Computación y Sistemas Vol. 8 Núm. 1, pp. 001-014
© 2004, CIC-IPN, ISSN 1405-5546, Impreso en México

An Adaptive Coherence-Replacement Protocol for Web Proxy Cache Systems

Un Protocolo de Reemplazo y Coherencia Adaptativo para Sistemas de Manejo de Caches-Proxy en la Web

Jose Aguilar [1] y Ernst L. Leiss [2]
[1] CEMISID, Departamento de Computación
Facultad de Ingenieria, Universidad de los Andes
Mérida, Venezuela 5101
aguilar@ing.ula.ve
[2] Department of Computer Science
University of Houston
Houston, TX 77204-3475, USA
coscel@cs.uh.edu

Article received on June 28, 2002; accepted on June 30, 2004

Abstract

As World Wide Web usage has grown dramatically in recent years, so has grown the recognition that Web caches (especially proxy caches) will have an important role in reducing server loads, client request latencies, and network traffic. In this paper, we propose an adaptive cache coherence-replacement scheme for web proxy cache systems that is based on several criteria about the system and applications, with the objective of optimizing the distributed cache system performance. Our coherence-replacement scheme assigns a replacement priority value to each cache block according to a set of criteria to decide which block to remove. The goal is to provide an effective utilization of the distributed cache memory and a good application performance.
Keywords: Web Caching, Web caching performance, Replacement techniques, Coherency techniques.

Resumen

Como el uso de Internet se ha desarrollado dramáticamente en los últimos años, se ha reconocido que las caches en la Web (especialmente las Cache-Proxy) tienen un importante rol para reducir las cargas de los servidores, las latencias de los requerimientos de los clientes y el tráfico en la red. En este articulo, nosotros proponemos un esquema de reemplazo y coherencia adaptativo de caches para los Sistemas de Manejo de los Cache-Proxy en la Web que esta basado en varios criterios sobre el sistema y las aplicaciones, con el objetivo de optimizar el rendimiento del Sistema Distribuido de Caches. Nuestro esquema de reemplazo y coherencia asigna un valor de prioridad de reemplazo a cada bloque de cache según un conjunto de criterios para decidir cuales bloques eliminar. El objetivo es proveer una eficiente utilización de la memoria cache distribuida y un buen rendimiento para las aplicaciones.
Palabras Clave: Rendimiento de las Cache en la Web, técnicas de Reemplazo, técnicas de Coherencia.

1 Introduction

One of the major challenges associated with the Internet is the problem of increased response time caused due to the ever – increasing traffic on the Internet [1, 3, 9, 20, 22]. Many solutions, both hardware and software, have been suggested to overcome this challenge. The popular hardware solutions are to increase the bandwidth of the connection and to replicate the web documents at many locations. Increasing the bandwidth will increase the data transfer rate, and hence decrease the response time. The replication of documents will facilitate the nearest document to be fetched, minimizing the response time.

Recently, much research has focused on improving Web performance by reducing the bandwidth consumption and WWW traffic [3, 4, 7]. It means that fewer requests and responses need to go over the network and fewer request for a server to handle. Despite the fact that there have been great efforts for this purpose the results are not sufficient. The most popular software solution to the problem of increased response time is web caching. Web Caching is the technique of locally storing the frequently requested documents so that repeated requests to the same document can be serviced from the cache itself instead of fetching the document from the remote server. This will considerably decrease the response time involved. Several studies have presented the Web caching as the most beneficial solution for Web performance improvement. Web caching systems can lead to significant bandwidth savings, higher content availability, reducing client latency and increasing server's scalability and availability.

There are different architectures for web caching [11, 22]. These architectures include proxy caching, cooperative caching, adaptive caching, push caching and active caching. Proxy caching has become a well-established technique for enabling effective file delivery within the WWW architecture. One drawback of caching is the potential of using an out-of-date object stored in a cache instead of fetching the current object from the origin server. Many studies have examined policies for cache replacement for Web Proxy Cache Systems; however, these studies have rarely taken into account the combined effects of cache replacement and coherence policies [1, 9, 10].

In this paper we propose an adaptive cache coherence-replacement scheme for web proxy cache systems, and we analyze its performance in terms of hit – ratio and the total delay incurred in servicing the requests. Our approach combines classical coherence protocols (write-update and write-invalid protocols) and replacement policies (LRU, LFU, etc.) to optimize the overall performance (based on criteria such as network traffic, application execution time, data consistence, etc.). The cache coherence mechanism is responsible for determining whether a copy in the distributed cache system is stale or valid. At the same time, it must update the invalid copies when a given site requires a block. As a cache server has a fixed amount of storage, when this storage space becomes full, the cache server must choose a set of objects (or a set of victim blocks) to evict to make room for newly requested objects/blocks. An adaptive replacement mechanism is used for this task. Our approach attempts to improve the performance of the distributed cache memory system by assigning a replacement priority value to each cache block according to a set of criteria to select the block/object to remove (the state of the cache block, etc.). In addition, our scheme uses an adaptive replacement strategy that looks at the information available (reference history, access frequency, objects/blocks size, etc.) to make the decision what replacement technique to use, without a proportional increase in the space/time requirements. The rest of the paper is organized as follows: first, the web caching, coherence and replacement problems are presented. Then, our general coherence-replacement mechanism for distributed web proxy cache systems is developed. Finally, some results are presented.

