This action involves a request to the server, along with bandwidth and transfer latency costs corresponding to the size of the resource and the distance from the server.

Considering these three actions, we use a normalized cost model for each of the three evaluation criteria and each of the actions where $c[a, e]$ represents the cost for action a and evaluation criterion e. We let C denote the matrix representing all combinations of proxy cache actions and evaluation criteria. In our work, each $c[a, e]$ is computed using the data from our test logs. These costs are shown in Tables 1 and 2 and explained below.

For each evaluation criterion, the cost of an average Get request with full resource reply (status 200) is normalized to 1.00 based on the values in the log. In the Digital logs, the actual values are 12279 bytes of bandwidth (includes contents and headers), 3.5 seconds of latency and one request for an average retrieval. In the AT&T logs, the actual values are 8822 bytes of bandwidth, 2.5 seconds of latency and one request.

Table 1: Normalized Cost Matrix C for Digital Logs

	Evaluation Criterion (e)			
Action (a)	Response	Bandwidth	Request	Avg.
Get	1.00	1.00	1.00	1.00
Validate	0.36	0.03	1.00	0.47
In Cache	0.00	0.00	0.00	0.00

Table 2: Normalized Cost Matrix C for AT&T Logs

	Evaluation Criterion (e)			
Action (a)	Response	Bandwidth	Request	Avg.
Get	1.00	1.00	1.00	1.00
Validate	0.12	0.04	1.00	0.39
In Cache	0.00	0.00	0.00	0.00

The cost of using a resource from cache is defined to be zero. This definition focuses on the external costs for resource access, although internal networks, computers and even the proxy cache itself contribute some latency [13].

The intermediate cost of a Validate request, which returns a validation of the current cache copy (response 304), is computed relative to the cost of a full Get request. As shown in Tables 1 and 2, this action is just as expensive as a full Get request in terms of requests, of intermediate cost in terms of response latency and of little cost in terms of bandwidth.

Tables 1 and 2 show a fourth evaluation criterion, which is the average of the costs for the other three criterion. This criterion is introduced as a composite criterion that assigns equal importance to each of the standard criteria.

Given the matrix C, we can compute the total cost for a cache coherency policy p by knowing the relative proportion of each cache action a for the policy. We let $w[p, a]$ represent the proportional weight for the occurrence of action a while using policy p and let W denote the matrix of all combinations of policies and actions. The matrix W varies depending on the simulation parameters used, but for illustration, Tables 3 and 4 show W when using the base set of parameters discussed in Section 4.3.

Using the matrices C and W, the total cost $t[p, e]$

Table 3: Representative Weight Matrix W for Digital Logs

	Action (a)		
Policy (p)	Get	Validate	In Cache
pcvfix	0.5407	0.0070	0.4522
pcvadapt	0.5429	0.0079	0.4492
ttlfix	0.5457	0.0953	0.3590
ttladapt	0.5520	0.0296	0.4184
alwaysvalidate	0.5542	0.4458	0.0000
nevervalidate	0.5391	0.0000	0.4609

Table 4: Representative Weight Matrix W for AT&T Logs

	Action (a)		
Policy (p)	Get	Validate	In Cache
pcvfix	0.5183	0.0337	0.4480
pcvadapt	0.5221	0.0351	0.4428
ttlfix	0.5238	0.2188	0.2574
ttladapt	0.5279	0.1182	0.3539
alwaysvalidate	0.5308	0.4692	0.0000
nevervalidate	0.4980	0.0000	0.5020

for each policy p and evaluation criterion e is easily computed with the matrix product $T = W \cdot C$. The resulting $t[p, e]$ values are used in reporting costs for cache coherency policies in this paper.

In analyzing Tables 3 and 4, the ratio of resources causing a full Get request is relatively constant for all cache coherence policies across the two logs. This figure is primarily dependent on the performance of the cache replacement policy. As defined, the alwaysvalidate policy never serves a resource directly from the cache, while the nevervalidate policy never generates a Validate request. Of more interest to our overall performance study, the PCV policies generate the least number of Validate requests.

In calculating the performance results reported in this paper two adjustments are made. First, the cost of a full Get request represents the costs for actual resources retrieved. Thus, if the average size or latency for resources actually retrieved is larger than the average for the entire log then $c[\mathrm{Get}, \mathrm{Bandwidth}]$ and $c[\mathrm{Get}, \mathrm{Response}]$ are greater than one. This situation may occur if a cache replacement policy caches smaller resources to increase the hit rate, but results in higher costs for cache misses.

