BROWN UNIVERSITY
Department of Computer Science
Master's Thesis
CS-90-M3

''Report on Implementing Caching for ObServer Clients''

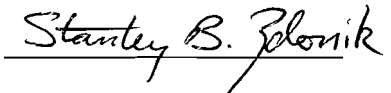
by
Richard M. Kogut

Report on Implementing Caching for ObServer Clients

Thesis

Richard M. Kogut
Department of Computer Science
Brown University
April 26, 1990

Submitted in partial fulfillment of the requirements for the degree of Master of Science in the Department of Computer Science at Brown University



Professor Stanley B. Zdonik
Advisor

OBJECTIVES

The goal of this project was to enhance ObServer and the ENCORE application interface [1] to implement the caching of objects by the application interface across transactions. To achieve this goal, preliminary work was required to restructure the way ObServer processes client requests and packages objects that are to be sent, to restructure the way the applications interface processes messages from ObServer, and to fix various bugs in both ObServer and in the application interface.

INTRODUCTION

ObServer is the persistent object store and transaction manager for an Object-Oriented Database system designed and implemented at Brown University. ENCORE is the higher-level portion of the database system that supports the type scheme. As such, ENCORE is a client of ObServer. An application interface is available to ObServer clients. Actually, several application interfaces exist. One simply mirrors all of the functions provided by ObServer onto a one-to-one mapping of procedure calls. This interface is often referred to as the *ObServer/client* interface, and is used by the other application interfaces. A second application interface supports the GARDEN [2] application. The application interface primarily that most directly concerns this project was specifically designed for ENCORE. It is intended to facilitate the use of ObServer to implement a simple two-phase locking transaction protocol [3].

DESIGN PHILOSOPHY

The goal of object caching is to reduce both disk I/O at the object server and network traffic between the object server and the client. This is achieved at the cost of extra processing and complexity in both the server and the client, and potentially increased paging or disk I/O at the client.

The overall method of operation is for the client to maintain copies of objects after releasing the locks held on them. Subsequent requests by the client for the same object will cause the object server to grant the required lock, and then determine if the copy maintained by the client is up-to-date. If it is, the client is so informed; if not, a new copy of the object is sent. The number of network messages is not reduced as compared to the non-caching case; any performance improvements come from reduced message sizes and from avoiding disk I/O operations that might have been needed at the server to reread objects from disk (and possibly to write other objects out to disk to make room in memory for the requested ones).

An object should be placed in the cache at the time the transaction's read or write lock on it is released. A lock is released implicitly by committing a transaction or a specific object, or explicitly by downgrading a read or write lock to a null lock. Note that a null lock does not entitle a client to access an object, but is primarily a mechanism for enabling various ObServer lock communication modes. Caching is defined on a client basis rather than on a transaction basis, as the basic idea is to cache objects across transactions. If several transactions are to be simultaneously active for a given client, then the cache manager (generally embedded in the application interface) has to ensure that one transaction cannot access a copy of an object modified by a transaction of the same client before the object has been registered or committed at the object server.

Object caching requires cooperation between the object server and the client. The server must know which objects (and what versions) are being held in the client cache. The client cannot independently decide to use the cached copy of an object; it must request that the server both grant a lock and validate the client's version (more specifically, the timestamp in the case of ObServer). The original (partially

implemented) ObServer design had the server pass timestamps along with objects when they were sent to the client, and had the client specify a timestamp when it requested an object. One problem with that design was that ObServer assigned a timestamp when it updated an object, but the client had no way of knowing exactly what the value of that timestamp was. To avoid this problem and to simplify the ObServer/client interface, I chose to maintain all timestamp information within ObServer itself.

ObServer now maintains a list of cached objects with associated timestamps for each client. When a client request for accessing an object is processed, this list is searched to obtain the timestamp (if any) of the client's copy. This timestamp is then compared with the timestamp of the last update of the object to determine if a new copy must be sent. The client may force a new copy to be sent by requesting ObServer to remove its record of the client's copy. This is accomplished for individual objects by specifying the appropriate parameter when the object is requested, or globally by invoking functions in ObServer's *SVRcache_ops()* procedure. A more detailed description of maintaining and using the cached object list is given below in "REFID=details not deFIned".

