Preventing Cache Overflows in an Object-Oriented Database Management System with the Object-Descriptor Architecture

Byung S. Lee
Department of Computer Science, University of Vermont, Burlington, VT 05405
E-mails: bslee@cs.uvm.edu

Abstract

In an object-oriented database management system built in the object-descriptor architecture, there is a concern about cached object-descriptors (CODs) filling up the client object cache space and, consequently, failing a database transaction. This phenomenon occurs especially during the data loading which keeps creating new objects in the object cache space. In this paper, we reexamine the structure and use of CODs, discuss the pros and cons of retaining CODs in an object cache, and propose a resolution to the problem of cache overflow. Our solution requires only a slight modification of the transaction termination protocol.

1 Introduction

Most object-oriented database management systems (OODBMSs) are built in the object-descriptor architecture (ODA)[1, 2]. The ODA is characterized with using a surrogate record as an object identifier (OID). We call the surrogate record a "cached object descriptor" (COD) after the commercial OODBMS used in our previous works[4, 5].

An OODBMS has often been criticized for its apparent lack of scalability, especially with respect to the data volume. This triggered our benchmark experiment in [5] with a focus on the algorithmic scalability of the ODA OODBMS for database loading. The result showed a limit of the OODBMS when a large number (e.g., several millions) of objects were handled. Specifically, the object cache space became full, entailing a fatal system error. (We did not consider overcoming the limit by increasing the cache space because our objective was in assessing the algorithmic scalability.)

Why does the object cache overflow happen? It is because CODs, once created, are retained in the object cache as long as the application program[1] is running and, thus, keep accumulating even if the objects are removed from the object cache. They are retained to facilitate re-fetching objects from the database, as will be explained in Section 4. This poses a trade-off between the object retrieval time and the available object cache space, and warrants a strategy for retaining or removing CODs toward balancing the trade-off. We proposed our approach to determining the COD release time in this paper.

[1]We assume a strongly-typed language, like C++, as the application programming language.

After describing the COD and the object caching mechanism in Section 2 and Section 3, respectively, we explain the reason for retaining CODs in the object cache space in Section 4, discuss the problem caused by it in Section 5, propose our resolution to the problem in Section 6, and conclude the paper in Section 7.

2 Cached Object Descriptors

In the ODA, cached objects are addressed indirectly through CODs. Figure 1 shows the fields in the COD used in our benchmark OODBMS [5]. A COD maintains the mapping between an OID and an object in the object cache. For this mapping, one field contains the main memory address of the cached object. The state field is a bitmap of 32 bits. It includes bits describing the lock level (e.g., shared, update, exclusive), the pinning status (where 'pinned' means "held in an object cache space"), and the markers (i.e., new, update, delete). These markers are set when a cached object is newly created, updated, and deleted, respectively.

object identifier (8 bytes)
main memory address of an object (4 bytes)
state (4 bytes)

Figure 1: COD record fields.

A COD is allocated in the object cache when a new object is created or a persistent object is fetched from the database. A COD is also allocated for every object reference embedded within each cached object, in which case the main memory address fields are set to null.

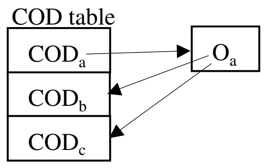
For example, consider the example class schema in Figure 2. Figure 3a shows the memory allocation of objects and CODs after the instance O_a of the class A is allocated. The DBMS creates COD_a and have it point to O_a. It also creates COD_b and COD_c and covert the object references

O_a.b and O_a.c to the pointers to COD_b and COD_c, respectively. (This process of converting a reference field to a main memory pointer is called the "pointer swizzling.")

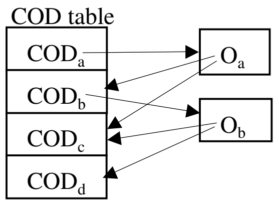
Figure 3b shows the memory allocation after the instance O_b of the class B is allocated as a result of navigating from O_a following the reference O_a.b. Assuming O_b.c is set equal to O_a.c (i.e., O_b.c := O_a.c), the reference O_b.c is converted to a pointer to COD_c. In addition, a new COD_d is created to swizzle the reference O_b.d. These CODs, once created during the execution of a program, remain in the object cache until the application program terminates.

```
class A {            class B {
  ref<B> b;            ref<C> c;
  ref<C> c;            ref<D> d;
};                   };
```

Figure 2: An example class schema.



a) After creating/fetching an instance of the class A.



b) After creating/fetching an instance of the class B.