Second, it is not fair to measure the impact of PCV policies without accounting for their increased costs. Consequently the bandwidth and response costs are increased for these policies based on the size in bytes of a piggybacking validation, the response time for the server to do the validation and the number of piggyback validations per request. In the simulation, 50 bytes are added to the request packet and 0.1ms is added to the response time for each piggybacked validation. The number of validations varies, but for the pcvadapt policy using the standard parameters, the average number of piggybacked validations requests is 1.1 for the Digital and 7.0 for the AT&T logs. The difference between the two averages seems to stem from the fact that there is a higher degree of locality of reference in the AT&T logs than in the Digital logs, possibly due to the higher user population in the latter case.

5.2 Staleness Evaluation

The staleness ratio is the number of known stale cache hits divided by the number of total requests (both serviced from cache and retrieved from the server). We chose not to measure staleness as the more traditional ratio of stale cache hits divided by number of cache hits because of differences in the in-cache hit ratio (shown in Tables 3 and 4) caused by differences in the validation and invalidation approaches of the policies. Using our staleness ratio definition allows for a fairer comparison of the policies, although it deflates the ratios in comparison to measuring the ratio of stale cache hits.

5.3 Goodness Evaluation

We believe that the cost and staleness evaluations provide fair and appropriate measures to compare the various policies. However, a good cache coherency policy should minimize both cost (relative to the cache replacement policy) and staleness. To combine cost and staleness into a single metric, we compute an overall "goodness" metric, which combines the average cost and staleness relative to the range defined by the alwaysvalidate and nevervalidate policies for a given cache size. These two policies define the minimum and maximum costs and staleness relative to a cache replacement policy and cache size. It is subjective as to whether minimizing cost or staleness is more important. In the results presented in this paper they are given equal weight when computing the goodness metric.

6 Results

6.1 Variation of Cache Size Parameter

The following results are shown for variation in cache size from 1MB to 8GB. Figures 1–4 show the response time, bandwidth, request message and average costs for the respective policies using the Digital logs. As expected, the alwaysvalidate policy performs the worst while nevervalidate performs the best. In fact, Figure 3 shows that the alwaysvalidate policy provides the worst possible performance (cost of 1.0) for the request message cost because it generates a request (either Get or Validate) for each requested resource.

