Implementing Persistence for an Object Oriented Language

J. Eliot B. Moss

COINS Technical Report 87–69
September 1987

A report on work in progress, presented at the Workshop on Persistent Object Systems: Their Design, Implementation, and Use, Port Appin, Scotland, August 25–28, 1987; revised September, 1987, for the final proceedings.

1 Introduction

Recently we have been interfacing and integrating an object server (ObServer, designed and built by Professor Stan Zdonik and his group at Brown University [Skarra, et al. 86]) with an object oriented language (Trellis/Owl, under development in the Object Based Systems group at Digital Equipment Corporation [Schaffert, et al. 85, Schaffert, et al. 85, O'Brien 85]). The goals are to support reasonably transparent persistence in a manner consistent with Owl and its heap-oriented, type-checked, multiple inheritance semantics, and to explore means for the programmer to express desired clustering of objects to improve performance. We describe our design and results to date.

2 Trellis/Owl

Trellis/Owl, or Owl for short, provides a heap-oriented, garbage collected object world. The language is similar to CLU [Liskov, et al. 1977, Liskov, et al. 1981], but features a class structure supporting multiple inheritance, while maintaining strong typing and compile-time type checking. Owl supports the notion of a workspace, which is simply an address space full of objects, but other than saving and restoring whole workspaces, it does not support persistence; neither does it support sharing among workspaces. Thus, its underlying object world is rather similar to that provided by Lisp or Smalltalk [Goldberg and Robson 83] – objects have unique identifiers, but they are not persistent. In the current implementation of Owl, object identifiers are main memory addresses, so the compacting garbage collector can actually change the identifiers over the course of a program execution.

3 Low Level Support for Persistence

We desired to add a shared persistent store to Owl. To keep the project manageable in size, and also to encourage its acceptance by other Owl users, we wanted to make minimal changes to the existing Owl system. Therefore, a running Owl program still uses the main memory address to access objects, be they ones retrieved from the persistent store, or ones locally created and not (yet) persistent. At a low level, this involves maintaining correspondence between external, persistent, globally unique identifiers, and the internal identifiers. We also needed means to transfer objects to and from the persistent store, while maintaining the inter-object relationships. The correspondence is readily implemented using a hash table to map from persistent unique identifiers (pids) to local object identifiers (oids). We must also map the other way (oid to pid) when returning or inserting objects into the persistent store. Thus we need two hash tables. To conserve space, while each

table has its own bucket pointers to the first entry of each hash bucket, the table entries are shared by using two chain fields, one for each table, as shown in Figure 1.

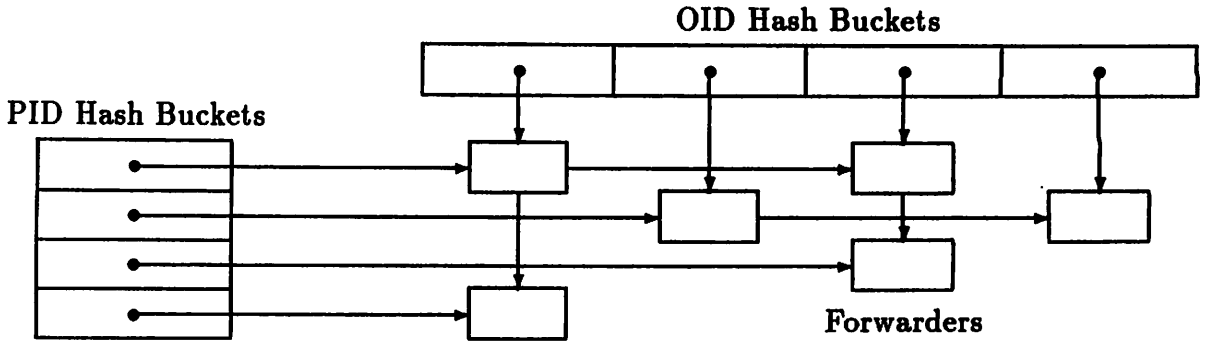

Figure 1: The Shared Hash Table Scheme

Such a scheme can be used by any persistent store mechanism that needs to convert between "local" and "global" unique identifiers (uids). We realize that this approach has an inherent cost. However, using global uids all the time has substantial costs associated with it as well, and that cost would be unacceptable on current hardware. Clearly this is a problem that needs addressing, but it lies outside the scope of our initial project described here .

