Proceedings of USITS' 99: The 2nd USENIX Symposium on Internet Technologies & Systems

Boulder, Colorado, USA, October 11–14, 1999

PROVIDING DYNAMIC AND CUSTOMIZABLE CACHING POLICIES

J. Fritz Barnes and Raju Pandey

Providing Dynamic and Customizable Caching Policies[*]

J. Fritz Barnes Raju Pandey
Parallel and Distributed Computing Laboratory
Computer Science Department
University of California, Davis, CA 95616
{barnes, pandey}@cs.ucdavis.edu

Abstract

Web caching has emerged as one solution for improving client latency on the web. Cache effectiveness depends on the policies used to route requests to other caches and servers, to maintain up-to-date web objects and to remove objects from the cache. Traditional caches apply one set of policies, which determines the efficiency as well as the effectiveness of the caches. This set of policies often does not exploit the diversity inherent in different web objects, caches and clients. Policies that do exploit this diversity result in convoluted caching policies that attempt to combine multiple policies and guess at the unknown characteristics of web objects, caches and clients.

In this paper, we present an extensible caching infrastructure in which cache administrators, servers, and end users can customize how web objects are cached, replaced, and kept consistent. The infrastructure includes a domain-specific language, CacheL, for defining customizable caching policies that can be changed dynamically. Analysis of our prototype, PoliSquid, shows the benefits of the infrastructure for variable coherency policies, localized removal policies, and early removal of objects from servers.

1 Introduction

Web caching [15, 21, 8] improves client latency through the use of intermediate servers or caches, that accept HTTP [10] requests from clients. When a request is made to a cache, the cache returns a copy of the web object being requested. If a replica does not exist, the cache contacts other caches or the origin server to retrieve a copy. Caches improve web performance by migrating web objects closer to clients. Further, by not contacting the server, caches reduce origin server load.

Caches use several policies for cache management. Cache policies determine:

- which objects and when to pre-fetch (*pre-fetch policy*),
- how requests for objects are routed (*routing policy*),
- where objects are placed in a cache hierarchy (*placement policy*),
- how cache objects are kept fresh (*coherency policy*), and
- which objects to remove when the cache is full (*removal policy*).

Research on caches has focused primarily on developing optimum policies for all users and objects. For instance, several removal policies such as Least Recently Used (LRU), Least Frequently Used (LFU), Perfect LFU [3], and removal of large objects from caches (SIZE) [23] try to reduce bytes requested from the server and client latencies. These policies depend on a single attribute of the web object. For instance, SIZE removes large objects from the cache which make room for multiple smaller objects. As a result, caches hit rate improves. However, there are a variety of cache performance measurements. Often a single web object attribute will not necessarily improve all measurements. The SIZE algorithm improves the hit rate, but may degrade client latencies

[*]The authors are partially supported by the Defense Advanced Research Project Agency (DARPA) and Rome Laboratory, Air Force Materiel Command, USAF, under agreement number F30602-97-1-0221.

due to the high network overhead of uncached large documents.

Several object replacement algorithms have been developed that extend the single attribute algorithms by weighting multiple attributes or applying cost functions. These algorithms include: a HYBRID policy [24] that uses a weighted combination of object attributes; Greedy-Dual-Size [4] that uses an appropriate cost function; and biased replacement policies [12] that use a combination of weights based on the origin server and LRU. These policies provide more flexibility in optimizing multiple cache performance measurements. However, knowledge about the cache environment, such as the bandwidth of the nearby network topology, cannot be explicitly represented in these algorithms. Parameterizations and cost functions are not sufficiently general enough to define policies for specific environments.

In addition to dependence on multiple parameters, cache policies may also depend on the semantics of the cache objects. For instance, if we consider objects in a cache, we observe that there is a large variance in hit rates for objects with different content types. Image requests generate a hit rate that is about 4-5 times greater than the hit rate for hypertext files [18]. In other words, a single set of policies cannot be applied to a diverse population of objects, caches and clients.

Further, cache policies are often dynamic in nature. An example is the applicability of cooperative algorithms in different operating conditions. Cooperative caching improves performance by sharing objects across multiple caches. This effectively increases the cache store size. Additionally it increases the number of potential clients, which increases the probability that two clients will request the same object. An analysis of cooperative algorithms [14] concludes that the overhead in communication between cooperating caches may outweigh the benefit of additional hits provided by a neighboring cache. Dynamic policies provide better support in these circumstances. Cooperation should be used when the overhead is inconsequential compared to the reduced bandwidth and increased hit rates. However, cooperation should be turned off when the overhead becomes too great. Therefore, caching policies should be dynamic in response to the ever-changing pattern of web requests.

Thus, what is needed is a caching infrastructure that allows both clients, servers, and caches to specify and customize policies to suit semantics of objects, variations in parameters, and the dynamic behavior of the web. In this paper, we present such a caching infrastructure. In this infrastructure, a cache policy is defined by registering a set of actions with a set of events. The events denote entry points for applying caching policies, whereas the actions implement specific caching policies. The infrastructure includes a domain specific language, called CacheL, for specifying the actions. The infrastructure also allows caching policies to be changed dynamically in order to take advantage of the changing environment.