Another design decision that must be made is how to determine which objects are to be kept in the cache and which are to be discarded. The object server is not involved in this decision. It is assumed that the size of the cache is determined by the application itself, most likely as a consequence of the application's processing characteristics and/or the size of the available swapping space on secondary storage.

Applications that have little locality of reference or use exotic locking and commit strategies (such as GARDEN) might decide to dispense with caching altogether. Transaction oriented applications might cache only the objects accessed by the previous N transactions on the assumption that several related transactions might be entered consecutively, but that unrelated transactions will not access the same objects. An LRU algorithm might be more appropriate for an application where the same objects tend to be accessed repeatedly. An application might find it useful to cache some objects and discard others. For example, ENCORE might want to cache type objects but discard instance objects.

It is evident that different applications' access patterns could render any particular caching strategy ineffective or even deleterious to performance. Therefore, the ENCORE interface allows the application to tailor the strategy actually used, and is designed to easily accomodate the inclusion of additional caching algorithms for experimentation.

The caching interface permits the application to suppress caching altogether if desired. This option is useful for transactions that bulk load data bases or so randomly access objects in a large database that the overhead of maintaining the cache doesn't justify the possible performance savings. Applications that do use caching can specify the size of the area used for cached objects as well as the maximum number of objects to maintain in the cache. The application can also specify the minimum space and the minimum number of objects to be freed up when the cache fills up and needs to be trimmed.

To take better advantage of guesses that applications may be able to make about the likelihood of needing to reaccess given objects, the caching interface allows the assignment of a cache priority from 0 to 9 to each object. When the cache needs to be trimmed, objects to be discarded are selected in priority order. No priority 9 objects are discarded until all priority 8 objects are discarded, etc.

Each priority actually represents a partition of the cache. Each partition is managed separately using the cache management algorithm assigned to it by the application. Two algorithms are currently available. The first is an LRU algorithm and the second discards all objects assigned to it. The priority 0 partition is assigned the discarding algorithm by default; all other partitions default to the LRU algorithm. Using these defaults, an application might assign priority 0 to objects that it is sure it will not need again, priority 2 to objects that will certainly be needed, and priority 1 to all other objects. This strategy will prevent the cache from being filled up with useless objects, and provide hints for which objects to discard should the cache fill up with useful ones.

IMPLEMENTATION DETAILS

This section will describe in detail the changes made to ObServer, the ENCORE application interface, and the ObServer/client interface. I will begin by describing the changes to the ObServer/client interface.

ObServer/client Interface

With a view to minimizing the work needed to modify existing application interfaces to conform to the new ObServer/client interface, changes were kept to the minimum. With one small exception, the sizes of the data structures passed across the interface are unchanged, although some fields have been redefined. However, clients may receive a different set of messages than before in response to a *SVRrequest_object()* call.

Five basic changes were made to the interface. Segments are no longer transmitted as special entities, but rather as multiple objects. Different information is sent in object headers than before. The message sent when the cached copy of an object is up-to-date contains additional information. A call with 4 different sub-functions is provided to control the maintenance of information about cached objects . Finally, a mechanism is provided so that the client can tell the server to ignore the cache information for individual objects.

ObServer is designed so that when an object is requested, related objects may be returned at the same time in anticipation of future need. This has the potential of reducing both disk I/O at the server and the number of message interactions at the cost of possibly wasted network traffic (if the objects sent in anticipation are not actually used). ObServer groups objects in segments, and all of the objects in a segment are assumed to be related for the purpose of sending them in anticipation. Two methods exist for requesting a segment's worth of objects. They can be requested explicitly via the *SVRfetch_segment()* call, or implicitly specifying a non-null segment lock on a *SVRrequest_object()* call.