In addition to being able to translate between pids and oids, we wanted the movement of objects to be as automatic and transparent as possible. Therefore, we designed low level mechanisms that "fault" on the basis of individual objects. Every reference to a persistent object, whether resident or not, is in fact a pointer to what we call a *forwarder object.* The forwarder is a local object that contains information such as whether the persistent object is resident (and if so, the oid for the resident version), the pid of the object, and so forth. The hash table entries mentioned above are in fact the forwarder objects, so forwarders have two chaining fields as well. Note that when a persistent object is made resident, we must create forwarders for each object it refers to, if such forwarders do not already exist.

When an attempt is made to perform an operation on a forwarder, a low level mechanism checks (reasonably efficiently) if the object is resident. If it is, the operation is simply forwarded to the resident object; otherwise, a fault occurs and the object is brought in from the persistent store. When and how objects are returned to and inserted in the store will be discussed later.

4 Sharing and ObServer

The persistent store is shared between multiple concurrent users on a local area network. Thus we needed to do something about concurrent access and update. ObServer, which we used for the store itself, provides a number of concurrency control features; in fact, one of its goals is to explore such mechanisms. We decided to use optimistic concurrency control in our design, for two reasons. First, it should be adequate for short transactions (such as check-in/check-out), and also for long transactions where conflict is not expected. Further, ObServer can notify us of conflicts as they occur, so we can abort before the transaction attempts to commit.

We are taking as a working hypothesis that the current mechanism is adequate – namely, that long design transactions can be expressed in terms of a number of short transactions. However, concurrency control is not the primary focus of the current implementation work. We are thinking about transaction models for cooperative design applications, and may eventually implement such a model in the Owl/ObServer system. In the area of robustness (crash recovery), ObServer provides no support, so we do not either. However, transactions can be rolled back before they are committed. Again, we must deal with the issue eventually, but we wish to gain more experience with using persistent store semantics in an object oriented setting before exerting that effort.

Transaction commit is the time when modified and newly created objects are written to the store, and all such objects are written back at once. We can retain local copies of the objects if we are going to continue working with them, or the local object cache can be flushed. In writing objects to the server, they are collected into what we term *groups*, which correspond roughly to disk blocks, and are sized for efficient transfer over the network and to secondary storage. To make retrieval fast, pids are a pair, (group-#, object-#), where object-# is the number of the object in the group. Pids should probably be location independent, so this may not be the best decision in the long run. However it serves our current purposes and is easier to implement (and likely performs better) than a location independent scheme.

ObServer has little notion of the semantics or content of objects. In terms of storage and retrieval, all ObServer does is associate a uid with a string of bytes, and allow the retrieval of the previously stored bytes given the uid. Because of ObServer's weak notion of object, but more to avoid per-object overhead, we used ObServer objects to store entire groups rather than single Owl objects.

Thus, since groups are the unit of transfer, we retrieve all objects in a group at once. However, our approach also implies that groups are the unit of locking for the concurrency

control algorithm, because ObServer's concurrency control is (naturally) in terms of ObServer objects. This certainly reduce concurrency control overhead, but because of the coarser granularity of locks, it may reduce concurrency. If, however, the clustering techniques work well, then concurrency will not be greatly reduced, since most of the objects in a group will be used together anyway. Furthermore, locking on the basis of individual objects is likely to be prohibitively expensive. This is also an area that will require more work as we delve more deeply into the issues.

5 Grouping Rules

As previously mentioned, one of our goals in this work is to investigate object clustering techniques for improving performance. Groups give a low level mechanism that helps, but the way in which objects are grouped is obviously important. We have devised a technique that groups objects according to declarative rules that are expressed at the source language level. The user can define any number of named *clusters*; a cluster consists of a set of groups, to which new groups can be allocated dynamically. The rules allow objects of specific types (Owl is a strongly typed language, but offers multiple inheritance) to be directed to specific clusters. In addition, specific components (*component* is the Owl term for "slot" or "attribute") of objects of a given type can also be directed to named clusters, and these directions override the usual specification for the component's type. Components can also be declared to go with the parent object, wherever that may be, rather than to a named cluster.

