A Web Proxy Cache Coherency and Replacement Approach

Jose Aguilar[1] and Ernst Leiss[2]

[1] CEMISID, Dpto. de Computacion, Facultad de Ingenieria, Universidad de Los Andes, Merida 5101, Venezuela
aguilar@ing.ula.ve

[2] Department of Computer Science, University of Houston, Houston, TX 77204-3475, USA
coscel@cs.uh.edu

Abstract. 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.

1 Introduction

Many studies have examined policies for cache replacement and cache coherence; however, these studies have rarely taken into account the combined effects of policies [2, 6]. In this paper we propose an adaptive cache coherence-replacement scheme for web proxy cache systems. This work is based on previous work we have done on cache replacement mechanisms which have shown that adaptive cache replacement policies improve the performance of computing systems [1]. 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. Because a cache has a fixed amount of storage, when this storage space becomes full, the cache must choose a set of objects (or a set of victim blocks) to evict to make room for newly requested objects/blocks. The 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. To fix this priority, we take into account the state of the cache block. In addition, our scheme uses an adaptive replacement strategy that looks at the

N. Zhong et al. (Eds.): WI 2001, LNAI 2198, pp. 75–84, 2001.

information available to make the decision what replacement technique to use, without a proportional increase in the space/time requirements.

2 Theoretical Aspects

2.1 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* [2, 3, 5, 6, 8, 11, 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). 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 [14]: 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. 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. 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.

2.2 Replacement Policy Problem

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, 4, 7, 9, 10, 15]:

- *First In-First Out (FIFO):* this is the simplest scheme; it is easily managed with a FIFO queue. When a replacement is necessary the first block entered at the cache memory (at the head of the queue) must be removed.

- *Most Recent Used (MRU):* Replaces the block in the cache, which has been more recently used. This is not used frequently on cache memory system because it has bad temporal locality. It is a typical property of the memory reference patterns of processors, page reference patterns in virtual memory patterns, etc.
- *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. There is a variant, called Early Eviction LRU (EELRU), proposed in [7]. 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. In [7], Lee et al. show that there exists a spectrum of block replacement policies that subsumes both the LRU and LFU policies. The spectrum is formed according to how much more weight is given to the recent history over the older history and is referred to as the LRFU (Least Recently/Frequently Used) policy.
- *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.
- *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 ac-

count coherency information and the expiration time of the cache (*GDSlifetime*). The second variation uses the observation that different types of applications change their references at different rates (*GDStype*). 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.

3 An Adaptive Coherence-Replacement Policy

The growth of the Internet and the WWW has significantly increased the amount of online information and services available. However, the client/server architecture employed by the current Web-based services is inherently unscalable. Web caches have been proposed as a solution to the scalability problem [4, 5, 6, 8, 12, 16]. 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.

Because a cache server has a fixed amount of storage, the server needs a cache replacement mechanism [4, 6]. 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 [16]. 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 this workload characterization. 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. If the object is fresh, the cache provides it directly, if not, the cache requests the object from its origin server.

Our adaptive coherence-replacement mechanism for Web caches is based on systems like Squid [13], 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 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 [13]:

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.

In this case, each cache block is in the following state:

Invalid: a stale copy.

Normally, there is only one state because the users typically do not write. Then, the adaptive cache coherence-replacement mechanism is as follows:

1. If *read miss* then
   1.1 Search for a valid copy (using the ICP). A read-miss request is sent using the ICP
   1.2 If cache is full, choose a replacement policy according to a *decision system*
   1.3 Receive a valid copy
   1.4 Read block
2. If *read hit* then
   2.1 Read block

3.1 The Replacement 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 [1, 2]. Our adaptive cache coherence-replacement mechanism exploits semantic information about the expected or observed access behavior of particular data shared objects on the size of the cache items, and the replacement phase employs several different 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. Thus, we define a set of parameters that we can use to select the best replacement policy in a dynamic environment:

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)).

C) Other information
- Cache conflict resolution mechanism
- Pre-fetching mechanism

An optimal cache replacement policy would know the future workload. In the real world, we must develop heuristics to approximate ideal behavior. For each of the policies we discussed in section 2.2, we list the information that is required by them:

- LFU: reference count.
- LRU: the program's memory access patterns.
- Priority Cache: information at runtime or compile time (data priority bit by cache/block).
- Prediction: a summary of the entire program's memory access pattern.
- FBR: the program's memory access patterns and organization of the cache memory.
- MRU: the program's memory access patterns.
- FIFO: the program's memory access patterns.
- GDS: size of the objects, information to calculate the cost function, reference count.
- Aging approaches: GDS-aging: GDS age factor; LFU-aging: LFU age factor.

We define one expression, called the *key value*, to define the priority of replacement of each block/object. According to this value, the system chooses the block with higher priority to replace (low key value). The key value is defined as:

$$\text{Key-Value} = (CF+A+FC)/S + \text{cache factor} \qquad (1)$$

where, - FC is the frequency/reference count, that is the number of times that a block has been referenced,
- A is the age factor,
- S is the size of the block/object,
- CF is the cost function that can include costs such as latency or network bandwidth.