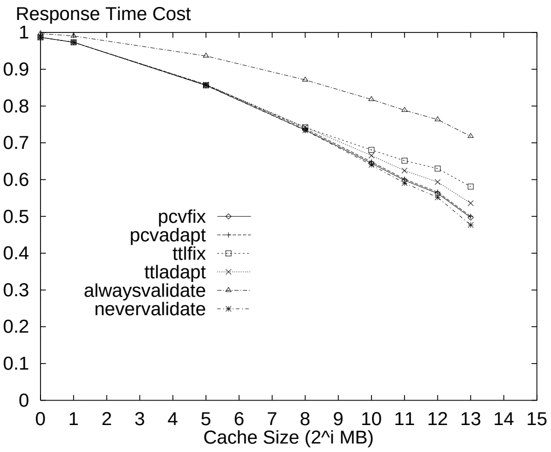

Figure 1: Response Time Cost versus Cache Size for Digital Logs

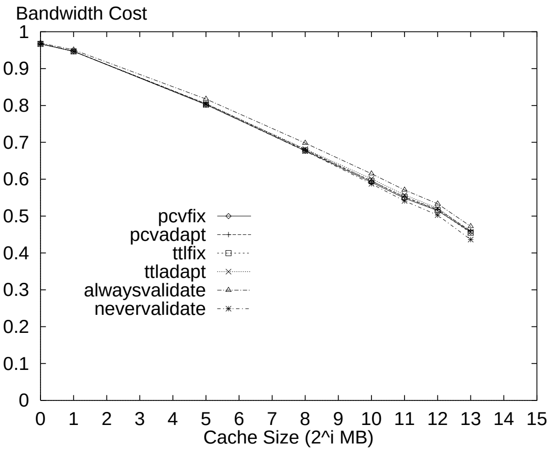

Figure 2: Bandwidth Cost versus Cache Size for Digital Logs

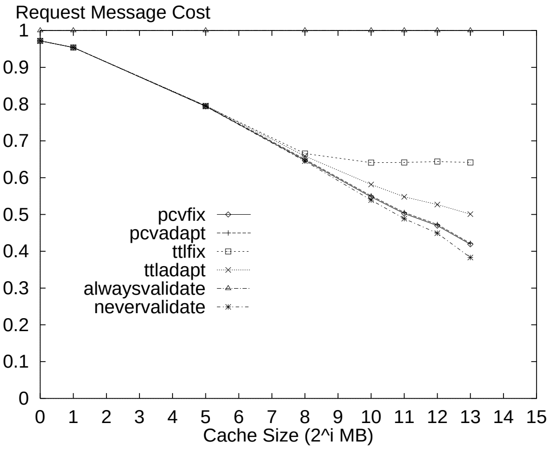

Figure 3: Request Message Cost versus Cache Size for Digital Logs

The differences between policies in the figures is primarily a function of ratio of Validate requests that each generates and the evaluation criterion cost for such a request. Thus, there is little distinction between the policies in Figure 2 because the bandwidth costs for a Validate request are minimal. On the other hand, there is more differentiation between the policies for the other evaluation criteria because the Validate request costs for these criteria are nontrivial.

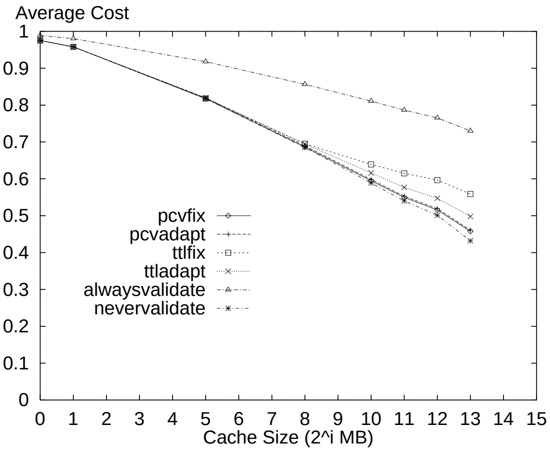

Figure 4: Average Cost versus Cache Size for Digital Logs

In comparing the policies, the PCV policies are next to the nevervalidate policy in incurring the least cost. This result is because the PCV policies reduce the number of Validate requests (also see Table 3) in comparison to the TTL-only policies therefore making better use of the cache contents. Specifically, the pcvadapt policy reduces the number of request messages by 16% and the average cost by 8% in comparison to the ttladapt policy for a 8GB cache.

In reporting the results, we note that the costs for the PCV policies are overstated. Due to the nature of the simulation we do not discover the invalidation of a cached resource until the next time that resource is accessed. If a cached resource is invalidated and then removed by the cache replacement policy before its next access then the invalidation is not discovered at all in our simulation. The cost results are pessimistic because the cache space for these invalidated resources is not freed at the time of invalidation as would be the case in an actual implementation. With this limitation, we measure an invalidation rate of 0.26 invalidations for every 100 resources accessed (not just those in cache) in the Digital logs for the standard parameter set and the pcvadapt policy.

Figure 5 shows the ratio of all requested resources returned as stale from the cache relative to the cache size. While nevervalidate has low staleness values for small caches, it rises to 4% for a cache size of 8GB in the Digital logs. The alwaysvalidate policy results in no stale resources. The other policies all have relatively low staleness ratios (less than 1%) with the pcvadapt policy clearly providing the strongest coherency. In comparison to the ttladapt policy, the pcvadapt policy reduces the staleness ratio by 65% for a 8GB cache.