2 Theoretical Aspects

2.1 Web Caching

Caching is a technique intended to minimize the time involved in data access [3, 4, 7, 9, 11, 16, 20, 21]. The process of caching is described as follows: In response to a sequence of requests for data, a cache of requested data is stored in a fast access device. A cache table is maintained which associates the stored data with size and other parameters. These parameters are used to decide which data to replace and when such a situation arises. When a new request arrives a table lookup is performed on the cache table. If the requested data is not in the cache, it is made available from the actual storage device and is also stored in the cache. If the data is already stored in cache it is made available directly from the cache. If the available free memory in cache is not enough for caching new data, space is freed by evicting data from the cache. Replacement policies decide which data is to reside in cache memory. In addition, a cache must determine if it can service a request, and if so, if each object it provides is fresh. This is a typical question to be solve with a cache coherence mechanism.

Caching occurs at various levels. At the processor level a small amount of RAM, access to which is much faster than the main memory, is built within the processor. Instructions and data that are frequently referenced are stored within this cache. At disk level caching, a portion of main memory functions as cache for data residing in disk. At the web caching level, storage space for cache is reserved in secondary memory. Web documents that are frequently requested are cached locally within this disk cache memory. Web caching can be done at various levels [1,3], in browsers, in intermediate servers and in the web servers. The main characteristics that differentiate web caching from others are variable object size and larger time scales, which also necessitates the development of more sophisticated replacement-coherence policies.

In general, the cache is a software that is in charge of storing on disks, data elements that are accessed by a number of clients. Web caches have been proposed as a solution to the scalability problem [5, 6, 7, 9, 13, 18]. Web caches store copies of previously retrieved objects to avoid transferring those objects in response to subsequent requests. Web caches are located throughout the Internet, from the user's browser cache through local proxy caches and backbone caches, to the so-called reverse proxy caches located near the origin of the content. Client browsers may be configured to connect to a proxy server, which then forwards the request on behalf of the client. All Web caches must try to keep cached pages up to date with the master copies of those pages, to avoid returning stale pages to users. There are strong benefits for the proxy to cache popular requests locally. Users will receive cached documents more quickly. Additionally, the organization reduces the amount of traffic imposed on its wide-area Internet connection. There are different architectures for web caching [9, 22]. At the following we present the most important web caching architectures implemented in earlier research efforts.

- *Proxy caching:* A proxy cache server receives HTTP requests from clients for a web object and if it finds the requested object in its cache, it returns the object to the user without disturbing the upstream network connection or destination server. If it is not available in the cache, the proxy attempts to fetch the object directly from the object's home server. Finally the originating server, which has the object, gets it, possibly deposits it and returns the object to the user. The benefits of proxy caching are supposed to reduce network traffic and reduce average latency. Proxy caches are often located near network gateways to reduce the bandwidth required over expensive dedicated Internet connections. When shared with other users, the proxies serve many clients with cached objects from many servers. One disadvantage to this design is that the cache represents a single point of failure in the network. When the cache is unavailable , the network also appears unavailable to users. Furthermore, another drawback is that all user web browsers are manually configured to use the appropriate proxy cache. So, if the server is unavailable all of the users must reconfigure their browsers in order to use a different cache. A final issue related to the standalone approach is that there is no way to dynamically add more caches when needed. The Squid and the Microsoft Proxy Server are two proxies which are available as stand-alone systems.

- *Reverse Proxy Caching:* An interesting variation to the proxy cache approach is the notion of reverse proxy caching, in which caches deployed near the servers, instead of near the clients. This is an attractive solution for servers that expect a high number of requests and want to assure a high level of quality of service (QoS). Reverse proxy caching is a useful mechanism when supporting virtual domains mapped to a single physical site, which is an popular service for many different service providers.

- *Transparent Caching:* One of the main drawbacks of the proxy server approach is the requirement to configure web browsers. The architecture of transparent caching eliminates this handicap. Transparent caches work by intercepting HTTP requests and redirecting them to web cache servers or clusters. There are two ways to deploy transparent proxy caching: at the switch level and at the router level. Router-based transparent proxy caching uses policy-based routing to direct requests to the appropriate cache or caches. For example, requests from certain clients can be associated with a particular cache. In switch-based transparent proxy caching the switch acts as a dedicated load balancer. This approach is attractive because it reduces the overhead normally incurred by policy-based routing. Although it adds extra cost to the deployment, switches are generally less expensive than routers.

- *Cooperative Caching-Swalla architecture:* A different approach to improving Web access performance is presented in [6], by recognizing that processor utilization rather than network bandwidth is the bottleneck in Web sites accessing. This applies especially to sites making extensive use of requests for dynamic content. The solution is a distributed Web server, called Swalla, which cooperatively caches the results of requests. Swalla is a multi-threaded, distributed Web Server that runs on a cluster of workstations and shares cache information and cache data between nodes. The server saves the execution results of programs with dynamic content requests and stores information (meta-data) about the cached data in the cache directory. Each node communicates with each others to exchange cache data and meta-data.

- *Adaptive web caching:* Authors of [8] argued that an adaptive, highly scalable, and robust web caching system is needed to effectively handle the exponential growth and extreme dynamic environment of the World Wide Web. The system must evolve towards a more scalable, adaptive, efficient, and self-configuring web-caching system in order to effectively support the phenomenal growth in demand for web content on the Internet. The adaptive web caching system provides an effective evolutionary step towards the above goal. Adaptive caching consists of multiple, distributed caches which dynamically join and leave cache groups based on content demand. The general architecture of the envisioned adaptive web caching system would be comprised of many cache servers that self-organize themselves into a tight mesh of overlapping multicast groups and adapt themselves as necessary to changing conditions. This mesh of overlapping groups forms a scalable, implicit hierarchy that is used to efficiently diffuse popular web content towards the demand. There are two main components, which are the underlying communication paths between neighboring caches and the set of requests for data along paths. Adaptive caching uses the Cache Group Management Protocol (CGMP) and the Content Routing Protocol (CRP). CGMP specifies how meshes are formed and how individual caches join and leave those meshes. CRP is used to locate cached content from within the existing meshes.