The first part of Equation (1) is typical for the GDS, LRU and LFU policies (using information about objects to reference and not about cache blocks). The cache factor is defined according to the replacement policy used:

- LFU: blocks with a high frequency count have the highest cache factor.
- LRU: the least recently used block has the highest cache factor.
- Priority Cache: defined at runtime or compile-time.
- Prediction: the least used block in the future has the highest cache factor.
- FBR: the least recently used block has the highest cache factor.
- MRU: the most recently used block has the highest cache factor.
- FIFO: the block at the head of the queue has the highest cache factor.
- GDS: not applicable.
- Aging approaches: FC/A, with a reset factor that restarts this value after a given number of ages or when the age average is more than a given value.

The coherence-replacement policy defines the cache factor so that: blocks in invalid state have the highest priority to be chosen to replace. Otherwise, blocks in shared states must be chosen to replace, then blocks in exclusive states, and finally, blocks in modified states. If there are several blocks in a particular state, we use the replacement policy specified in our *decision system* [1]. 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 *TL is low and the system's memory access pattern is large* then
    Use a MFU 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

4 Result Analysis

We constructed a trace-driven simulation to study our approach using a set of client traces from Digital Equipment Corporation [6]. We compare our approach with [6]. These traces are distinguished from many proxy logs in that they contain last modification time. We use four evaluation criteria: response latency, bandwidth, hit rates and number of request. 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 1 shows the average costs of the best policy proposed on [6] and of our work. The approach proposed on [6] 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 can make an appropriate inclusion of coherency characteristics on the replacement policy.

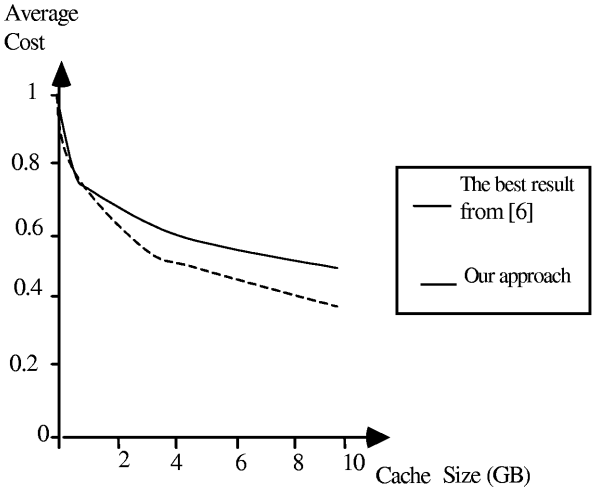

Fig. 1. Average Cost vs. Cache Size

5 Conclusions

The goal of this research was to formulate an overarching framework subsuming various cache management strategies in the context of different distributed platforms. We have proposed an adaptive coherence-replacement policy. 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 issues maybe can reduce the overhead of our approach but doesn't guarantee a better performance.

References

1. Aguilar J., Leiss E. A Proposal for a Consistent Framework of Dynamic/Adaptive Policies for Cache Memory Management, Technical Report, Department of Computer Sciences, University of Houston, (2000).

2. Cho S., King J., Lee G. Coherence and Replacement Protocol of DICE-A Bus Based COMA Multiprocessor, Journal of Parallel and Distributed Computing, Vol. 57 (1999) 14-32.
3. Choi L. Techniques for compiler-directed Cache Coherence. IEEE Parallel Distributed Technology, Winter 1996.
4. Dilley J., Arlitt M. Improving Proxy Cache Performance: Analysis of Three Replacement Policies, IEEE Internet Computing, November, (1999) 44-50.
5. Krishnamurthy B., Wills C. Piggyback Server Invalidation for Proxy Cache Coherency, Proc. 7th Intl. World Wide Web Conf., (1998) 185-193.
6. Krishnamurthy B., Wills C. Proxy Cache Coherency and Replacement-Towards a More Complete Picture, IEEE Computer, Vol. 6, (1999) 332-339.
7. Lee D., Choi J., Noh S., Cho Y., Kim J., Kim C. On the Existence of a Spectrum of Policies that Subsumes the Least Recently Used (LRU) and Least Frequently Used (LFU) Policies, Performance Evaluation Review, Vol. 27 (1999). 134-143.
8. Liu C., Cao P. Maintaining Strong Cache Consistency in the WWW, Proc. 17th IEEE Intl. Conf. on Distributed Computing Systems, (1997).
9. Mounes F., Lilja D. The Effect of Using State-based Priority Information in a Shared-Memory Multiprocessor Cache Replacement Policy, IEEE Computer, Vol. 2 (1998) 217-224.
10. Obaidat M., Khalid H. Estimating NN-Based Algorithm for Adaptive Cache Replacement, IEEE Transaction on System, Man and Cybernetic, Vol. 28 (1998) 602-611.
11. Sandhu H., Sevcik; K. 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.
12. Shim J., Scheuermann P., Vingralek R. Proxy Cache Design: Algorithms, Implementation and Performance, IEEE Trans. on Knowledge and Data Engineering, (1999).
13. Squid Internet object cache. http://squid.nlanr.net/Squid.
14. Stenstrom P. A Survey of Cache Coherence Schemes for Multiprocessors. IEEE Computer, (1990) 12-24.
15. G. Tyson, M. Fonrens, J. Matthews and A. Pleczkun, “Managing Data Caches Using Selective Cache Lien Replacement”, International Journal of Parallel Programming, (1997) 25(3) 213-242.
16. Wills C., Mikhailov M. Towards a better Understanding of Web Resources and Server Responses for Improved Caching, Proc. 8th Intl. World Web Conf., (1999).