When an entire segment is to be sent, the current version of ObServer attempts to grant locks on each object in the segment, and then sends the segment as a single entity. I chose to modify this procedure for several reasons. The first is that a segment is really just a grouping of objects, a logical grouping for the client and a physical grouping for the server. It seemed unnecessarily complex to require the client to know the internal details of how the server physically groups objects, as required by the current ObServer/client interface, and to provide additional code to interpret segments. The second reason is that when receiving an entire segment, the client becomes responsible for processing a set of messages, one for each object, denoting whether or not the requested lock was granted. These messages may, in theory, either precede or follow the arrival of the segment, and must be matched with the individual objects. Lastly, in the case where locks are not available for all of the non-specifically requested objects in a segment, objects for which locks are not granted are delivered to the client along with the other objects in the segment, and the client must avoid using them by mistake. Also, such objects are sure to become out-of-date as soon as the lock holder frees the lock if the lock is a write lock.

To provide a cleaner interface, segments are no longer sent as separate entities. Where the segment would be sent in the current version of ObServer, the new ObServer sends each object in the segment, with the exception of those for which the lock was not granted either because it was queued (hard lock) or not grantable (soft lock).

To reduce the number of messages and to simplify processing, the MSG$LOCK_GRANTED message, formerly sent by the server to the client, has been eliminated. Receiving an object implies that a lock has been granted; to simplify bookkeeping, the value of the lock is directly included in the object header for each object.

Note: The lock occupies the first word of the field previously used for the timestamp. The other word now contains the segment id (SID) of the segment from which the object was read.

Separate lock notification messages were also removed for the case where ObServer informs the client that the cached copy is still valid. To support this, the message class of the MSG$HAVE_REQ_OBJ message was changed so that a SVR_UID_LOCK_INFO data structure is returned instead of a SVR_UID data structure (see [4]). This results in lock information being included in addition to the unique object identifier (UID).

The *SVRflush_client()* was renamed *SVRcache_ops()*, and additional functions provided. The *SVRflush_client()* call is still accepted by the low-level ObServer client interface, but is mapped into a sub-function of the new procedure.

A mechanism has been added to allow the client to specify whether or not it has a cached copy of each object that it requests. The fourth byte of the lock specification passed as a parameter to *SVRrequest_obj()*, formerly unused, is now used as a flag to inform the server to either use or discard any knowledge it may have of a copy present in the client's cache.

ObServer

Work done on ObServer falls roughly into three categories: restructuring of the code that determines which objects and segments need to be sent to the client and sends them, support for object caching, and bug fixes. The code restructuring was necessary to permit a straightforward implementation of object caching. Control flow was simplified by combining and centralizing identical or nearly identical code from six separate procedures, eliminating data structures and subroutines that no longer serve a useful purpose, reorganizing some routines into cleaner functional units, and introducing a new method for processing object and segment requests. Most of the work is concentrated in SVRsvr_local.c.

Low level details of the changes are described in [5]. Other than the object caching support itself, the primary change of general interest is the new way segment and object requests are queued and processed. Objects may be sent to clients synchronously as a result of calls to such procedures as *SVRrequest_obj()*, *SVRfetch_segment()*, etc., or asynchronously, when a lock request queued as a result of a previous call is granted. Queued lock requests may be granted when locks are removed or downgraded as a result of commits, aborts, or explicit lock modifications. When this occurs, objects may be sent to several different clients as part of the same operation.

Client requests involving objects and segments are performed in two phases. A list of objects and segments to send is constructed in the first phase, and the actual transmission is performed when this list is processed in the second phase. List entries contain the requested unique object identifier (UID), segment identifier (SID), transaction identifier (TID)[1], segment lock, and index. When the UID is valid, the index contains the displacement of the object in the segment data structure. For replicated objects, the SID chosen is that of a segment the client has already received if one exists, otherwise it is the first SID in the list of SIDs for the object. An invalid UID is used to indicate that the entire segment is to be sent. Specifically requested objects are locked before entries are placed in the list. Locks for objects sent when entire segments are requested (either explicitly by *SVRfetch_segment()* or implicitly by *SVRrequest_obj()*) are obtained at the time the segment is processed using the segment lock in the list entry.

---

1 Actually, it is the address of the data structure describing the transaction that is maintained in the list, but it is simpler to describe processing in terms of the TID.