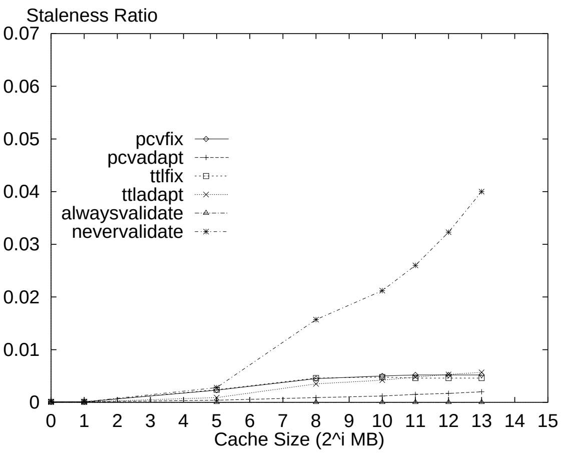

Figure 5: Staleness Ratio versus Cache Size for Digital Logs

Figure 6 shows the goodness metric of combining average cost and staleness performance measures using a weighted sum with each performance metric having equal weight. By definition, the alwaysvalidate and nevervalidate policies have a value of 0.5 at all cache sizes because they define the upper and lower bounds for goodness. As shown, the pcvadapt policy exhibits a high degree of goodness for large cache sizes indicating it is an excellent policy in providing up-to-date resources at a low cost. The pcvfix policy is the second best. For larger cache sizes the nevervalidate staleness ratio goes up (Figure 5), which causes less differentiation in the relative policy staleness ratios. This effect causes the relative differences in policy costs to be reflected more in the goodness metric and contributes to the dropoff in the goodness metric for TTL-based policies.

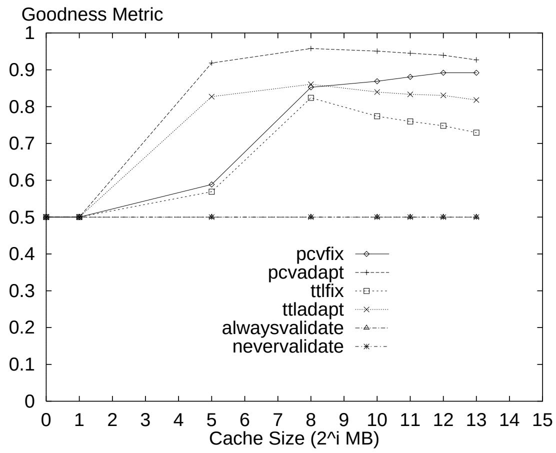

Figure 6: Goodness Metric versus Cache Size for Digital Logs

Figures 7–10 show the response time, bandwidth, request message and average costs for the respective policies as the cache size is varied using the AT&T logs. The relative ordering of the policies is the same as for the Digital logs with the differences between the policies generally more pronounced. This differentiation is a result of relatively more Validate requests being generated for the AT&T data as shown in Table 4. In comparison with the Digital data, the bandwidth costs can be differentiated in Figure 8 with the PCV policies resulting in the highest bandwidth costs. This result indicates that the piggybacked validations for the AT&T data add enough bytes to slightly increase the costs relative to the other policies, although the PCV policies provide lowest response time, request and average costs. Specific comparisons for a 8GB cache show the pcvadapt policy reduces message cost by 17% and the average cost by 6% in comparison to the ttladapt policy. The invalidation rate for the AT&T logs is 2.4 per 100 accesses, which is higher than for the Digital logs due to more piggyback validations being generated.

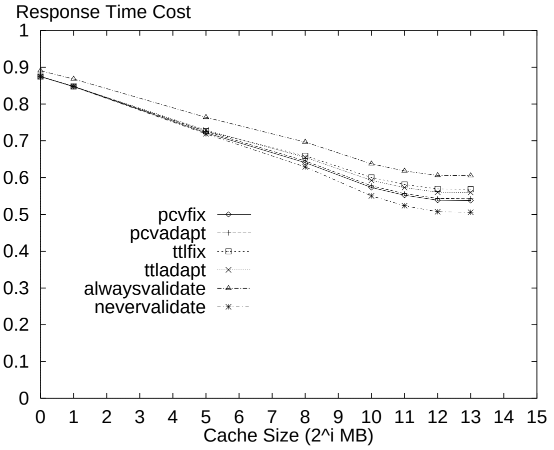

Figure 7: Response Time Cost versus Cache Size for AT&T Logs

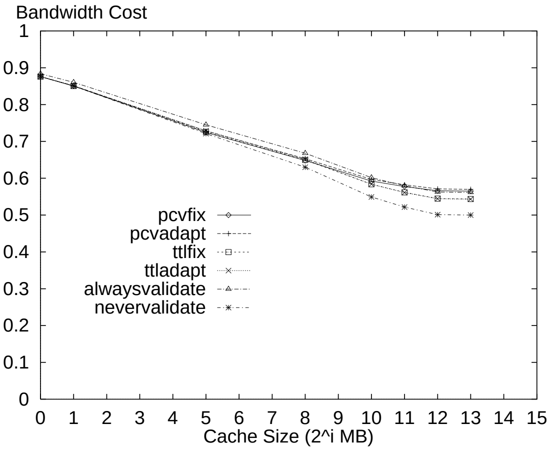

Figure 8: Bandwidth Cost versus Cache Size for AT&T Logs

Figure 11 shows similar staleness results for the AT&T logs as for the Digital logs with the nevervalidate policy returning nearly 7% stale resources for large cache sizes. The other policies return close to 1% stale resources with the pcvadapt policy providing the most strongly coherent results (57% improvement over the ttladapt policy for a 8GB cache) next to the alwaysvalidate policy. The resulting goodness metric results for the AT&T logs are shown in Figure 12 with comparable results to those shown for the Digital logs.