- *Push Cashing:* The idea of having a server decide when and where to cache its documents, was introduced as push-cashing in [9]. The key idea behind this architecture is to keep cached data close to those clients requesting that information. Data is dynamically mirrored as the originating server identifies where requests originate. For example, if a traffic to a West Coast based site started to rise because of increasing requests from the coast, the West Coast site would respond by initiating an East Coast based cache. One main assumption of push cashing is the ability to launch caches that may cross administrative boundaries. Finally, push caching is targeted mostly at content providers, which will most likely control the potential sites at which the caches will be deployed.

- *Active caching*: An active cache scheme is proposed in [10] to support caching of dynamic contents at Web proxies. The growth of the Internet and the World Wide Web has significantly increased the amount of online information and services available to the general population of the society. The Active Cache is a scheme which migrates parts of server processing on each user request to the caching proxy in a flexible, on demand fashion via "cache applets". A cache applet is a server-supplied code that is attached with a URL or a collection of URLs. The code is typically written in a platform independent programming such as Java. Adaptive cache uses applets, located in the cache, to customize objects that could otherwise not cached.

2.1.1 Web Cache Performance

When a user requests a Web document, the request goes through a proxy. If the document is in the proxy cache (cache hit) the proxy can immediately respond to the client's respond. If the requested document is not found (a cache miss) the proxy then attempts to retrieve the document from another location such as a peer or parent proxy cache or the origin server. Once the copy of the document has been retrieved the proxy can complete its response to the client. If the document is cacheable (based on information provided by the origin server or determined from the URL) the proxy may decide to add a copy of the document to its cache. Unfortunately, recent results suggest that the maximum cache hit rate that can be achieved by any caching algorithm is usually no more than 40% to 50%. This means that one out of two documents can not be found in the cache. In this section we will refer to the main performance metrics and the main factors which affect the Web cache performance.

2.1.1.1 Performance Metrics

Several metrics are commonly used when evaluating web cache performance. There are several criteria to measure the performance of a cache replacement technique [1, 2]:

- *Miss rate:* number of miss references. It is the most popular measure of cache efficiency. A hit rate of 70 percent indicates that seven of every ten requests to the cache found the requested object.
- *Object Retrieval time:* It is especially of interest to end users. Latency is inversely proportional to object hit rate because a cache hit can be served more quickly than a request that must pass through the cache to an origin server and back.
- *CPU, I/O system and network utilization*: The fraction of total available CPU cycles or disk or memory or network bandwidth consumed by the replacement technique (update time overhead and execution time overhead).
- *Stale ratio*: The ratio of stale blocks/objects versus total size in the cache system.
- *Byte hit rate*: The number of bytes returned directly from the cache as a fraction of the total bytes resqueted. This measure is not often used in system architecture cache studies because the objects (cache lines) are of constant size, and therefore the byte hit rate is directly proportional to the object hit rate. It is interesting on the Internet because external network bandwidth is a limited and often expensive resource. A byte rate hit of 30 percent indicates that three of every ten bytes requested were returned from the cache, while 70 percent of all bytes returned to users were retrieved across the external network.

Although these measures are related, optimizing one measurement may not optimize another. For example an increase in byte hit rate does not mean that it will necessarily reduce the network traffic.

2.1.1.2 Performance Factors

Various factors affect Web cache performance. The behavior of the user population that request documents from the cache is characterized by the user-access pattern. If a user accesses a small number of documents most of the time then these documents are obvious candidates for caching. Use-access patterns are usually not static and this implies that an effective cache policy should not be static. There are cache replacement policies which decide which document to remove when the cache is full. The cache removal period dictates at what point in time may a document (or documents) be removed. A continuous removal period implies that documents will be removed when there is no space in the cache to hold the active document. The active document is the document currently being accessed. A fixed cache removal period indicates that documents will only be removed at the beginning of the removal period.

Cache size is another factor influencing cache performance. The larger the cache size is the more documents it can maintain and the higher the cache hit ratio is. But, cache space is expensive. Therefore, an optimal cache size involves a trade off between cache cost and cache performance. Document size is also associated with cache performance. Given a certain cache size, the cache can store more small sized documents or fewer large sized documents. Maximum cacheable document size is a user-defined factor that places a ceiling on the size of documents that are allowed to be stored in the cache. Furthermore, there are two others factors which are cooperation and consistency. Cooperation refers to the coordination of users requests among many proxy caches in a hierarchical proxy cache environment. Cache consistency refers to maintaining copies of documents in cache that are not outdated. There are also factors which indirectly affect proxy cache performance such as protection copyright, which increases the complexity of proxy cache design. Finally uncacheable documents are a potential concern [11].

2.2 Coherence Problem

Distributed cache systems provide decreased latency at a cost: every cache will sometimes provide users with *stale* pages. Every local cache must somehow update pages in its cache so that it can give users pages which are as fresh as possible. Indeed, the problem of keeping cached pages up to date is not new to cache systems: after all, the cache is really just an enormous distributed file system, and distributed file systems have been with us for years. In conventional distributed systems terminology, the problem of updating cached pages is called *coherence* [1, 5, 8, 9, 10, 14].