We have implemented the infrastructure as a cache simulator, DavisSim and as part of a web cache, PoliSquid. We evaluate the benefit of: allowing variation in client tolerance for fresh documents; customizing removal policies to the network topology and using object semantics to remove objects earlier than the standard removal policy. The results of the experiments demonstrate that the customizations help improve cache performance. Finally, we investigate the overhead associated with adding extensibility to a caching system, Squid [20]. The results show that the overhead is moderate (about 8.5%) and can be improved significantly with aggressive optimizations.

This paper is organized as follows: We present a taxonomy of the different caching policies and how they can be customized in Section 2. In Section 3, we describe an infrastructure for providing these customizations. In Section 4, we briefly present the design of a caching simulator and our prototype implementation. We present the performance analysis of our infrastructure in Section 5. We describe related work in Section 6. We conclude with a summary and discuss future work.

2 Customizable Caching Policies

In this section, we look at the various cache policies and how they can be extended. The behavior of a cache is defined by a set of cache policies: pre-fetch, routing, placement, coherency, and removal policies. In Table 1, we list the different policies and the applicable customizations. Below we focus on the pre-fetch, routing, and coherency policies in more detail.

Policies	**Creator of customized policy**		
	Client	**Cache**	**Server/Web Object**
pre-fetch policy	client-side pre-fetch	traditional pre-fetch	push-caching
routing policy	routing based on clients	complicated peering arrangements	mirroring servers
placement policy	not-applicable	cooperative caching	push-caching
coherency policy	personalize user tradeoffs	policy for shared documents	specify when objects expire
removal policy	not-applicable	take advantage of costs realized at the cache	take advantage of semantic content
miscellaneous	content transducers	measurement & tuning	protocol extensions

Table 1: Analysis of different caching policies

Pre-fetching policies request objects before they are requested by a client. Pre-fetching increases the hit rate because the first document access might result in a hit. However, pre-fetching can increase the amount of bandwidth requested by the server if a pre-fetched object is never accessed. The effectiveness of pre-fetching depends on how well the pre-fetching policy can predict which objects will be accessed and therefore should be retrieved in advance.

Pre-fetching can be used in different ways. A client might use pre-fetching to load objects from the cache into the browser. This may be of particular interest in reducing the latency due to slow dialup connections. Allowing pre-fetch customization allows the cache to support different browsers using different schemes to perform pre-fetching of documents into the browsers.

Caches could be customized to accept objects pushed out from servers. This provides a technique whereby servers can perform push-caching of popular documents. As a result, fewer requests will be served at the origin server reducing server contention and improving performance.

Routing policies determine how a cache retrieves an object. Clients, caches and servers might each use the routing policy in a different manner. The manner in which the cache routes outgoing requests might be determined by the client. Let us consider a network that supports differentiated services. Clients could specify routing policies that use different priorities of services. A cache administrator, on the other hand, might specify a routing policy that allows cooperation among multiple caches. A routing policy provided by a web site might be used to alternate requests between different mirrors of that web site. Even better, the routing policy might determine the optimal mirror the cache should contact.

Coherency policies determine how a cache responds to a request for an object in the cache. The coherency policy decides either to consider the object fresh or stale. In the case of stale objects, the cache consults with the server to verify that the copy is up-to-date. A commonly used algorithm in deciding coherency is the TTL algorithm [6]. This algorithm determines whether an object is fresh by evaluating the equation:

$$T_{now} - T_{in} < k\,(T_{in} - T_{last\_mod}) \qquad (1)$$

where T_{now} is the time of the request, T_{in} is the time when the object entered the cache or was last verified, and T_{last_mod} is the time when the object was last changed. If the boolean expression is true than the document is considered fresh. Different clients may desire different values of k. Larger values of k result in fewer validation checks which would increase hit rates with a tradeoff of sometimes returning stale objects. Caches set the global default coherency policy. Additionally, a cache might place limits on the range of client customization. Servers can specify an invalidation scheme to use, or an algorithm for deciding coherency that takes into account their object/server specific characteristics. For example, a server specified coherency policy might take advantage of the fact that all of its objects are only updated in the morning.

In the next section, we describe in more detail our infrastructure for specifying these customized policies.

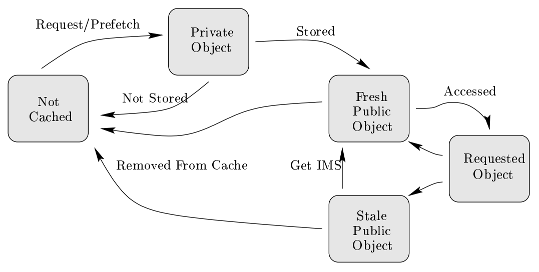

Figure 1: States of a document in a cache

3 Customizable Infrastructure

In this section, we discuss our caching infrastructure for creating customizable policies. Our goal is to create a caching system in which clients, cache administrators, or web object authors can customize caching policies. The caching infrastructure provides the ability to specify different routing, placement, coherency, and removal policies for objects. We first discuss an event-based model, which provides the technique for specification of policies. These policies are written in a domain-specific language, CacheL. We next describe the architecture used to attach policies to web objects. We conclude this section with an example that uses CacheL to provide a customized removal policy.

3.1 Event-based Architecture

The caching infrastructure is based on subdividing a policy into actions that are taken when specific events occur. In Figure 1, we show the pertinent states in the lifetime of a web object. The arcs in the figure represent events that cause an object state to change.