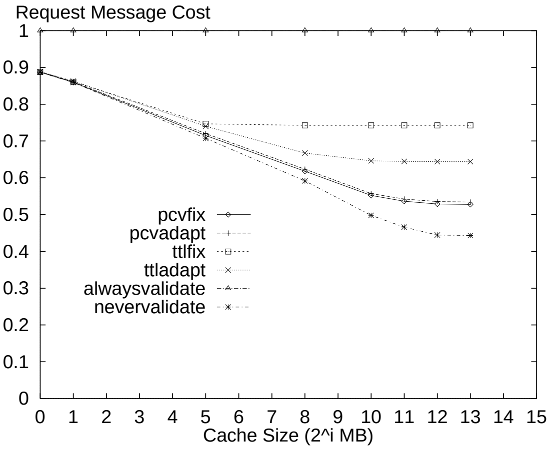

Figure 9: Request Message Cost versus Cache Size for AT&T Logs

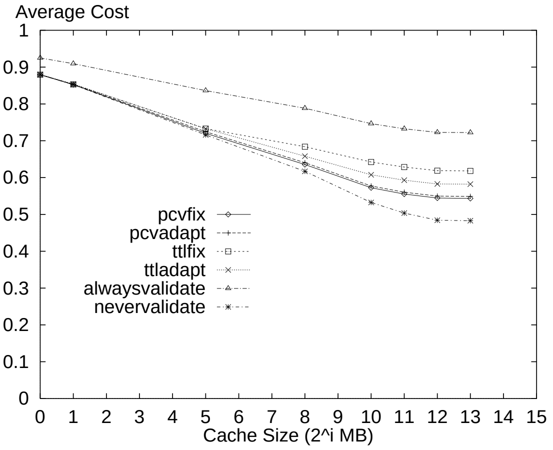

Figure 10: Average Cost versus Cache Size for AT&T Logs

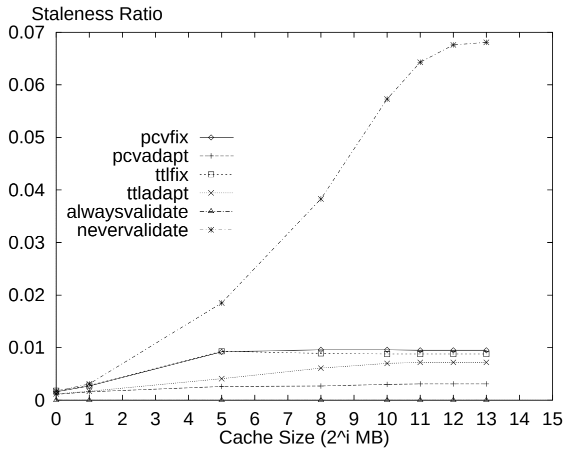

Figure 11: Staleness Ratio versus Cache Size for AT&T Logs

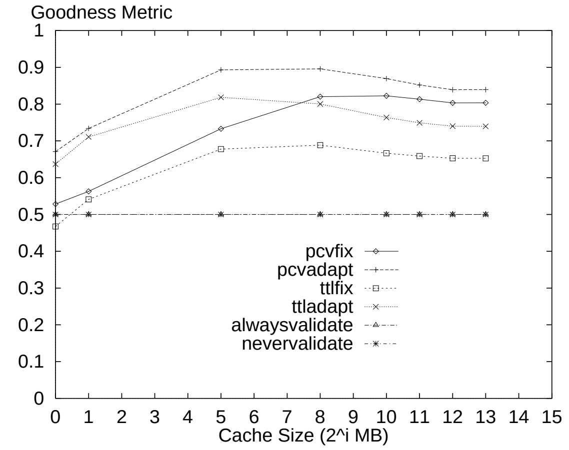

Figure 12: Goodness Metric versus Cache Size for AT&T Logs

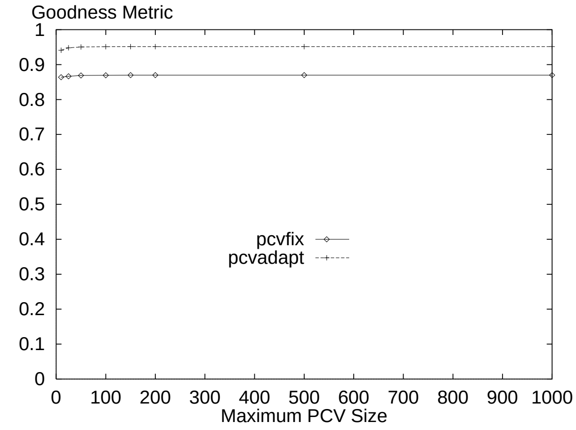

Figure 13: Goodness Metric versus Maximum PCV Size for Digital Logs

6.2 Variation of Other Parameters

The remaining results summarize the effects on the goodness metric of varying the maximum PCV list size, time-to-live and adaptive threshold parameters for the Digital and AT&T logs (the specific cost and staleness results are not shown). These results reflect the output of simulations where the dependent parameter was varied while the other parameters were set to their base values where the base cache size is 1GB. The goodness metric is again calculated relative to the alwaysvalidate and nevervalidate policies for a cache of this size.

Figures 13 and 14 show there is little variation in results based on the maximum PCV list size. Smaller maximums than the default of 50 result in slightly degraded performance, but reflect that there is generally enough traffic between the client proxy and the server to satisfy the piggybacking needs of the proxy. Allowing larger PCV lists does not improve the performance of the PCV policies.

Figures 15 and 16 show some variation between the policies for smaller TTL parameters, but the performance ordering of the policies remains the same. As the TTL value is raised, the two adaptive policies perform similarly as do the two fixed TTL approaches. Use of a bigger TTL value reduces the potential benefit of the PCV policies.

Finally, Figures 17 and 18 show the expected that the lower the adaptive threshold the better the relative results for the adaptive policies. The base value of 0.1 is relatively good in the range.