Specifically, the cache coherence problem consists of keeping a data element found in several caches current with each other and with the value in main memory (or local memories). That is, cache coherence is the problem of maintaining consistency among multiple copies of the cache memory in a multiprocessor system. A *cache coherence protocol* ensures the data consistency of the system: the value returned by a read must always be the last value written to that location. There are two classes of cache coherence protocols: write-invalidate and write-update. In a *write-invalidate* protocol, a write request to a block invalidates all other shared copies of that block. If a processor issues a read request to a block that has been invalidated, there will be a coherence miss. That is, in *write-invalidate* protocols whenever a processor modifies its cache block, a *bus invalidation signal* is sent to all other caches in order to invalidate their content. In a *write-update* protocol on the other hand, each write request to shared data updates all other copies of the block, and the block remains shared. That is, in *write-update* protocols a copy of the new data is sent to all caches that share the old data. Although there are fewer read misses for a write-update protocol, the write traffic on the bus is often so much higher that the overall performance is decreased. A variety of mechanisms have been proposed for solving the cache coherence problem. The optimal solution for a multiprocessor system depends on several factors, such as the size of the system (i.e., the number of processors), etc.

2.3 Replacement Policy Problem

Cache replacement algorithms play a central role in the design of any caching component. These algorithms usually maximize the cache hit ratio (the number of times that objects in the cache are referenced) by attempting to cache the data items which are most likely to be referenced in the near future. A replacement policy specifies which block should be removed when a new block must be entered into an already full cache; it should be chosen so as to ensure that blocks likely to be referenced in the near future are retained in the cache. The choice of replacement policy is one of the most critical cache design issues and has a significant impact on the overall system performance. Common replacement algorithms used with such caches are [1, 2, 4, 5, 7, 13, 15, 17]:

- *Least Recently Used (LRU):* Replaces/evicts the block/object in the cache that has not been used for the longest period of time. The basic premise is that blocks that have been referenced in the recent past will likely be referenced again in the near future (temporal locality). This policy works well when there is a high temporal locality of references in the workload. This policy uses a program's memory access patterns to guess that the block that is least likely to be accessed in the near future is the one that has been accessed least recently. There is a variant, called Early Eviction LRU (EELRU), proposed in [8]. EELRU performs LRU replacement by default but diverges from LRU and evicts pages early when it notes that too many pages are being touched in a roughly cyclic pattern that is larger than the main memory.
- *Least Frequently Used (LFU):* It is based on the frequency with which a block is accessed. LFU requires that a references count be maintained for each block in the cache. A block/object's referenced count is incremented by one with each reference to it. When a replacement is necessary, the LFU replaces/evicts the blocks/objects with the lowest reference count. The motivation for LFU and other frequency based algorithms is that the reference count can be used as an estimate of the probability of a block being referenced.
- *Least Frequently Used (LFU)-Aging*: The LFU policy can suffer from cache pollution (an effect of temporal locality): if a formerly popular object becomes unpopular, it will remain in the cache for a long time, preventing other newly or slightly less popular objects from replacing it. *LFU-Aging* addresses cache pollution when it considers both a block/object's access frequency and its age in cache. One solution to this is to introduce some form of reference count "aging". The average reference count is maintained dynamically (over all blocks currently in the cache). Whenever this average counts exceeds some predetermined maximum value (a parameter to the algorithm) every reference count is reduced. There is a variant, called LFU with Dynamic Aging (LFUDA), that uses dynamic aging to accommodate shifts in the set of popular objects. It adds a cache age factor to the reference count when a new object is added to the cache or when an existing object is re-referenced. LFUDA increments the cache ages when evicting blocks/objects by setting it to the evicted object's key value. Thus, the cache age is always less than or equal to the minimum key value in the cache.
- *Greedy Dual Size (GDS):* It combines temporal locality, size, and other cost information. The algorithm assigns a *cost/size* value to each cache block. In the simplest case the cost is set to 1 to maximize the hit ratio, but costs such as latency, network bandwidth can be explored. GDS assigns a key value to each object. The key is computed as the object's reference count plus the cost information divided by its size. The algorithm takes into account recency for a block by inflating the key value (*cost/size* value) for an accessed block by the least value of currently cached blocks. The *GDS-aging* version adds the cache age factor to the key factor. By adding the cache age factor, it limits the influence of previously popular documents. The algorithm is simple to implement with a priority queue. There are several variations of the GDS algorithm each of which takes into account coherency information and the expiration time of the cache. *GDSlifetime* uses the remaining lifetime of a cached object in cache as part of the key value (*lifetime/size*) to lower the priority of cached blocks about to expire. The second variation uses the observation that different types of applications change their references at different rates. The *GDStype* cache replacement policy assigns different key values to different types of applications (for example, HTML and text applications change more frequently; they have a high priority (*2/size*), and one uses *1* for all other applications). A last GDS variation is *GDSlatency*, which uses as key value for an object the quantity *latency/size* where latency is the measured delay for the last retrieval of the object.
- *Frequency Based Replacement (FBR):* This is a hybrid replacement policy, attempting to capture the benefits of both LRU and LFU without the associated drawbacks. FBR maintains the LRU ordering of all blocks in the cache, but the replacement decision is primarily based upon the frequency count. To accomplish this, FBR divides the cache into three partitions: a new partition, a middle partition and an old partition. The new partition contains the most recent used blocks (MRU) and the old partition the LRU blocks. The middle section consists of those blocks not in either the new or the old section. When a reference occurs to a block in the new section, its reference count is not incremented. References to the middle and old sections do cause the reference counts to be incremented. When a block must be chosen for replacement, FBR chooses the block with the lowest reference count, but only among those blocks that are in the old section.
- *Priority Cache (PC):* Uses both runtime and compile-time information to select a block for replacement. PC associates a data priority bit with each cache block. The compiler, through two additional bits associated with each memory access instruction, assigns priorities. These two bits indicate whether the data priority bit should be set as well as the priority of the block, i.e., low or high. The cache block with the lowest priority is the one to be replaced.

