A General Adaptive Cache Coherency-Replacement Scheme for Distributed Systems

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 distributed systems that is based on several criteria about the system and applications, with the objective of optimizing the distributed cache system performance. We examine different distributed platforms (shared memory systems, distributed memory systems, and web proxy cache systems) and the potential of incorporating coherency-replacement issues in the cache memory management system. 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

The performance of distributed caching mechanisms is an active area of research [2, 3, 4, 5, 7, 8, 10]. A distributed cache memory is the simplest cost-effective way to achieve a high-speed memory hierarchy. A cache provides, with high probability, instructions and data needed by the local CPU at a rate that is more in line with the local CPU's demand rate. 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, 5]. In this paper we propose an adaptive cache coherence-replacement scheme for distributed systems. 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.). 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]. 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. We study the impact of our scheme in different distributed systems: shared-memory systems, distributed-memory systems, and web proxy systems.

T. Böhme and H. Unger (Eds.): IICS 2001, LNCS 2060, pp. 116–125, 2001.

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, 4, 5, 7, 12].

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 [12]: 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. The main types of coherence mechanisms are [12]: Snooping Coherence, Directory Coherence, and Software Cache Coherence Mechanism.

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, 3, 6, 8, 9]: First In-First Out (FIFO), Most Recent Used (MRU), Least Recently Used (LRU), Least Frequently Used (LFU), Least Frequently Used (LFU)-Aging, Greedy Dual Size (GDS), Frequency Based Replacement (FBR), Random (RAND), Priority Cache (PC), Prediction.

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

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]. 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. 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. Our system can be applied to different distributed systems independent of the coherence protocol. We assume that if a place has an invalid copy of a block, a request to this block is a miss access. In this way, we guarantee to search for a valid copy of the block. We assume three types of target systems:

3.1 Cache Coherence-Replacement in a Shared Memory Multiprocessor

In a shared-memory multiprocessor system, *local caches* are used to reduce memory access latency and network traffic [8, 12]. Each processor is connected to a fast memory 'backed up' by a large (and slower) main memory. This configuration enables processors to work on local copies of main memory blocks, greatly reducing the number of memory accesses that the processor must perform during program execution. Although local caches improve system performance, they introduce the *cache coherence problem*: multiple cached copies of the same block of memory must be consistent at any time during a run of the system. In general, each cache block can be in one of the following four states:

Invalid: a stale copy.
Shared: multiple copies of the block exist.
Exclusive: only one processor has a copy of the block.
Modified: the processor has the only valid copy of the block in cache.

We can use both coherence protocols (write-invalid and write-update). In this case, our adaptive cache coherence-replacement mechanism is as follows:

1. If *write miss* (if the processor has a copy of the block, it is invalid) then
   1.1 Search for a valid copy (shared memory or remote cache memory). A *read-miss* request is sent to the system
   1.2 If cache is full, choose a replacement policy according to a *decision system*.
   1.3 Receive a valid copy
   1.4 Modify block (critical section)
   1.5 Call coherence protocol (write-invalidate or write-update protocol)
   1.6 Change state to modified (if we use write-invalidate protocol) or shared (if we use write-update protocol) or exclusive (if it has the only copy in the system)
   1.7 Change state of all other copies of this block to invalid (if we use write-invalidate protocol) or shared (if we use write-update protocol)
2. If *read miss* then
   2.1 Search for a valid copy (shared memory or remote cache memory). A *read-miss* request is sent to the system
   2.2 If cache is full, choose a replacement policy according to a *decision system*.
   2.3 Receive a valid copy
   2.4 Change state of the different copies of the block to shared or exclusive (if it has the only copy on the system)
   2.5 Read block
3. If *write hit* then
   3.1 Modify block (critical section)
   3.2 Call coherence protocol (write-invalidate or write-update protocol)
   3.3 Change state to modified (if we use write-invalidate protocol) or shared (if we use write-update protocol) or exclusive (if it has the only copy on the system)
   3.4 Change state of all other copies of this block to invalid (if we use write-invalidate protocol) or shared (if we use write-update protocol)
4. If *read hit* then
   4.1 Read block

3.2 Cache Coherence-Replacement in a Distributed Memory Multiprocessor

In this case, we study a distributed memory multiprocessor (DM) in which each processor is associated with a private cache. Local cache memory is the simplest cost-effective way to achieve a high-speed memory hierarchy in a distributed memory system. A local cache provides, with high probability, instructions and data needed by the local CPU at a rate that is more in line with the CPU's demand rate [2, 12]. In this case, each cache block can be in one of the following state:

Invalid: a stale copy.