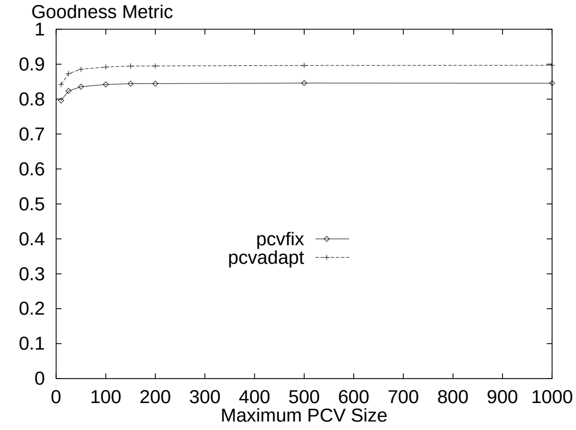

Figure 14: Goodness Metric versus Maximum PCV Size for AT&T Logs

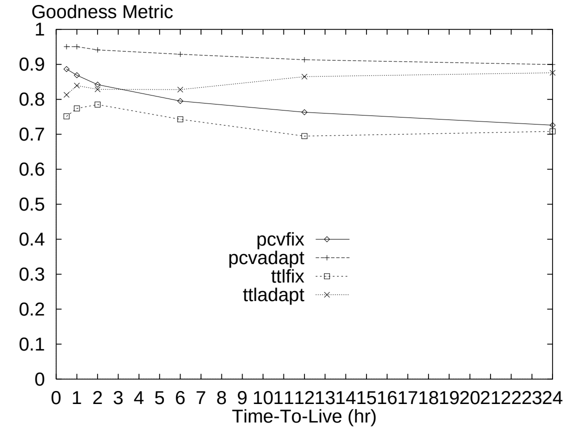

Figure 15: Goodness Metric versus Time-To-Live for Digital Logs

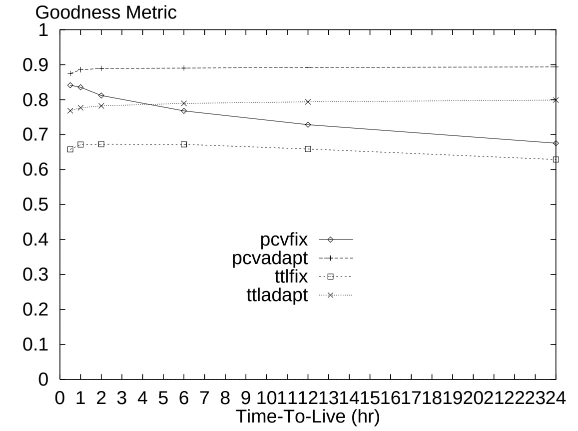

Figure 16: Goodness Metric versus Time-To-Live for AT&T Logs

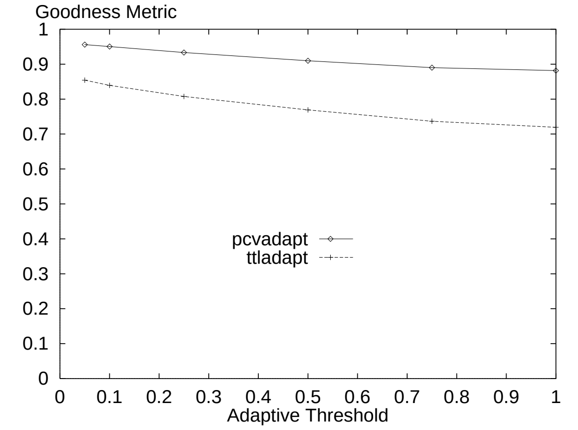

Figure 17: Goodness Metric versus Adaptive Threshold for Digital Logs

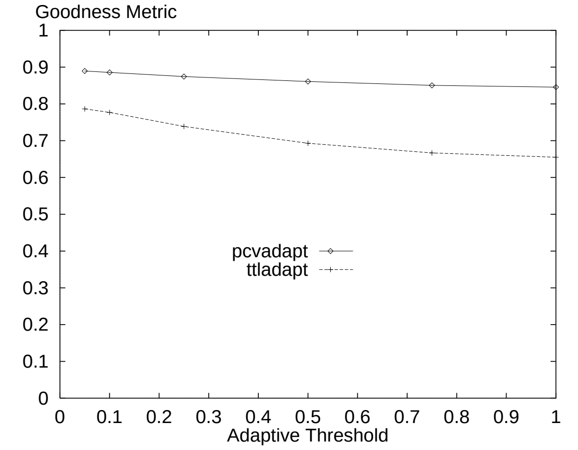

Figure 18: Goodness Metric versus Adaptive Threshold for AT&T Logs

7 Summary and Future Work

We believe the similarity of results on two large, independent data sets clearly demonstrates the merits of the piggyback cache validation approach. By combining piggybacking with an adaptive threshold, the pcvadapt policy is clearly the best at providing close to strong coherency at a relatively low cost when we consider response latency, request messages, bandwidth and average costs. In comparison to the ttladapt policy, the best TTL-based policy, the pcvadapt policy, the best PCV-based policy, reduces the number of messages to the server by 16-17% and the average cost by 6-8%. The pcvadapt policy reduces the staleness ratio by 57-65% in comparison to the ttladapt policy. Additionally, the PCV policies can easily be implemented with the HTTP 1.1 protocol.

Other directions for piggybacking of information include a server piggybacking resource invalidations on replies to a client. These invalidations could be for all the resources at its site or selected subsets of resources [15]. Clients then use the list of invalidations to remove stale copies. Piggybacking of such information onto existing server replies does not introduce new network traffic and alleviates servers having to maintain client lists in comparison to previous server invalidation work [14].

Although not studied in this work, cache coherency is related to cache replacement as a proxy cache works to provide up-to-date resources at the lowest cost. Piggybacking has the potential to positively affect cache replacement decisions. For example, frequently invalidated resources would be good candidates for cache replacement. Proxy caches can also use information about resource usage piggybacked on server replies in making replacement decisions. Additionally, recent work [7] shows that resources that change are accessed more often, there is variation in the rate of change across content types, and the most frequently referenced resources cluster around specific periods of time—all of which can be used while deciding what and when to piggyback.

8 Acknowledgments

We thank Digital Equipment Corporation for making their proxy traces available to us. We thank Anja Feldmann for help with the AT&T trace, Steve Bellovin and Phong Vo for discussions, Jeffrey Mogul, Misha Rabinovich, Jennifer Rexford, Fred Douglis, and the anonymous reviewers for making comments on draft versions of the paper.