In general, the policies anticipate future memory references by looking at the past behavior of the programs (program's memory access patterns). Their job is to identify a line/block (containing memory references) which should be thrown away in order to make room for the newly referenced line that experienced a miss in the cache.
Many cache replacement policies for web caching have been described and analyzed, which are aimed at reducing the cache hit – ratio. Site – based LRU (SBLRU) approach as proposed by Wong and Yeung [20] is aimed at designing fast proxy servers by reducing the CPU load. In this policy only the site name, instead of the document name, is considered in making caching decisions. They have shown that this approach will not degrade the cache performance when compared to LRU policy. Rizzo and Vicisano [16] have presented a policy called the Lowest Relative Value (LRV), which selects for replacement the document with the lowest relative value as computed by a cost/benefit model. The value associated with each document as proposed by them has a strong correlation to the probability with which the document will be accessed in the future. Cao and Irani [4] proposed the Greedy Dual – Size (GDS) policy, which takes size and a cost function for retrieving objects from the servers as the deciding factor in cache replacements. The document with the minimum value for the ratio (retrieval cost/size) is chosen for replacement. Dilley and Arlitt [7] have designed two replacement polices LFU with Dynamic Aging (LFUDA) and GDS-Frequency (GDSF) which are variants of LFU and GDS respectively. The LFUDA introduces an aging factor to prevent previously popular documents from polluting the cache. The GDSF policy is optimized to keep the smaller more popular documents in cache to maximize the object hit – ratio.

3 Our Adaptive Coherence-Replacement Policy

The FIFO and LRU policies as well as others such as LFU, etc. were developed for processor or disk level caching . The underlying assumption in these policies is that every document incurs the same amount of delay to be fetched, which is valid for processor or disk level of caching. But in the case of web caching this basic assumption does not hold true [4]. This is because in the case of web documents the remote web servers in which the documents are stored may be located at widely separated geographical locations [7]. This requires that the replacement policies for web caching should include this factor of varying delay in fetching a document over the Internet as a major parameter. Recent studies on web workload have shown tremendous breadth and turnover in the popular object set-the set of objects that are currently being accessed by users. The popular object set can change when new objects are published, such as news stories or sports scores, which replace previously popular objects. We should define cache replacement policies based on these ideas. In addition, a cache must determine if it can service a request, and if so, if each object it provides is fresh. This is a typical question to be solved with a cache coherence mechanism. If the object is fresh, the cache provides it directly, if not, the cache requests the object from its origin server.

3.1 The Model

Our adaptive coherence-replacement mechanism for Web caches is based on systems like Squid [18], which caches Internet data. It does this by accepting requests for objects that people want to download and by processing their requests at their sites. In other words, if users want to download a web page, they ask Squid to get the page for them. Then Squid connects to the remote server and requests the page. It then transparently streams the data through itself to the client machine, but at the same time keeps a copy. The next time someone wants that same page, Squid simply reads it from its disks, transferring the data to the client machine almost immediately (Internet caching). Normally, in Internet caching cache hierarchies are used. The general procedure in an Internet caching system is [18]:

1. The cache system gets a query from some machine
2. It checks to see if the object is on its disk
3. If it is, then it checks if the document is up-to-date, if so, it proceeds to feed it back to the client
4. If the object isn't there or is out-of-date, it checks to see if its neighbors have the object required.
5. The neighboring servers then check if they have the object on their disks
6. The original machine waits for the answers and then decides which cache it should get the object from

The Internet Cache Protocol (ICP) describes the cache hierarchies. The ICP's role is to provide a quick and efficient method of intercache communication, offering a mechanism for establishing complex cache hierarchies. ICP allows one cache to ask another if it has a valid copy of a object. Squid ICP is based on the following procedure [18]:

1. Squid sends an ICP query message to its neighbors (URL requested)
2. Each neighbor receives its ICP query and looks up the URL in its own cache. If a valid copy exists, the cache sends ICP_HIT, otherwise ICP_MISS
3. The querying cache collects the ICP replies from its peers. If the cache receives several ICP_HIT replies from its peers (neighbors), it chooses the peer whose reply was the first to arrive in order to receive the object. If all replies are ICP_MISS, Squid forwards the request to the neighbors of its neighbors, until to find a valid copy.

Neighbors refer to other caches in a hierarchy (a parent cache, a sibling cache or the origin server). Squid offers numerous modifications to this mechanism, for example:

- Send ICP queries to some neighbors and not to others
- Include the origin sever in the ICP "ping" so that if the origin servers reply arrives before any ICP-hits, the request is forward there directly.
- Disallow or require the use of some peers for certain requests.

Our model is defined according to the following ideas:

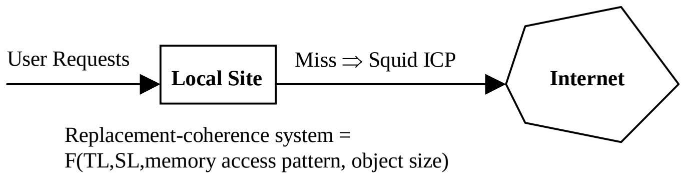
User Requests
Local Site
Miss ⇒ Squid ICP
Internet
Replacement-coherence system =
F(TL,SL,memory access pattern, object size)

Fig. 1. The Replacement-Coherence System

That is, our Replacement-Coherence System depend on TL, SL, memory access pattern and object size. TL to guarantee keep the objects currently used. SL to guarantee keep the neighbors of the objects currently used. Memory access pattern to know the behavior of the users and in this way if the TL is a valid parameter. Finally, object sizes to minimize the communication cost (we prefer to keep large objects). These parameter are using according to different priorities: TL has the largest priority, then the object size and finally SL. The general procedure of our adaptive cache coherence-replacement mechanism is as follows:

```
1. If read miss then
    1.1  Search for a valid copy (using the Squid ICP).
    1.2 If cache is full, call the replacement-coherent system.
    1.3 Receive a valid copy
    1.4 Read block
1.  If read hit then
    1.1 Read block
```