For example, we will discuss the changes in state of the web object, `http://wonderland.net/tea-party.html`, as first the Dormouse and later Alice request the tea party object. Initially, the tea party object exists in the Not Cached state. When the Dormouse makes a request for the tea party object, the routing policy is used to determine how the cache retrieves a copy of the tea party. This object enters the Private Object state. Now, the placement policy determines whether we store the tea party object in the cache. If we store the object it enters the Fresh Public Object state. Later, when Alice requests the tea party object, we enter the Requested state. At this point the coherency policy determines whether the document is fresh or stale and, if stale, checks whether the original has changed using a Get If-Modified-Since (IMS) request.

The amount of space available in the cache also affects the state of web objects. When the cached bytes exceed capacity, the cache uses a removal policy to determine the best objects to be removed. These objects leave the Fresh or Stale Public Object states and return to the Not Cached State.

3.2 CacheL

CacheL defines policies with a policy script. A policy script specifies a set of actions to take when different cache events occur. An action consists of a set of cache operations and expressions. Cache operations include contacting other caches and servers, changing the state of local objects, and setting alarms to schedule an action in the future.

The language supports simple variables; relational, arithmetic and logical expressions; conditional execution; set, array, numeric, and string data types; and iteration over sets. It also supports facilities for manipulating date/time values, as well as MIME headers associated with objects.

3.2.1 Cache Events

The infrastructure provides a predefined set of events. We describe what causes these events as well as required and optional cache operations that should be associated with these events.

Route events occur when objects must be retrieved. The action associated with a Route event should send an HTTP request to a cache or the origin server, return an alternate document in the cache, or respond with an error.

New-Store events occur after new documents are retrieved or Get IMS requests for stored documents return new documents. Associated actions should decide whether the object should become a Fresh Public Object or return to the Not Cached state.

Access-Inline events occur before responding to a request made for an object in the cache. Associated actions should handle coherency policies.

Actions can also modify the MIME headers of the object requested.

Access-Offline events are triggered by a request but handled independent of the response. Actions that do not affect the response should be attached to this event. For instance, a counter could be used to keep track of object accesses.

High-Water events occur when the bytes stored in the cache exceed a predetermined level. Associated actions should purge one or more objects from the cache.

Purge events occur when a document is removed from the cache. This event allows actions to inform another party when an object is removed, or to adjust policy-specific state.

Timer events occur as a result of policy-defined alarms. Actions are policy dependent.

3.2.2 Cache Operations

We enumerate some of the pertinent cache operations:

CacheFetch(URL) requests a web object from the object's originating server.

CacheFetchIMS(URL) performs an If-Modified-Since request for the given web object.

CachePost(URL, Data) contacts a host with a post request and includes the Data in the request. This is used primarily for server-specific policies to pass information back to their originating server.

CacheICPFetch(HOST, URL) requests a web object from another cache using the Internet Caching Protocol.

CacheResponse(URL) sends a local copy of URL to the client as a response to the client's request.

CachePurge(URL) removes the specified object from the local store.

CacheStore(URL) instructs the cache to store the object that is being requested.

CacheSetTimer(Date, Time) sets an alarm.

CacheLog(Message) provides a debugging mechanism that writes the given message to the cache's log file.

3.3 Cache Policy Management

The infrastructure provides support for cache clients, cache administrators, and the web site author to specify caching policies. Cache administrators specify policies locally and can refuse to allow servers to modify policies. Currently, we require a cache administrator to specify client policies by mapping client identifiers to policies. However, in future work we would like to investigate alternative techniques for clients to post policies.

In order to specify policies, cache administrators construct a configuration file that contains a list of web objects and the URLs of policy scripts. The Web objects can occur multiple times within the configuration file for different policies. Cache administrators can utilize wildcards for easy specification of policies that apply to multiple objects and hosts.

In addition to allowing cache administrators to define policy mechanisms, server administrators can define policies relevant to a web object on their server. Policies are specified by adding an X-Cache-Policy MIME header that specifies the URLs of the policy scripts that apply to the object. When a cache retrieves the object with an X-Cache-Policy header, it retrieves the policies if they are not already in the cache. Then it applies the actions for the web object so that the desired policies can be enforced.

3.4 Removal Policy Example

We provide an example situation where a cache administrator wishes to customize the cache removal policy. This example provides a concrete demonstration of actions written in CacheL. More examples can be found on our web page `http://pdclab.cs.ucdavis.edu/qosweb/CacheL/` and in a previous paper [1].