Figure 3: Memory allocation for objects and CODs.

3 Object Caching Mechanism

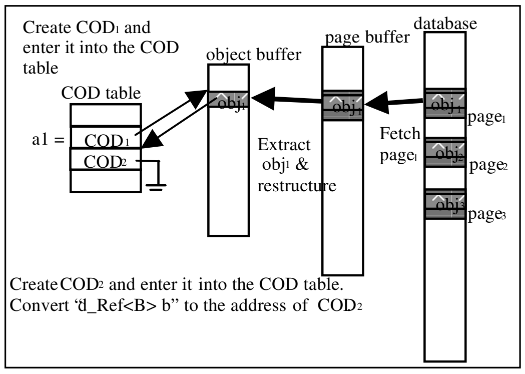
The object cache space is configured in two tiers – the object buffer (i.e., cache) and the page buffer. (This is called "dual buffering.") There exists one page buffer per server database and one object buffer per client application program.

In this dual buffering architecture, the object caching works as follows: (1) Locate the page containing a target object in the database and fetch it into the page buffer; (2) Extract the target object from the page in the page buffer, restructure it from the disk format to the main memory format, and place it in the object buffer; (3) Create the COD for the object in the object buffer and enter it into the COD table, and have the COD point to the object; (4) For each external object reference in the fetched object, create a new COD, enter it into the COD table, and have the object point to the COD.

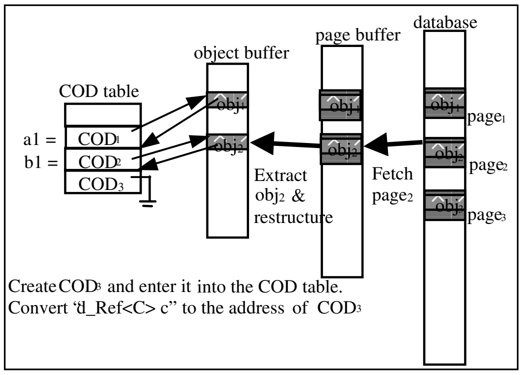
Figure 5 illustrates it given the example code in Figure 4

```
class A: public d_Object {
public:
  d_Ref<B> b;
}

class B: public d_Object {
public:
  d_Ref<C> c;
}

main() {
...
   d_Ref<A> a1;
// Get a1 from the database.
   d_oql_execute(query, a1);
// Navigate to b and get b1.
   d_Ref<B> b1 = a1->b;
// Navigate to c and get c1.
   d_Ref<C> c1 = b1->c;
...
}
```

Figure 4: An example code for fetching objects.


Create COD_1 and enter it into the COD table
object buffer
page buffer
database
COD table
a1 =
COD_1
COD_2
obj_1
Extract obj_1 & restructure
Fetch $page_1$
obj_2
obj_3
$page_1$
$page_2$
$page_3$
Create COD_2 and enter it into the COD table.
Convert "d_Ref<B> b" to the address of COD_2

a) After d_oql_execute(query, a1);



b) After d_Ref<B> b1 = a1->b;

Figure 5: An example of caching objects.

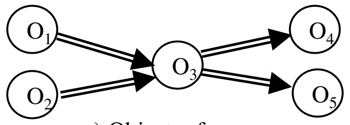
(written in ODMG C++ [3]). In Figure 5a, the query in Line 11 fetches the object obj_1 from the database into the object buffer through the page buffer. Then, COD_1 is created, entered into the COD table, set to point to obj_1, and assigned to the program variable a1 (declared in Line 10). In addition, COD_2 is created for the reference field in obj_1 (in Line 3) and entered into the COD table, and the reference field is converted to a pointer to COD_2. Besides, a new OID is created and stored in the OID field of COD_2.