3.2 The Replacement-Coherent System

Normally, user cache access patterns affect cache replacement decisions while block characteristics affect cache coherency decisions. Therefore, it is reasonable to consider replacing cache blocks that have expired or are closed to expiring because their next access will result in an invalidation message. In this way, we propose a cache coherence-replacement mechanism that incorporates the state information into an adaptive replacement policy. The basic idea behind the proposed mechanism is to combine a coherence mechanism with our adaptive cache replacement algorithm. Our adaptive cache coherence-replacement mechanism exploits semantic information about the expected or observed access behavior of particular data shared, and the replacement phase employs several different mechanisms (for example, LRU, LFU or GDS mechanisms), each one appropriate for a different situation. Since our coherence-replacement is provided in software, we expect the overhead of providing our mechanism to be offset by the increase in performance that such a mechanism will provide. That is, in our approach we examine if the overall performance can be improved by considering coherency issues as part of the cache replacement decision. We incorporate the additional information about a program's characteristics, which is available in the form of the cache block states, in our replacement system. In addition, when several blocks are in the same state, we define a set of parameters that we can use to select the best replacement policy in this dynamic environment, These parameters are:

A) Information about the system
- Workload, Bandwidth, Latency, CPU Utilization.
- Type of system (Shared memory, etc.)

B) Information about the application
- Information about the data and cache block or objects
  - + Frequency
  - + Age
  - + Size
  - + Length of the past information (patterns)
  - + State (invalid, shared, etc.)
- Type an degree of access pattern on the system
  - + High or low spatial locality (SL)
  - + High or low temporal locality (TL)

An optimal cache replacement policy would know the future workload. In the real world, we must develop heuristics to approximate ideal behavior. Our system uses a *write-invalidate* coherent protocol. In this case, each cache block is in the following state:

Invalid: a stale copy.
Valid: an up to date copy

In addition, the coherence-replacement policy defines one expression to fix the priority of replacement of each block/object. According to this value, the system chooses the block with higher priority to replace. The priority is defined as:

1. Blocks with stale copies (invalid state) have the highest priority to be chosen to replace.
2. Otherwise, blocks in valid states must be chosen to replace,

If there are several valid blocks with the highest priority, we use the replacement policy specified by our *decision system* [1, 2] to choose the block to be replaced. The *decision system* is composed of a set of rules to decide the replacement policy to use. Each rule selects a replacement policy to apply according to different criteria:

```
If TL is high and the system's memory access pattern is regular then
        Use a LRU replacement policy
If TL is low and the system's memory access pattern is regular then
        Use a LFU replacement policy
If we require a precise decision using a large system's memory access pattern history then
        Use a Prediction replacement policy
If objects/blocks have variable sizes then
        Use a GDS replacement policy
If a fast decision is required then
        Use a RAND replacement policy
If there is a large number of LRU candidate blocks then
        Use a FBR replacement policy
If SL is high then
        Use a hybrid FBR + GDS replacement policy
If the system's memory access pattern is irregular then
        Use an age replacement policy
```

The different parameters of the rules are calculated using the information that the operating systems have about the utilization of the memory, CPU, etc.

4 Performance Comparision

We constructed a trace-driven simulation to study our approach using a set of client traces from Digital Equipment Corporation [9]. These traces are distinguished from many proxy logs in that they contain last modification time. The simulation model consisted of three entities: web servers, web browsers and proxy servers. The simulations were performed at different network loads but all the proxy servers received the same sequence of requests.

In the following discussions we compare the performance of our approach (AA) against others policies (DSLRU [15], LRU, and an approach proposed in [9]). We compared those policies based on different metrics: the hit – ratio, the total delay involved in servicing a sequence of requests, the response latency, the bandwidth and number of request. For the first case, we use a normalized cost model for each of these criteria where each of these costs is defined 0 if a "get request" can be retrieved from the proxy cache, or 1 for a "get request" to a server. The total cost for a simulation is the average of these normalized costs. Figure 2 shows the average costs of the best policy proposed on [9], LRU, DSLRU and our work. LRU has the highest cost. For a 10 GB cache, the cost saving is 4%. Our results indicate that for caches where the cache space is small, the cache replacement policy primarily determines the costs. For cache operating in configurations with large amounts of cache space, the cache coherency policy primarily determines the overall costs. To reduce the overhead of our approach, we have included an appropriate inclusion of coherency characteristics on the replacement policy.

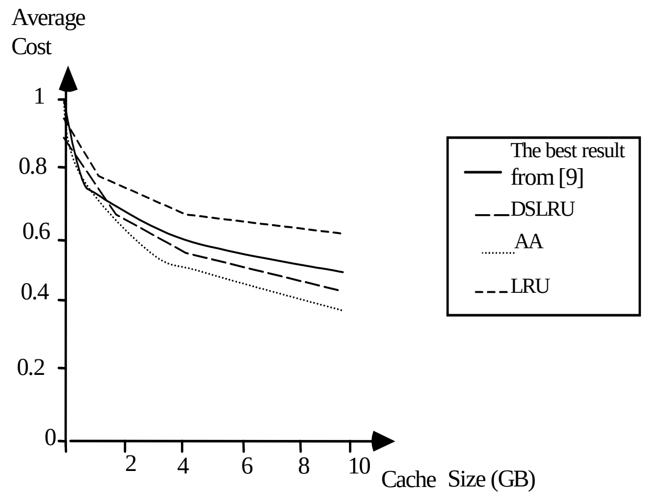

Fig. 2. Average Cost vs. Cache Size

The graph in Figure 3 shows the results in terms of the total number of hits obtained by simulations of the policies for different sequences of requests. Of the policies our model has the highest hit – ratio, followed by LRU, DSLRU and [9]. This is well evident from the factor that LRU purely tries to maximize the hit – ratio while the rest completely disregards this. These simulations gave us an insight into a subtle demerit in the [9] approach. The aim in proposing the [9] policy was to reduce the communication delay ratio. In doing so the hit –ratio decreased as evident by the simulation results. A decrease in hit–ratio indicates that more number of requests are directed to the web servers which in turn means that the those documents will have to be stored in the cache. This ultimately leads to an increase in I/O activity. The Table 1 shows the hit–ratios for these polices. It shows a difference in hit–ratio between AA and the rest of policies as the number of requests increase.