In this example, the caching administrator controls caches distributed across several different work sites. A cache exists at each of these sites and a single connection to the Internet is provided. The bandwidth to the Internet is limited, and the bandwidth between the cache locations is plentiful. This scenario is shown in Figure 2, where Sprockets International has offices in San Jose, Boulder, and Boston, with

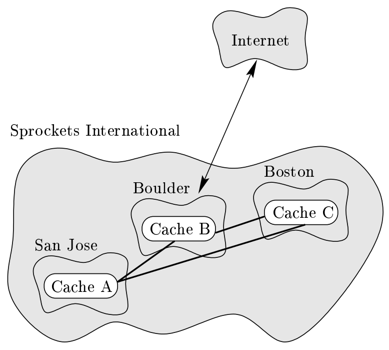

Figure 2: Cooperative Caches located at different sites for Sprockets International

```
Event: Initialize
   CreateList( "Unique" );
   CreateList( "Duplicate" );

Event: New-Store( obj )
   if ( CacheDirectoryLookup( obj ) == "NotFound" )
       ListAdd( "Unique", obj );
   else
       ListAdd( "Duplicate", obj );
   endif

Event: High-Water
   if ( ! IsListEmpty( "Duplicate" ) ) {
      obj = GetLast( "Duplicate" );
      if ( CacheDirectoryLookup( obj ) != "NotFound" )
         CachePurge( obj );
         ListRemove( "Duplicate", obj );
      else
         ListRemove( "Duplicate", obj );
         ListAdd( "Unique", obj );
      endif
   else
      obj = GetLast( "Unique" );
      CachePurge( obj );
      ListRemove( "Unique", obj );
   endif
```

Figure 3: Actions used to implement a specialized removal policy

a single connection to the Internet. As a result of the limited bandwidth to the Internet, the cache administrator would like to favor removal policies that evict objects that are cached in multiple locations, thus maximizing the total number of objects cached.

We show an implementation of this policy in Figure 3. The policy employs several actions associated with different events. Our infrastructure provides multiple linked lists and priority queues that can be used for implementing the removal policy. To implement the desired cache policy, we assume that each cache has a directory containing objects cached by nearby servers [19, 17, 9]. The CacheDirectoryLookup operation consults the local entries in the cache table and returns the neighbor that caches an object or NotFound.

The implementation associates actions with the New-Store and Highwater events as well as initially creating the linked lists that will be used by this policy. When an object is first stored in the cache, the removal policy determines whether another cache already caches the object. If the object is already cached it is stored in the Duplicate linked list; otherwise it is stored in the Unique linked list. When a Highwater event occurs, the cache attempts to purge objects that are cached by other caches. The Highwater action requires that we doublecheck that the objects are still cached by a neighboring cache since one of the neighbors may have removed it.

4 Infrastructure Design and Implementation

We have implemented two caching systems that are based on our extensible caching infrastructure. The first, DavisSim, is an event-based cache simulator. The second, PoliSquid extends the popular Squid [20] web cache by adding the ability to specify cache policies. We have used an event based design for implementing the caching systems. The event-based design matches well with the implementation of Squid, which uses events to handle asynchronous I/O. When an event occurs the caching infrastructure will determine whether an action is attached to that event and execute all attached actions. Below we briefly describe the CacheL interpreter, DavisSim, and PoliSquid.

CacheL Interpreter: We have implemented an interpreter that is invoked for executing actions specified in CacheL. Our current implementation uses a recursive descent interpreter and does not transform the input into an optimized abstract representation. As a result, each execution of an action requires a parsing step. In future versions, we plan to store the abstract representation within the cache in order to avoid the overhead of parsing a policy script every time it is executed.

DavisSim: is based on the Wisconsin Cache Simulator[1]. It uses a pre-processed web trace. This web trace contains the time of request and server, document and client identifiers. Additional information

[1] Available at: `http://www.cs.wisc.edu/~cao/webcache-simulator.html`

about the object, such as last modified time, size, and perceived latency, are included in the trace. Each request in the trace causes either a New-Store event if an object is not already cached, or an Access-Inline event if the object is cached. We do not support Routing events at this time because DavisSim does not simulate cooperative caching.

The cache operations allow scripts to store objects in the cache and purge objects to make more space. Internally the simulator maintains statistics about true/false hits, true/false misses, and latencies. A true hit occurs when the object being requested hits in the cache and the object has not been modified from the version stored in the cache. If the object was modified it is considered a false hit. Similarly, if an IMS request is generated and the object has been modified we refer to this as a true miss, otherwise a false miss. In some of our experiments, we utilize CacheL to provide additional statistics. This is useful when we want to keep statistics about two different types of objects.

PoliSquid: allows us to load CacheL scripts specified by the cache administrator in a configuration file. We store policy files as another object in the cache. When an event occurs for an object, the policy executor looks up the URL associated with that script and then passes this to the interpreter.

5 Analysis

We now analyze the performance behavior of the extensible caching infrastructure. The primary goals of analysis are:

- *Do caches benefit from supporting customizable policies?*
- *What is the overhead of adding extensibility to a caching system?*

In the remainder of this section, we describe the experiments we have conducted in order to address the above questions.

5.1 Effectiveness of Customizable Caching Policies

In this subsection, we analyze the performance of the extensible caching infrastructure when the clients, caches and origin servers implement different caching policies. We consider the benefit of client, cache and server customized policies through three experiments. In the first two experiments, we use DavisSim to evaluate the effects on the cache. We examine a weeks worth of the DEC Squid traces[2] in our simulations. We further refine the dataset for coherency experiments by removing files without last modified times. These requests are ignored because we want to get an idea of how changing factors in the TTL computation affect coherency. We also remove those documents that are requested only once. Experiments involving removal policies use all requests in the week, and use an optimal prediction of whether a file has changed.