To optimize processing, the list is sorted by SID. Entries involving segments currently in ObServer's in-memory buffers are inserted at the head of the list, entries requiring disk accesses are added to the tail of the list. Duplicate entries are eliminated. For a given SID, entries grouped by TID, and sorted by index within each TID group. This allows the procedure which processes the list to step through each segment once per TID. As the segment is processed, specifically requested objects are sent one at a time, and the other objects in the segment are also sent if the segment lock is non-null[2] and obtainable. Non-specifically requested objects will also not be sent if the segment was previously processed for the same client.

The principal procedures for client object caching is located in a new module, SVRobjc.c. SVRobjc.c maintains a table containing a timestamped entry for each unique object identifier/client identifier (UID/CID) pair. The entries are chained by UID; sub-chains are anchored off the first UID entry if more than one client holds an object with the same UID. To improve access time, the table is implemented as a set of lists selected by a hash function.

Manipulation of table entries is bound by certain constraints. The principal rule is that as long as one lock is held on a UID, cache-entries may be timestamp-updated (register or commit), deleted (flush_client or receive_obj when cache information is to be ignored), or added (receive_obj, fetch_segment, commit of newly created objects). When it is necessary to determine if the client's copy of an object is valid, the timestamp in the object cache table entry is compared to the timestamp in the LOCK_OBJECT[3], if they are identical, the copy is up-to-date. When the last lock is removed for a UID, all table entries with out-of-date timestamps for that UID are removed. Any remaining entries will all have the same timestamp. The next time a lock is granted for the UID, the lock manager will search for an entry in the object cache table. If one is found, the lock manager will use that entry's timestamp value to set the timestamp in the LOCK_OBJECT. If no entry is found, the timestamp is set to the time-of-day. As ObServer does not maintain timestamps in persistent storage, this procedure is necessary to keep a record of the last valid timestamp for an object so long as at least one client owns a cached copy of that object. Object cache table entries are not written to persistent storage when database checkpoints occur. The loss of current timestamp information can result only in extra copies of objects being sent to clients; no erroneous data or messages can be generated.

Table entry manipulation is further controlled by the operation of *SVRcache_ops()*. Four sub-functions are available with *SVRcache_ops()*. SVR_CACHE_OP_FLUSH causes all cache information for the calling client to be discarded. SVR_CACHE_OP_CACHE_OFF effectively suppresses caching by suppressing the construction of the required table entries. SVR_CACHE_OP_CACHE_ON enables the maintenance of cache information. The number of objects for which information is kept is limited by a parameter. The default is 2000. SVR_CACHE_OP_DISCARD tells the server to discard cache information for specific objects. It is intended to be used by clients as they discard objects from their caches.

---

2 The segment lock set set null by *SVRrequest_single_obj()* and similar routines to supress sending other than the specifically requested object. Note that the *SVRrequest_single_obj()* call is now redundant; specifying a null segment lock with *SVRrequest_obj()* will have the same effect.

3 The LOCK_OBJECT is a data structure of the lock manager. One LOCK_OBJECT is maintained for each UID that currently has a lock held on it; the timestamp in it contains either the last time the object was updated or the time the lock was granted if the last update time is unavailable.

ENCORE application Interface

A few changes were made to the functions provided by the ENCORE application interface; numerous modifications were made to the code. Most of the changes were made to fix bugs, to provide missing function, to simplify the internals, and to properly handle both synchronous and asynchronous messages from ObServer.

The caching related functional changes include adding a call to set the maximum size of the cache (*CNTset_cache_limits()*), a call to set the cache priorities for objects subsequently accessed either explicitly or implicitly (*CNTset_cache_priorities()*), and a call to assign an algorithm to a given cache priority (*CNTset_cache_algorithm()*). Independently of the caching support, deficiencies in the functional interface were corrected by adding a call for deleting objects (*CNTdelete_obj()*), adding a call for replicating objects (*CNTduplicate_obj()*), and modifying the parameters passed for changing an object's size (*CNTchange_obj_size()*) to conform to the documented interface.

Message handling was radically modified. The original ENCORE application interface handled incoming messages directly in the procedure for each different call to ObServer. Messages that were unexpected or unrelated to the most recent call were not handled properly. In particular, when waiting for a queued lock to be granted, the arrival of any message at all was incorrectly interpreted as a lock granted message. Also, timeouts were not handled properly.