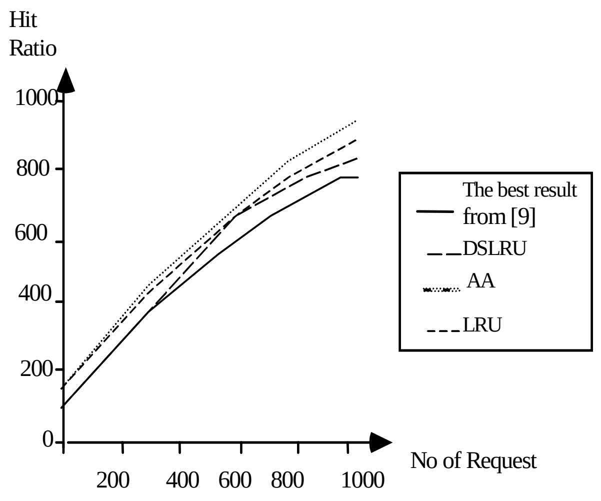

Fig. 3. Number of hits vs. Number of Requests

Number of Requests	DSLRU	[9]	LRU	AA
100	0.66	0.65	0.70	0.70
200	0.74	0.73	0.75	0.75
400	0.75	0.67	0.79	0.82
800	0.74	0.68	0.80	0.85

Table 1: Comparison of Hit – ratios

The simulations reveal the significance of AA policy. The total delay in fetching the documents increases at a faster rate as the number of requests increases for LRU policy, whereas it increases at a slower rate for AA and [9]. The graph of Figure 4 shows that there is a wide margin between AA and LRU when the number of requested documents is high. The marginal difference between AA and the rest of policies is due to the increased hit–ratio in AA. Thus, the simulation studies confirm that the use of delay sensitive polices for replacement in web caches is necessary.

5 Conclusion

Web caching is the best solution to reduce the Internet traffic and bandwidth consumption. It is also a low cost technique for improving the Web latency. Nowadays, proxy caches are increasingly used around the world to reduce bandwidth and make less severe delays associated with delays. Web proxy servers sharing their cache directories through a common mapping service that can be queried with at most one message exchange. By considering that it is useful to be able to assess the performance of proxy caches, we have presented a coherent-replacement protocol. Our approach includes additional information/factors such as frequency of block use, state of the blocks, etc., in replacement decisions. It takes into consideration that coherency and replacement decisions affect each other. This adaptive policy system has been validated by experimental work. Our majors results are: a) cache replacement and coherency are both important in reducing the costs for a proxy cache, b) direct inclusion of cache coherency reduces the overhead of our approach and guarantees a better performance.

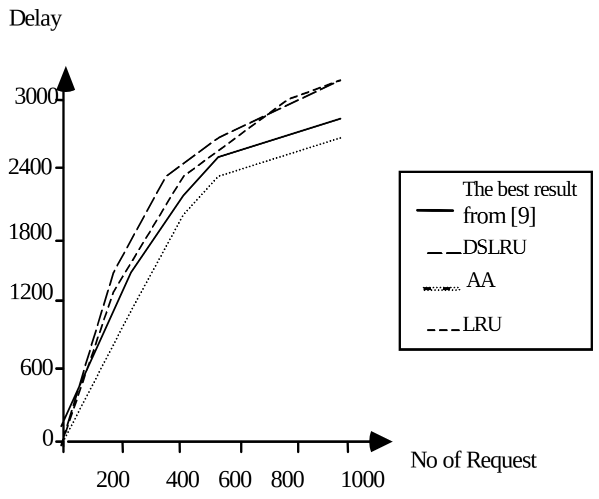

Fig. 4. Delay vs Number of Requests

Acknowledgement

This work was partially supported by a CONICIT-Venezuela grant (subprograma de pasantías postdoctorales)