We make the following assumptions in the cache simulator. We use the LFU algorithm by default unless otherwise stated for cache removal. We use a value of $k = 0.5$ in the TTL algorithm (Equation 1) to decide coherency unless otherwise stated. If a new request exceeds the size of the cache, then the request is considered a miss and is not stored in the cache. The simulator creates high water events when the size of the documents stored in the cache exceeds 95% of the cache size. The cache discards documents until the stored documents occupy less than 90% of the cache size.

5.1.1 Client-Customized Policies

We perform two experiments that look at the effects of different clients selecting a different factor in the TTL calculation of document freshness. The first experiment explores the benefit of allowing clients to specify different factors in the TTL calculation. In this experiment, we subdivide the clients into two groups: strict clients and lax clients. The strict clients have minimal tolerance for web objects that have changed. As a result they utilize a small constant ($k = 0.001$) in the TTL calculation. However, this small constant penalizes the lax clients that can tolerate some old information. In Figure 4, we show the results of the experiment, varying the

[2]Provided by Compaq at: `ftp://ftp.digital.com/pub/DEC/traces/proxy/webtraces.html`

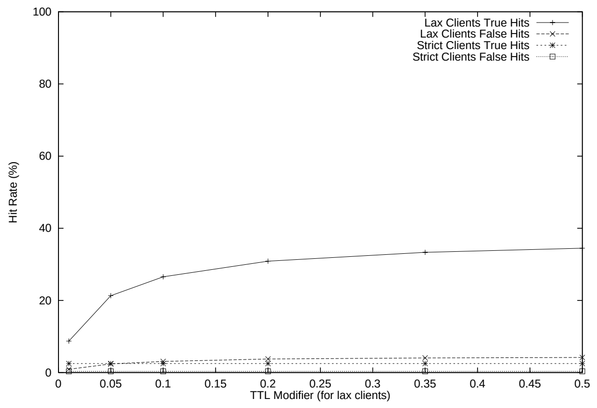

Figure 4: Cache Hit rates when different coefficients are used for lax clients.

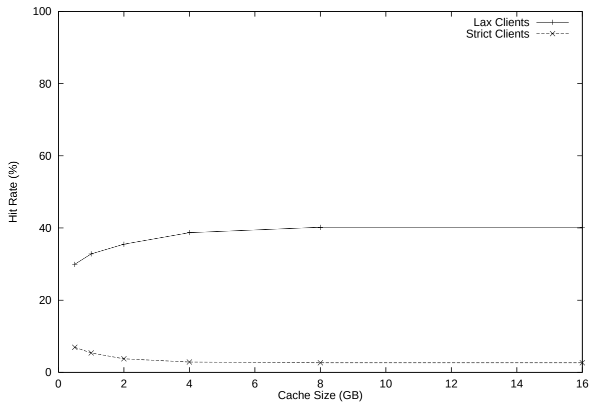

Figure 5: Change in latency when half of the cache clients utilize a restrictive caching policy and the others use a variable caching policy.

TTL constant for the lax clients. The graph plots the true/false hit rates achieved by the clients. The strict clients have a low true hit rate because they require many IMS requests to verify coherency, or false misses. As a tradeoff however, they minimize the number of false hits to about 0.3%. In contrast, the lax clients achieve hit rates 10 times the level of the strict clients at the cost of a higher false hit rate of 4.5%. This experiment demonstrates that it is desirable to allow clients to specify their desired level of coherency.

In the second experiment, we look at the effects of changing the cache size on multiple coherency policies. In this experiment we use a constant k of 0.5 for the lax clients. The results of the experiment, shown in Figure 5, demonstrate that hit rates improve as the cache size increases for the lax clients. However, we see the hit rates decrease for the strict

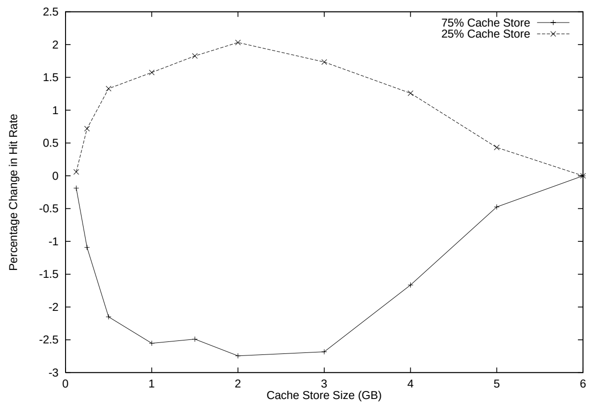

Figure 6: Percentage Differences in hit rates for objects subdivided by incoming network

clients. The reduction in the hit rates for the strict clients is due to the greater reuse of the objects by the lax clients. When the cache is small, it is more likely that an object will not exist in the cache and therefore the the lax clients will miss more often, resulting in objects that are fresh for small k values. However, as the cache size increases, the lax clients requests will generate more hits, thereby decreasing the number of objects that are fresh for small k values.

5.1.2 Cache-Customized Policies

In this experiment, we explore cache-customized policies that take advantage of the network environment near a cache. We assume that the cache's host has a high-bandwidth and low-bandwidth network link. The cache administrator would like to allocate a greater percentage of the cache to documents arriving over the low-bandwidth link in order to increase the hit rate on that link.