Shared_up: multiple copies of the block exist and all memory copies are up-to-date.
Shared_nup: multiple copies of the block exist and not all memory copies are up-to-date.
Exclusive_up: only one processor has a copy of the block, and the local memory copy is up-to-date.
Exclusive_nup: only one processor has a copy of the block, and the local memory copy is not up-to-date.
Modified: the processor has the only valid copy of the block and all memory copies are stale.

In our approach, we assume a protocol which guarantees that the local memory has the same version of a given block as its cache memory. That means, the differentiation between Shared_up and Shared_nup is not necessary (only a Shared state is needed); similarly for Exclusive_up and Exclusive_nup, where only an Exclusive state is required. In addition, when we update the state of a block at a given site (according to the coherence protocol), both block copies are updated (local memory and its cache memory) if the cache memory has a copy. With these assumptions, the adaptive cache coherence-replacement mechanism for this case is the same as the previous one.

3.3 Cache Coherence-Replacement in a Web Proxy Cache

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 [3, 4, 5, 7, 10, 13]. 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 [3, 5]. 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 [13]. 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 [11], 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 [11]:

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: i) Send ICP queries to some neighbors and not to others, ii) 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, iii) Disallow or require the use of some peers for certain requests. In this case, each cache block can be in one of the following states:

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

3.4 Our Generic Replacement Subsystem

Typically, a cache replacement technique must be evaluated with respect to an offered workload that describes the characteristics of the requests to the cache. Of particular interest are patterns in the objects referenced and the relationships among accesses. Workload is sufficiently complicated that we can use other types of information to try to solve this problem. Thus, we define a set of parameters that we can use to select the best replacement policy in a dynamic environment:

- Information about the system: Workload, Bandwidth, Latency, CPU Utilization, Type of system (Shared memory, etc.)
- 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)).
- 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 listed in section 2.2, we define 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 or 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:

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

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
If *the Local CPU utilization is low* then
Choose a process to suspend (In this case, we can use a PC policy to select the process) and call the algorithm again for this reduced set of processes.

For all other situations, choose randomly a replacement policy. The CPU utilization criterion avoids the starvation or thrashing problem on the system. In general, these rules are based on the average of the different criteria in the system. We can take into account the victim's information (the process that has been selected to expel its page) with the next rule:

If *the victim's SL or TL is high* then
Lock this page % Choose another page
And call the decision system again with this smaller problem.

If we don't find a page to expel according to the last rule, we choose the first victim process that we had selected and we suspend this process. In the case of web proxy systems, we don't use the last two rules because they are very specific for multiprocessing systems.

4. 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 must be validated by experimental work in the future, for example, using the Squid open source proxy cache [11]. In general, we plan to do a prototype implementation of our techniques and study their impact on the quality of the cache management, taken into consideration the additional cost (in time and space) our approach requires.

Acknowledgment

Jose Aguilar was supported by a CONICIT-Venezuela grant (subprograma de pasantías postdoctorales).

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. Dilley J., Arlitt M. Improving Proxy Cache Performance: Analysis of Three Replacement Policies, IEEE Internet Computing, November, (1999) 44-50.
4. Krishnamurthy B., Wills C. Piggyback Server Invalidation for Proxy Cache Coherency, Proc. 7th Intl. World Wide Web Conf., (1998) 185-193.
5. Krishnamurthy B., Wills C. Proxy Cache Coherency and Replacement-Towards a More Complete Picture, IEEE Computer, Vol. 6, (1999) 332-339.
6. 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.
7. Liu C., Cao P. Maintaining Strong Cache Consistency in the WWW, Proc. 17th IEEE Intl. Conf. on Distributed Computing Systems, (1997).
8. 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.
9. Obaidat M., Khalid H. Estimating NN-Based Algorithm for Adaptive Cache Replacement, IEEE Transaction on System, Man and Cybernetic, Vol. 28 (1998) 602-611.
10. Shim J., Scheuermann P., Vingralek R. Proxy Cache Design: Algorithms, Implementation and Performance, IEEE Trans. on Knowledge and Data Engineering, (1999).
11. Squid Internet object cache. http://squid.nlanr.net/Squid.
12. Stenstrom P. A Survey of Cache Coherence Schemes for Multiprocessors. IEEE Computer, (1990) 12-24. 1990.
13. Wills C., Mikhailov M. Towards a better Understanding of Web Resources and Server Responses for Improved Caching, Proc. 8th Intl. World Web Conf., (1999).