In Figure 5b, the navigation in Line 12 finds COD_1 in the COD table given the program variable a1, follows the pointer to obj_1 in the object buffer and then to COD_2 in the COD table. Then, it retrieves the OID field in COD_2, fetches obj_2 (with the OID) from the database into the object buffer through the page buffer, and set COD_2 to point to obj_2. In addition, COD_3 is created for the reference field in obj_2 (in Line 7) and entered into the COD table, and the reference field is converted to a pointer to COD_3.

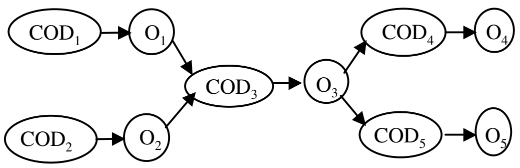
This object caching causes a bottleneck to the scalability because an object cache is allocated in the virtual memory whose size is limited. The system is "choked" if the virtual memory becomes full and no memory can be released from it.

4 Rationale behind Retaining CODs

As illustrated in Figure 6, object de-referencing is done indirectly through the COD of the referenced object. This indirect addressing is an important feature for rendering object identity immune to the change of object location.



a) Object references.



b) Pointer linkage of objects and CODs.

Figure 6: Implementation of object references.

Given the COD of an object, the object with the OID (stored in the COD) can be fetched from the database. Later on, if the COD is released along with the cached object it is pointing to, two problems may occur. First, it may leave dangling pointers from other objects to the released COD, thus invalidating references to the removed object. Second, since the OID of the removed object is no longer available, a query must be executed to retrieve the object from the database if the object is needed again. Unfortunately, this querying is not so transparent, nor convenient, as de-referencing an object reference by looking up the object's COD. Moreover, a query is much more expensive than fetching the object with the particular OID. The ODA retains CODs for these reasons.

It is possible to release a COD manually using a special system function (called "zapcods" in the OODBMS we used). But, it may inadvertently leave dangling pointers to the removed COD. Hence, users would want to do this only if the object cache space is severely short and are sure it is safe, that is, does not leave any dangling pointers.

5 Problem of Retaining CODs

As already mentioned, a COD is never released once created in the object cache. This imposes an upper bound on the number of objects that can be allocated in the object cache, and the upper bound is no more than the ratio of object cache size to the COD size. For example, a 128 Mbyte object cache can accommodate a maximum of 8.2 million CODs even in such an unrealistic case as there are only CODs and no object in the object cache. Worse, the number of CODs tends to be far larger than the number of objects given the ODA's pointer swizzling mechanism. (That is, one COD is created for each object reference within a cached object.)

There may be a locality of reference manifested in an application's data accesses and, in this case, the number of CODs may stay low enough to avoid cache overflow. Unfortunately, however, this has not been the case in our benchmark experiments[4, 5]. In this case, objects are loaded into the database while an index is constructed on the loaded objects. The entire loading process must belong to one (long) transaction in order to avoid fragmenting the index. Thus, more and more new objects are created as the application session continues and, eventually, the object cache becomes full without the possibility of releasing any existing COD.

6 Resolution

One idea naturally stemming from the above observations is that we should release some CODs early enough in order to prevent an object cache flow. When is "early enough"? How do we select the CODs to be discarded?

We consider three different approaches – optimistic, pessimistic, and hybrid. In the optimistic approach, we assume an object cache overflow will not occur and, therefore, retain all CODs. Naturally, the benefit of this approach is the efficiency of re-fetching objects, and the cost is the risk of an object cache overflow, which leads to the failure of an application program.

If an object cache flow does occur, then the DBMS catches the system error signal and starts garbage collection in the cache to free all CODs that are not pointed from any object. There have been significant research works about garbage collection in OODBMS [6, 7, 8], but all of them dealt with releasing objects from the database in disk. Our problem is about CODs, which are main memory-resident structures. Therefore, we rely on traditional garbage collection algorithms designed for a main memory heap [9, 10, 11].