DavisSim uses a user-defined function that determines whether incoming requests traverse the high-bandwidth or the low-bandwidth link. We subdivide the cache into two portions: we use 75% of the cache store for the slow link, the rest for the other link. The resulting hit rates are shown in Figure 6. The graph represents the difference in the hit rate when the cache is subdivided to hit rates when the cache is not subdivided and demonstrates that at small and large cache sizes there is not much benefit to using subdivided caching policies. However, at medium cache sizes the hit rate for the slow link is improved by 2%, with a decrease in the hit rate for

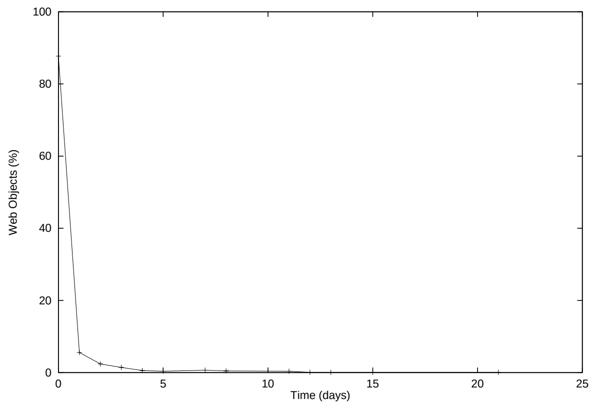

Figure 7: Percentage of objects accessed within x days of the original access

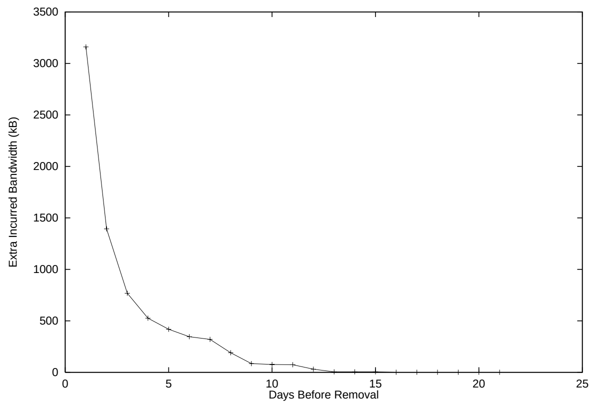

Figure 8: Extra Bandwidth necessary if files are removed early

documents over the fast link. These results, demonstrate that subdividing the cache can adjust the hit rates for different sets of objects when the cache is large enough that objects are not removed immediately and not so large that all objects fit within the cache.

5.1.3 Server-Customized Policies

In this experiment we explore the benefits of using information based on the server's web content for customizing caching policies. The idea in this experiment is to consider a web newspaper that adds new articles each day. We use NLANR logs from December 21, 1998-February 2, 1999 to investigate the unique behavior of news articles and images from the ABCNews web site[3]. The files in our trace represent 29 MB of requested objects. In Figure 7, we plot the time over which a document continues to be requested. We notice that 87% of the web objects are not accessed after 24 hours from the initial request. Therefore, by taking advantage of the object semantics to remove objects early, we can make room for other objects in the cache. To evaluate the benefits, we apply a specialized removal policy that purges documents from the cache after a given number of days have passed. Figure 8 shows the additional bandwidth required to access files that were removed early. If documents are purged after 1-2 days, there will be .5-1.5 MB of documents that will be requested that were previously cached. Most of this bandwidth comes from a few files. A smarter policy would allow exceptions to the one-day purge rule for these few files. We conclude that using the document semantics can provide a benefit in decreased cache utilization by files that won't be accessed again and a slight decrease in hit rate for a few popular pages.

[3]`http://www.abcnews.com/`

5.2 Overhead of Customizable Policies

We now describe an experiment for assessing the overhead of using extensible policies. We wish to measure the overhead of generating events, interpreting CacheL scripts, and taking appropriate actions. Our experiment involves specifying a script to handle Access-Inline events. These events occur every time a hit occurs for an object. Therefore, when we ran the experiment we specified a trace where we achieve a 100% hit rate after a warmup period. In our experiment we wrote the coherency policy contained within Squid, including special cases, as a script, which consisted of 20 lines of CacheL code.

We use three 350 MHz Pentium II PCs running the Linux operating system to perform the experiments. Two machines were used as client/server processes for the Polygraph[4] web proxy performance benchmark. We performed an experiment to measure the latency observed in accessing a 4KB file. We use a single process accessing the files with a short delay between the requests. The experiment measures the mean response latency.

We find that the latency of requests using PoliSquid to calculate the coherency policy incurs an 8.5% overhead compared to the latency for Squid. Fig-

[4]Polygraph is available at: `http://www.ircache.net/Polygraph/`

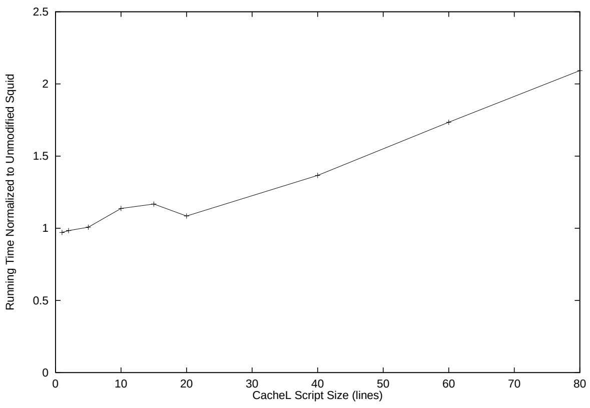

Figure 9: Response overhead of PoliSquid normalized to the running time for Squid using the same requests.

ure 9 contains the results as we vary the number of lines in the coherency policy script. The non-linear result is due to the manner in which different length scripts were created. Scripts were created as follows:

- **1-20 lines**: repetitive TTL calculations (Equation 1), the most expensive statement in the coherency policy script;
- **20 lines**: duplicating squids hard-coded policy in squid;
- **20+ lines**: inserting additional TTL calculations.