All message handling was moved to a central message processing routine. Message handling was generalized to handle more types of messages, multiple instances of the same message (particularly for the case when multiple objects are received), and to more often interpret the messages' contents, rather than make assumptions about them. The new message handler can be easily expanded to accomodate multiple transactions and notify messages.

Various bugs not related to message handling were fixed. Many problems were found with the code for handling requests for retrieving objects when a valid copy was present in the cache. This is understandable since this case probably never occurred with the version of ObServer the code was designed to work with. Other fixes include checking return codes from MALLOC and not trying to commit objects for which only read locks were held.

Two performance changes not related to caching were made. An unnecessary call to *SVRget_uid_segment()* following every request for an object was eliminated. Also, received objects were unnecessarily being copied from where the data message unpackaging routines placed them. The copying and related calls to MALLOC and FREE were eliminated.

Further details of changes made to the ENCORE application interface are documented in [5].

CONCLUSIONS AND SUGGESTIONS FOR FURTHER WORK

The project was a success in the sense that a cleaner and usable version of both ObServer and the ENCORE client were produced. Basic functions required for caching to be usable were made available. However, good performance is more dependent on the interaction of the parts than on the good performance of each part. Often, low-level optimizations do not contribute to overall performance. In terms of this project, so much time was spent on detecting on eliminating design deficiencies and bugs in the ENCORE client, that little time was left for examining the overall effect of the caching changes. It is my guess that for applications that do not user very large objects, the caching scheme will increase the number of network communications, and thereby result in decreased performance.

This can be addressed in the ENCORE client if, whenever the application requests an object for which a copy is found in the cache, the interface automatically generated a request to read every object in the cache with the same segment identifier. This would result in a single network message exchange instead of one for each object. Note that if the requested object is out-of-date, the problem is moot, because the entire segment will be reread from disk and its objects will be sent to the client. But if the object is valid, the server cannot determine what other objects are in the segment without rereading the segment to find out, so only the requested object's timestamp is verified. Care should be taken to avoid paging if this strategy is implemented in the ENCORE client. I would suggest a single MALLOC to pre-allocate data structures for describing the cache entries (this is possible as the maximum cache size is known), and thereby localize them in virtual memory. Another set of hash tables and pointer chains might be used to chain objects belonging to the same segment.

The effectiveness of a given caching strategy will depend on the number and size of the objects in a typical segment, whether or not the server is running on the same node as the client, whether or not other clients modify objects used by the first client, and whether or not segments accessed by the client are likely to remain in server memory. This last question depends on the number and size of segments accessed by the client, the size of the memory pool dedicated to holding segments in the server, and activity by other clients that might cause segments used by the first client to be displaced from main memory.

All of these factors should be considered when designing benchmarks or improvements to the caching algorithms. Two strategies that seem relatively invariant to changes in the above characteristics may be worthy of further study. One is the use of optimistic caching strategies that would assume that copies of the objects in the local cache were valid, and have the commit refused if this assumption turned out to be false. In this context, one might have an option of permitting null locks to to be held between transactions combined with a commit option to transform locks instead of releasing them. This would allow the use of ObServer notification modes to inform clients when cached entries became invalid.

The other strategy would be the use of locks on the segment level, perhaps in conjunction with object locks as a two level tree. This could reduce the number of cache timestamp validations and associated network validations.

REFERENCES

1. Hornik, M.H. and Zdonik, S.B. *A Shared, Segmented Memory System for an Object-Oriented Database* ACM Transactions on Office Information Systems 5,1 (Jan. 1987), 70-95.

2. Reiss, S.P. *An object-oriented framework for graphical programming* SIGPLAN Not. 21,10 (Oct. 1986) 49-57.

3. Lo S.W. *Senior Project: ObServer Client Module* Brown University Dept. of Computer Science, May 1989 (unpublished).

4. Fernandez, M. *ObServer II Server Interface Specification* Brown University Dept. of Computer Science, 1988 (unpublished).

5. Kogut, R. *Informal report on changes made to ObServer* Brown University Dept. of Computer Science, 1989 (unpublished).