In the pessimistic approach, we release CODs at the earliest possible time. There are three phases of releasing CODs. In the first phase, a COD is released immediately after the associated object is deleted from the cache space as long as it is safe. In order to support this immediate COD release, we maintain the reference count as an additional field in a COD. (The reference count refers to the number of objects that are pointing to the COD.) Every time an object is removed from the object cache, the DBMS checks the reference count of its COD and, if zero, removes the COD as well.

The second phase occurs when all objects are removed from the object cache at the end of the application program (i.e., transaction commit or rollback). All CODs of the removed objects can be released safely at this point. This allows for de-allocating the COD table as one atomic operation instead of the individual CODs in it, thus more efficient. If an object cache overflow still occurs despite all the efforts, then, in the last phase, the garbage collection is performed. The pros and cons of this approach are the opposite of the optimistic approach.

The hybrid approach decides when to release CODs while monitoring the use of object cache space. It initially takes the optimistic approach and switches to the pessimistic if the available cache space falls below a threshold and switches back to the optimistic mode if it rises above a threshold. Thus, this approach adapts to the actual object cache usage of the application program. The cost of this approach is the monitoring overhead each time an object or COD is allocated or de-allocated. This is an unnecessary effort if the application is small enough to ensure no overflow of the object cache space.

The relative effectiveness of these three approaches depends upon the actual object cache usage pattern of the application program. A system administrator can adjust the rate of releasing CODs by designating one of the three approaches and, for the hybrid approach, additionally setting the lower and upper thresholds of the available object cache space.

7 Conclusion

We have addressed the object cache overflow problem inherent in the ODA OODBMS. In the ODA, CODs are retained in the cache space for the efficiency of re-fetching objects during the execution of an application program. Ironically, this causes object cache to overflow in case too many objects (and their CODs) are allocated in it.

We have proposed three alternative resolutions - optimistic, pessimistic, and hybrid. The optimistic approach releases CODs as late as possible for the re-fetch efficiency. The pessimistic approach releases CODs as early as possible for reserving enough cache space. The hybrid approach enables the DBMS to switch between the two approaches depending on the size of the remaining object cache space.

References

[1] W. Kim, Introduction to Object-Oriented Databases, The MIT Press, 1992, Chapter 13, pp. 176-179.

[2] M.E.S. Loomis, Object Database: The Essentials, Addison-Wesley Publishing Company, 1995, Chapter 7, pp. 128-129.

[3] D. Jordan, C++ Object Databases: Programming with the ODMG Standard, Addison-Wesley Publishing Company, 1998.

[4] M.R. Olson and B.S. Lee, "Object Databases for SGML Document Management," Proceedings of the 30th Hawaiian International Conference on Systems Sciences (HICSS), Volume III, pp. 39-48, January 1997.

[5] K.Y. Yu, B.S.Lee, and M.R. Olson, "The Scalability of an Object Descriptor Architecture OODBMS," Proceedings of the IDEAS99 International Database Engineering and Application Symposium, pp. 370-377, August 1999.

[6] L. Amsalog, M. Franklin, and O. Grber, "Efficient Incremental Garbage Collection for Client-server Object Database Systems," Proceedings of the 21st VLDB International Conference on Very Large Data Bases, September 1995.

[7] J.E. Cook, A.W. Klauser, A.L. Wolf, and B.G. Zorn, "Semi-automatic, Self-adaptive Control of Garbage Collection Rates in Object Databases," Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 377-388, June 1996.

[8] J.E. Cook, A.L. Wolf, and B.G. Zorn, "Partition Selection Policies in Object Database Garbage Collection," Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 371-382, May 1994.

[9] L.P. Deutsch and D.G. Bobrow, "An Efficient, Incremental, Automatic Garbage Collector," Communications of the ACM, Vol. 19, No. 9, pp. 522-526, September 1976.

[10] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens, "On-the-fly Garbage Collection: An Exercise in Cooperation," Communications of the ACM, Vol. 21, No. 11, pp. 966-975, November 1978.

[11] P.R. Wilson, and B. Hayes, "Garbage collection in object oriented systems," ACM SIGPLAN OOPS Messenger, Vol. 3, No. 4, pp. 63-71, October 1992.