We note that an actual coherency policy would be shorter than 20 lines because it would be customized to the attributes of the object. For example, it would not use conditional statements to determine which coherency policy to employ, rather different policies would be attached depending on the object's attributes. In actual operation, the overheads would be less apparent due to larger file sizes, and I/O overheads due to network and disk accesses by a large group of users. In addition, our modified version of Squid has not been optimized, which we believe can reduce the overhead substantially.

6 Related Work

There are two relevant areas of previous work that relate to extensible web caching: techniques to allow customization of caches and schemes that give greater control over caches to servers.

6.1 Cache Customization

The Squid web cache [8, 22, 20] is one of the more popular web caches currently deployed in caching architectures. Squid supports composition of and parameterization of policies. For example, Squid uses expiration, time-to-live (TTL), or constant lifetime coherency policies. The policy depends on whether an expiration or last-modified time was included with the object. Squid allows the cache administrator to customize caching policies through modification of parameters and weights. This differs with our infrastructure, which allows a cache to specify different policies using an interpreted language.

An alternative for communicating between caches and organizing the hierarchies is discussed by Zhang et al. [25]. Their adaptive technique for organizing caches uses separate hierarchies for different servers, to avoid the overload that would occur at a single server. The adaptive cache configures itself and allows hosts to enter and exit cooperation.

Nottingham's work in optimizing object freshness controls [16] considers techniques to avoid object validation. He evaluates optimum parameters for freshness calculation that depend on the object type and the location from which the object was retrieved.

Other languages have been proposed for use on the web. One of these is WebL [13], used for document processing. The goal of this system is to provide both reliable access to web services through the use of service combinators and techniques for gathering information and modifying documents through markup algebra. WebL is intended for a different purpose than CacheL, and in fact combining the two may be fruitful: it would allow caches to make simple customizations of web pages instead of forcing these requests to be handed by the origin server.

6.2 Server Control

Caughey et al. [7] describe an *open caching* architecture for caching web objects that exposes the caching decisions to users of the cache. Open caching allows both clients and servers to customize the caching infrastructure. Customization is performed through object-orientation of resources.

Push-caching [11] and server dissemination [2] are server-driven techniques for caching of web objects. An origin server contacts caches, performs the tasks of locating objects, maintains concurrency and removes objects from caches. Push caching allows servers to set policies for objects, but requires the server to negotiate resources with caches and maintain state about which cache maintains copies of objects.

Active Cache [5] allows Java programs to be executed in the cache. The objective of this scheme is to be able to cache dynamic objects within caches. As a result, this scheme does not provide the same abilities to modify the behavior of the caching policies.

7 Summary

Caching systems are becoming common. However, caches are often limited in the policies that can be applied to cached web objects. This paper describes an infrastructure that provides customizable and dynamic policies for web objects.

We have designed an infrastructure that allows caches, web authors, and clients to customize policies. We have used this design to simulate a web cache and used traces from the DEC Squid cache and the NLANR cache infrastructure to assess dynamic and customizable caching policies. We have incorporated CacheL into Squid in order to assess the overhead of using an interpreted language to handle policies. Performance analysis shows that customized policies indeed allow caches to adapt to the requirements of clients, servers, and other caches. The overhead of adding customizable policies to a cache system is moderate.

Areas of future work include: efficiency, and a toolkit to provide several different policies already created to simplify the work of cache administrators and web authors.

Availability

More information on this project can be found by visiting the CacheL web page (`http://pdclab.cs.ucdavis.edu/qosweb/CacheL/`). This page includes a formal description of the CacheL grammar, script files used in running the experiments in this paper, source code for the DavisSim cache simulator, and patches that allow the embedding of our infrastructure within Squid.

Acknowledgements

We gratefully acknowledge Earl Barr, Brant Hashii, Peter Honeyman, Scott Malabarba, and the anonymous reviewers for their valuable suggestions. Thanks also to Tom Kroeger, Jeff Mogul, Carlos Maltzahm, and Digital Equipment Corporation for making the DEC logs available. We thank Duane Wessels at NLANR for making the NLANR cache logs available. The NLANR data used in some simulations were collected by the National Laboratory for Applied Networks Research under National Science Foundation grants NCR-9616602 and NCR-9521745.

References

[1] J. Fritz Barnes and Raju Pandey. `CacheL`: Language support for customizable caching policies. In *Fourth International WWW Caching Workshop*, San Diego, CA, USA, 31 March–2 April 1999.

[2] Azer Bestavros. WWW traffic reduction and load balancing through server-based caching. *IEEE Concurrency*, 5(1):56–67, January–March 1997.

