Lockup of a Client Object Cache and How to Avoid It (Student Paper)

Mark Day*
mday@lcs.mit.edu
MIT Laboratory for Computer Science
545 Technology Square
Cambridge, MA 02139, USA

Abstract

I present the problem of client object cache lockup in a distributed object-oriented database system where storage is recovered by garbage collection. If garbage collection is augmented with the ability to discard reachable unmodified versions of persistent objects, cache lockup is greatly reduced. I outline some remaining questions about this technique for managing an object cache.

1 Introduction

In a distributed object-oriented database system, server machines store persistent objects shared by applications running on client machines. When an application invokes an operation on a persistent object, that operation must run on either the server or client. I consider the case where objects are moved or copied to the client for at least the duration of the client transaction. A number of existing object-oriented databases work partially or entirely in this mode of executing operations on the client machine: examples are O2 [1], GemStone [2, 9], and Orion [7].

2 Complicating factors

The Thor system[8] includes three features that improve performance but complicate cache management: inter-transaction caching, swizzling, and object groups as the unit of transfer.

*Telephone: +1 (617) 253-6015 Fax: +1 (617) 258-8682 This research was supported in part by the Advanced Research Projects Agency of the Department of Defense, monitored by the Office of Naval Research under contract N00014-91-J-4136 and in part by the National Science Foundation under Grant CCR-8822158.

2.1 Inter-transaction caching

Some systems discard all objects in the cache after a commit. This approach is simple and effective if transactions rarely share objects. However, in applications such as computer-aided design (CAD), a transaction on a large, complex data structure is often followed by another transaction on the same data structure. Inter-transaction caching can improve performance significantly for these applications.

2.2 Swizzling

Objects at the server refer to each other by names that are unrelated to their addresses in the client cache. A simple but inefficient way to implement references in the client cache is to maintain a table mapping server names to client addresses. To follow an inter-object reference, the system looks up the name of the referenced object and finds the corresponding address. For many workloads, it is more efficient to *swizzle* the inter-object reference at the client, actually mutating a cached copy of an object so that each server name is replaced by its corresponding client address[11].

2.3 Transferring object groups, not pages

In many systems, the unit transferred from server to client is a page. The client then manages a cache of pages much like a paged virtual memory, treating the server as a backing store.

Some complications arise from transferring pages because the client is working in terms of transactions on objects. One such complication is that a dirty page evicted from the client cache must not overwrite its old version at the server until the client transaction commits.

Another complication is that a typical page contains many objects. If concurrency control is done at the page level, there can be unnecessary conflicts or aborts due to false sharing. These conflicts and aborts may be avoided by more complicated concurrency control schemes, but only at a significant cost in complexity.

Yet another problem is that the use of virtual memory addresses as references in a page-based scheme can make it more difficult to manage client memory. In particular, it is useful to compact the reachable objects so as to recover space that was occupied by unreachable objects or objects that have moved, but this sort of compaction is difficult if objects are fixed in pages.

Finally, the need to cluster objects statically into pages limits the flexibility of prefetching, and may reduce performance accordingly[4].

3 Cache lockup and how to avoid it

Because Thor fetches object groups, swizzles references, and caches objects across transaction boundaries, it needs a different approach to cache management.

3.1 Garbage collection

Garbage collection is one obvious mechanism for managing the client object cache. A garbage collector treats the client application's variables as roots and recovers storage that is not reachable from those roots. However, garbage collection is not sufficient. If the cache fills with reachable objects, a garbage collector cannot recover any space. This condition can arise without warning: there is typically little or no degradation of performance before such a problem stops the computation, a condition I call "cache lockup". In a simulation of Thor using OO7 traversal 1 on a small OO7 database[5], caches smaller than 1571 KBytes locked up when using only garbage collection.

3.2 Shrinking to surrogates

Another way to recover space is to *shrink* an unmodified persistent object. This is roughly analogous to evicting a clean page, but because of swizzling there may be direct memory pointers to the storage previously occupied by the object. So when an object is shrunk, it is replaced by a *surrogate*, a small data structure containing only the information needed to refetch the object if needed (cf. *leaves* in LOOM[6], *forwarders* in Mneme[10]). The surrogate ensures that any attempt to use a shrunk object causes the object to be refetched.

When a garbage collector fails to reclaim enough space, it can shrink some objects and then proceed. The combination of shrinking and garbage collection is quite robust. In simulations of Thor using garbage collection and shrinking, I have demonstrated that computations can proceed without cache lockup with larger and larger workloads or smaller and smaller caches until performance has greatly degraded due to cache misses and other cache-related overheads[3].

Surrogates are not strictly necessary, since the garbage collector can fix up the relevant pointers; however, surrogates simplify the task of the garbage collector. In addition, surrogates allow *invalidation*: if a cached object is modified at the server by a committed transaction, the system can avoid an unnecessary abort by shrinking the outdated cached copy of that object, so that any computation touching that object must fetch the new version from the server.

4 Finding a Good Policy

There are a number of choices involved in combining garbage collection with shrinking:

- when does the garbage collector decide to shrink some objects?
- how many objects should be shrunk?
- how are the victims chosen for shrinking (e.g. LRU, random)?

As long as the cache is managed with both garbage collection and shrinking, it is relatively easy to avoid cache lockup; the avoidance of cache lockup does not seem to be very sensitive to the details of garbage collector or shrinking policy. However, two different implementations may have very different performance: both may complete a given computation without lockup, but one may run 10-100 times slower than the other. I am currently studying a number of different implementation techniques with a number of different simulated applications to understand what will make a good general-purpose combination of shrinking and garbage collection.

Acknowledgement

Thanks to Barbara Liskov for useful criticism of this note.

References

[1] François Bancilhon, Claude Delobel, and Paris Kanellakis, editors. *Building an Object-Oriented Database: The Story of* O_2. Morgan Kaufmann, 1992.

[2] Paul Butterworth, Allen Otis, and Jacob Stein. The GemStone object database management system. *Communications of the ACM*, 34(10):64–77, October 1991.

[3] Mark Day. Client cache management in a distributed object system. PhD Thesis, MIT Department of Electrical Engineering and Computer Science (forthcoming).

[4] Mark Day. Object groups may be better than pages. 4th Workshop on Workstation Operating Systems, Napa, California, October 1993.

[5] Michael Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 Benchmark. Technical Report, Department of Computer Science, University of Wisconsin, April 1993.

[6] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 87–106, 1986.

[7] Won Kim, Nat Ballou, Hong-Tai Chou, Jorge F. Garza, Darrell Woelk, and Jay Banerjee. Integrating an object-oriented programming system with a database system. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 142–152, 1988.

[8] Barbara Liskov, Mark Day, and Liuba Shrira. Distributed object management in Thor. In M. Tamer Özsu, Umesh Dayal, and Patrick Valduriez, editors, *Distributed Object Management*. Morgan Kaufmann, San Mateo, California, 1993.

[9] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an object-oriented DBMS. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, pages 472–482, 1986.

[10] J. Eliot B. Moss. Design of the Mneme persistent object store. *ACM Transactions on Information Systems*, 8(2):103–139, April 1990.

[11] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. Technical Report 90-38, COINS, University of Massachusetts - Amherst, 1990.