References

[1] **J. Aguilar, E. Leiss**, "A Web Proxy Cache Coherency and Replacement Approach", *Lecture Notes in Computer Science*, Springer-Verlag, Vol. 2198, pp. 75-94, 2001.
[2] **J. Aguilar, E. Leiss**, “A Dynamic/Adaptive Cache Replacement Algorithm", *Proceeding of the XXVII Latinoamerican Informatics Conference. Mérida, Venezuela*.pp. 123-135, September 2001.
[3] **G. Barish, K. Obraczka**, World Wide Web caching: trends and techniques, IEEE Communications magazine, Volume 38, Issue 5,2000, pp. 178-185.
[4] **P. Cao and S. Irani**, “Greedy Dual – Size:A cost aware WWW proxy caching algorithm,” in Proc. 2nd Web Caching Workshop, Boulder, CO, June 1997.
[5] **S. Cho, J. King, G. Lee**, “Coherence and Replacement Protocol of DICE-A Bus Based COMA Multiprocessor”, *Journal of Parallel and Distributed Computing*, Vol. 57, pp. 14-32, 1999.
[6] **L. Choi *et al.***, "Techniques for compiler-directed Cache Coherence". *IEEE Parallel Distributed Technology, Winter 1996.*
[7] **J. Dilley, M. Arlitt,** “Improving Proxy Cache Performance: Analysis of Three Replacement Policies”, *IEEE Internet Computing, November,* pp. 44-50, 1999.
[8] **B. Krishnamurthy, C. Wills**, "Piggyback Server Invalidation for Proxy Cache Coherency", *Proc. 7th Intl. World Wide Web Conf.*, pp. 185-193, 1998.
[9] **B. Krishnamurthy, C. Wills**, “Proxy Cache Coherency and Replacement-Towards a More Complete Picture*”, IEEE Computer,* Vol. 6, pp. 332-339, 1999.
[10] **C. Liu, P. Cao**, "Maintaining Strong Cache Consistency in the WWW*", Proc. 17th IEEE Intl. Conf. on Distributed Computing Systems*, 1997.
[11] **J. Menaud, V. Issarny, M. Bantre**, Improving effectiveness of Web caching, In Springer Verlag, editor, Recent Advances in Distributed Systems, volume LNCS 1752, 2000.
[12] **M. Obaidat, H. Khalid**, “Estimating NN-Based Algorithm for Adaptive Cache Replacement*”, IEEE Transaction on System, Man and Cybernetic*, Vol. 28, N. 4, pp. 602-611, 1998.
[13] **L. Rizzo, L. Vicisano**, “Replacement policies for a proxy cache,” IEEE Internet Computing Nov/Dec 1999.

[14] **H.Sandhu, K.Sevcik**; "An Analytic Study of Dynamic Hardware and Software Cache Coherence Strategies"; *Proc.1995 ACM SIGMETRICS Intl. Conf. on Measurement and Modeling of Computer Systems , pp. 167 - 177, 1995.*

[15] **S. Selvakumar* J. Smith**, Delay Sensitive LRU Policy for Replacement in Web caches. . *Proceeding of the World Multiconference on Systemics, Cybernetics and Informatics, (Ed. N. Callaos et al.),* International Institute of Informatics and Systemics, Vol. VII, pp. 336-341, Orlando, USA, Julio 2001.

[16] **J. Shim, P. Scheuermann, R. Vingralek**, "Proxy Cache Design: Algorithms, Implementation and Performance", *IEEE Trans. on Knowledge and Data Engineering*, 1999.

[17] **Y. Smaradakis, S Kaplan, P. Wilson**, "EELRU: Simple and Effective Adaptive Page Replacement", *Performance Evaluation Review*, Vol. 27, N, 1, pp. 122-133, January 1999.

[18**] Squid Internet object cache**. http://squid.nlanr.net/Squid.

[19] **G. Tyson, M. Fonrens, J. Matthews and A. Pleczkun**, "Managing Data Caches Using Selective Cache Lien Replacement", *International Journal of Parallel Programming*, Vol. 25, N. 3, pp. 213-242, 1997.

[20] **A. Vakali G. Pallis**, A Study on Web Caching Architectures and Performance. . *Proceeding of the World Multiconference on Systemics, Cybernetics and Informatics, (Ed. N. Callaos et al.),* International Institute of Informatics and Systemics, Vol. VII, pp. 309-314, Orlando, USA, Julio 2001.

[21] **E. Watson, Y. Shi Y. Chen**, A user-access model-driven approach to proxy cache performance analysis, Decision Support Systems, Volume 25, Issue 4,May 1999, pp. 309-338.

[22] **B. Williams**, "Transparent Web Caching Solutions," in Proc. 3 rd Intl. WWW Conference.

[23] **C. Wills, M. Mikhailov**, "Towards a better Understanding of Web Resources and Server Responses for Improved Caching", *Proc. 8th Intl. World Web Conf.*, 1999.

José Aguilar. *He was born in Valera-Venezuela. He received the B. S. degree in System Engineering in 1987 from the Universidad de los Andes-Merida-Venezuela, the M. Sc. degree in Computer Sciences in 1991 from the Universite Paul Sabatier-Toulouse-France, the Ph. D degree in Computer Sciences in 1995 from the Universite Rene Descartes-Paris-France and a Postdoctoral Research Fellow in 2000 from the Department of Computer Sciences at the University of Houston. He is a Full Professor in the Department of Computer Science at the Universidad de los Andes and researcher of the Center of Microcomputation and Distributed Systems (CEMISID). He has been a visiting researcher at different French, US and Spanish Universities. He has chaired many conferences and coeditor of many proceedings. He has authored more than 150 research papers in international journals and conferences or technical reports, and he is editor/author of 8 technical books. His research interest is Intelligence Computation (evolutionary computation, neural network, logic fuzzy, multi-agent systems) and distributed/parallel systems (I/O management, performance optimization, task assignment and scheduling, etc.).*

Ernst L. Leiss. *He received graduate degrees in computer science and in mathematics from the University of Waterloo (Canada) and the Technical University of Vienna (Austria). He joined the Department of Computer Science at the University of Houston in 1979. He has lectured in 23 countries and has supervised 13 doctoral dissertations and approximately 100 M.S. theses. Dr. Leiss is author of about 140 peer-reviewed papers; he wrote Principles of Data Security (1982, Plenum), Software Under Siege: Viruses and Worms (1990, Elsevier), Parallel and Vector Computing: A Practical Introduction (McGraw-Hill, 1995), and Language Equations (Springer, 1999). He has contributed articles on data-security and on computer viruses to the Encyclopedia of Physical Science and Technology (1987 and 1990, Academic Press). His research interests range from high-performance computing to data security, databases, and theory of formal languages.*