[3] Lee Breslau, Pei Cao, Li Fan, Graham Phillips, and Scott Shenker. Web caching and Zipf-like distributions: Evidence and implications. In *Proceedings of IEEE INFOCOM '99, the Conference on Computer Communications*, New York, NY, USA, 21–25 March 1999.

[4] Pei Cao and Sandy Irani. Cost-aware WWW proxy caching algorithms. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, pages 193–206, Monterey, CA, USA, 8–11 December 1997.

[5] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the web.

In *Middleware '98. IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing*, pages 373–88, Lake District, UK, 15–18 September 1998.

[6] Vincent Cate. Alex – a global filesystem. In *USENIX File Systems Workshop Proceedings*, pages 1–12. USENIX, May 1992.

[7] Steve J. Caughey, David B. Ingham, and Mark C. Little. Flexible open caching for the web. In *Proceedings Sixth International World-Wide Web Conference*, volume 29(8–13) of *Computer Networks and ISDN Systems*, pages 1007–17, Santa Clara, California, USA, 7–11 April 1997.

[8] Anawat Chankhunthod, Peter B. Danzig, Chuck Neerdaels, Michael F. Schwartz, and Kurt J. Worrell. A hierarchical Internet object cache. In *Proceedings of the USENIX 1996 Annual Technical Conference*, pages 153–63, 22–26 January 1996.

[9] Li Fan, Pei Cao, Jussara Almeida, and Andrei Z. Broder. Summary cache: A scalable wide-area web cache sharing protocol. In *ACM SIGCOMM '98 Conference. Applications, Technologies, Architectures, and Protocols for Computer Communication*, volume 28(4) of *Computer Communication Review*, pages 254–65, Vancouver, BC, Canada, 2–4 September 1998.

[10] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. RFC 2068, UC Irvine, Digital Equipment Corporation, M.I.T., January 1997.

[11] James S. Gwertzman and Margo Seltzer. The case for geographical push-caching. In *Proceedings Fifth Workshop on Hot Topics in Operating Systems (HotOS-V)*, pages 51–5, Orcas Island, WA, USA, 4–5 May 1995.

[12] Terence P. Kelly, Yee Man Chan, Sugih Jamin, and Jeffrey K. MacKie-Mason. Biased replacement policies for web caches: Differential quality-of-service and aggregate user value. In *Fourth International WWW Caching Workshop*, San Diego, CA, USA, 31 March–2 April 1999.

[13] Thomas Kistler and Hannes Marais. WebL – a programming language for the web. In *Seventh International World Wide Web Conference*, volume 30(1–7) of *Computer Networks and ISDN Systems*, pages 259–70, Brisbane, Qld., Australia, 14–18 April 1998.

[14] P. Krishnan and Binay Sugla. Utility of cooperating Web proxy caches. In *Seventh International World Wide Web Conference*, volume 30(1–7) of *Computer Networks and ISDN Systems*, pages 195–203, Brisbane, Qld., Australia, 14–18 April 1998.

[15] Ari Luotonen and Kevin Altis. World-wide web proxies. In *First International Conference on the World-Wide Web*, volume 27(2) of *Computer Networks and ISDN Systems*, pages 147–54. Elsevier Science BV, 1994. Available from: `http://www.cern.ch/PapersWWW94/luotonen.ps`.

[16] Mark Nottingham. Optimizing object freshness controls in web caches. In *Fourth International WWW Caching Workshop*, San Diego, CA, USA, 31 March–2 April 1999.

[17] Alex Rousskov and Duane Wessels. Cache digests. In *Third International WWW Caching Workshop*, Manchester, UK, 15–17 June 1998.

[18] NLANR hierarchical caching system usage statistics. `http://www.ircache.net/Cache/Statistics/`.

[19] Renu Tewari, Michael Dahlin, Harrick M. Vin, and Jonathan S. Kay. Beyond hierarchies: Design considerations for distributed caching on the internet. Technical Report TR98-04, The University of Texas at Austin, 1998.

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

[21] Duane Wessels. Intelligent caching for world-wide web objects. In *Proceedings INET '95*, Honolulu, HI, USA, 27–30 June 1995.

[22] Duane Wessels and K. Claffy. ICP and the squid web cache. *IEEE Journal on Selected Areas in Communication*, 16(3):345–357, April 1998. Available from: `http://ircache.nlanr.net/~wessels/Papers/`.

[23] Stephen Williams, Marc Abrams, Charles R. Standridge, Ghaleb Abdulla, and Edward A. Fox. Removal policies in network caches for world-wide web documents. In *ACM SIGCOMM '96 Conference Applications, Technologies, Architectures, and Protocols for Computer Communications*, volume 26(4) of *Computer Communications Review*, pages 293–305, Stanford, CA, USA, 26–30 August 1996. ACM.

[24] Roland P. Wooster and Marc Abrams. Proxy caching that estimates page load delays. In *Sixth International World Wide Web Conference*, volume 29(8-13) of *Computer Networks and ISDN Systems*, pages 977–86, Santa Clara, CA, USA, 7–11 April 1997. Elsevier.

[25] Lixia Zhang, Scott Michel, Khoi Nguyen, Adam Rosenstein, Sally Floyd, and Van Jacobson. Adaptive web caching: Towards a new caching architecture. In *Third International WWW Caching Workshop*, Manchester, UK, 15–17 June 1998.