References

[1] Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, Stephen Williams, and Edward A. Fox. Caching proxies: Limitations and potentials. In *Proceedings of the Fourth International World Wide Web Conference*, December 1995. `http://www.w3.org/pub/Conferences/WWW4/Papers/155/`.

[2] Azer Bestavros. Using speculation to reduce server load and service time on the WWW. In *Proceedings of the 4th ACM International Conference on Information and Knowledge Management*, November 1995.

[3] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Symposium on Internet Technology and Systems*. USENIX Association, December 1997.

[4] V. Cate. Alex – A global filesystem. In *Proceedings of the USENIX File System Workshop*, pages 1–12. USENIX Association, May 1992.

[5] Digital Equipment Corporation. Proxy cache log traces, September 1996. `ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`.

[6] Adam Dingle and Tomas Partl. Web cache coherence. In *Proceedings of the Fifth International World Wide Web Conference*, May 1996. `http://www5conf.inria.fr/fich_html/papers/P2/Overview.html`.

[7] Fred Douglis, Anja Feldmann, Balachander Krishnamurthy, and Jeff Mogul. Rate of change and other metrics: a live study of the world wide web. In *Symposium on Internet Technology and Systems*. USENIX Association, December 1997.

[8] James Gwertzman and Margo Seltzer. World-wide web cache consistency. In *Proceedings of the USENIX Technical Conference*, pages 141–152. USENIX Association, January 1996. `http://www.usenix.org/publications/library/proceedings/sd96/seltzer.html`.

[9] Barron C. Housel and David B. Lindquist. Webexpress: A system for optimizing web browsing in a wireless environment. In *Proceedings of the ACM/IEEE MOBICOM '96 Conference*, October 1996. `http://www.networking.ibm.com/art/artwewp.htm`.

[10] J. H. Howard, M. L. Kazar, S. G. Menees, D. A. Nichols, M. Satyanarayanan, and R. N. Sidebotham. Scale and performance in a distributed file system. *ACM Transactions on Computer Systems*, 6(1):55–81, February 1988.

[11] Internet Engineering Task Force. Hypertext transport protocol – HTTP/1.1, January 1997. `http://ds.internic.net/internet-drafts/`.

[12] Balachander Krishnamurthy and Craig E. Wills. Piggyback cache validation for proxy caches in the world wide web. In *Proceedings of the 2nd Web Caching Workshop*, Boulder, CO, June 1997. National Laboratory for Applied Network Research. `http://ircache.nlanr.net/Cache/Workshop97/Papers/Wills/wills.html`.

[13] Thomas M. Kroeger, Darrel D.E. Long, and Jeffrey C. Mogul. Exploring the bounds of web latency reduction from caching and prefetching. In *Symposium on Internet Technology and Systems*. USENIX Association, December 1997.

[14] Chengjie Liu and Pei Cao. Maintaining strong cache consistency in the world-wide web. In *Proceedings of the 17th IEEE International Conference on Distributed Computing Systems*, May 1997.

[15] Jeffrey Mogul. An alternative to explicit revocation?, January 1996. `http://weeble.lut.ac.uk/lists/http-caching/0045.html`.

[16] Jeffrey C. Mogul. Hinted caching in the web. In *Proceedings of the 1996 SIGOPS European Workshop*, 1996. `http://mosquitonet.stanford.edu/sigops96/papers/mogul.ps`.

[17] Jeffrey C. Mogul, Fred Douglis, Anja Feldmann, and Balachander Krishnamurthy. Potential benefits of delta-encoding and data compression for HTTP. In *ACM SIGCOMM'97 Conference*, September 1997. `http://www.acm.org/sigcomm/sigcomm97/papers/p156.html`.

[18] Michael N. Nelson, Brent B. Welch, and John K. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1):134–154, February 1988.

[19] Venkata N. Padmanabhan and Jeffrey C. Mogul. Using predictive prefetching to improve world wide web latency. *Computer Communication Review*, 26(3):22–36, 1996.

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

[21] Stephen Williams, Marc Abrams, Charles R. Standbridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *Proceedings of the ACM SIGCOMM Conference*, pages 293–305, August 1996. `http://www.acm.org/sigcomm/sigcomm96/williams/p156.html`.

[22] Craig E. Wills and Joel Sommers. Prefetching on the web through merger of client and server profiles, June 1997. `http://www.cs.wpi.edu/~cew/papers/webprofile.ps`.