The high level features of the system are *repositories*, *dbCollections*, and *dbObjects*, largely as described in [O'Brien, et al. 86]; however, dbObjects are new. A repository is a "database", in that it provides an address space of persistent, uniquely identified, objects. Multiple repositories cannot share objects in any direct way, and are not coordinated on access or update. A dbCollection is essentially a set of similarly typed objects, with a name to allow access once the containing repository has been opened for use. A dbObject holds just a single named object. Both dbCollections and dbObjects are primarily "handles" by which an application gains access to previously stored objects. However, the objects themselves, and their reachable components, are retrieved automatically as they are accessed – one does not "read" or "write" a dbCollection or dbObject. Furthermore, dbCollections and dbObjects can share persistent objects, just as objects can be shared in a normal Owl address space.

In addition to being handles, dbCollections and dbObjects can have clustering rules tailored for their "contained" object(s). These rules override those of the repository as a

whole. This mechanisms works as follows. When a transaction commits, we perform a recursive trace starting from the dbCollections and dbObjects the transaction accessed. When we encounter a *new* object that has been linked into the persistent data structure – that is, an object that does not yet have a forwarder – we construct a forwarder, form a pid for the object, and insert the forwarder on the appropriate hash chains.

In forming the pid, we need to allocate the new object to a specific group; here is where the clustering rules come into play. The applicable rules are first, those of the dbCollection or dbObject being recursively traced, then, those of the repository as whole. Let us use the term *parent* for the just traced object that contained the reference to the new object. If the parent's type has a component rule for the component in question, then that rule is used to determine the cluster for the object. Otherwise, we look for a rule for the new object's type. If no rule is found, the default cluster is used.

Once a cluster has been chosen for the new object, we must choose a specific group within that cluster. If the parent is in the same cluster, then we will choose the parent's group if there is room. Otherwise, each cluster has a default group being filled at any given time, and the object will be placed there if possible. If the default group is too full, we start a new default group, and so on. Note that the clustering rules can use context, even though they are static and declarative. In the near future we hope to begin serious evaluation of the effectiveness and convenience of this approach to clustering. However, the clustering would seem to work well when new objects can be reached from only one dbObject or dbCollection, but provide little control when new objects are reachable from more than one source: the first source traced will "claim" the new objects. We must perform experiments to see whether this is unsatisfactory.

6 Future Plans

In the immediate future we will complete the prototype, measure its performance, and evaluate the results. We are also starting a parallel project to build a more generic persistent object world that could be used from more traditional languages, such as Ada. This might not have the convenience of the tight integration seen with Owl, but is an interesting alternative to using files or full blown databases. We hope it will be a useful step on the way to more integrated environments and object oriented databases.

In addition to exploring interfaces to traditional languages, we intend to build a version of this system integrated into a Smalltalk system currently under construction. The Smalltalk system uses an object table, so some of the detailed implementation techniques are expected to be different from those used for Owl, and we hope more elegant because

of that. We further intend to examine techniques for storing objects that do not involve uid conversion between an external, persistent form and an internal, non-persistent form.

Finally, since our long range goal is support for design applications, we wish to extend the model, while maintaining language/database integration. Not only will we extend the data model, transaction model, and so on, but efficient storage support will be needed such as B-trees and other external storage organizations. To that end we are currently evaluating several other projects' software for potential use in a later version of our system. The candidates under active consideration include Exodus [Carey, et al.], Genesis [Batory, et al. 86], and Camelot [Spector 87].

References

[Batory, et al. 86] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twichell, T. E. Wise, "Genesis: A Reconfigurable Database Management System", Department of Computer Sciences, University of Texas at Austin, TR-86-07, March 1986.

[Carey, et al.] Michael J. Carey, David J. Dewitt, Joel E. Richardson, Eugene J. Shekita, "Object and File Management in the EXODUS Extensible Database System", Computer Sciences Department, University of Wisconsin, unnumbered technical report, undated (probably 1986).

[Goldberg and Robson 83] Adele Goldberg and David Robson, *Smalltalk-80: The Language and its Implementation*, Addison-Wesley, 1983.

[Liskov, et al. 1977] Barbara Liskov, Alan Snyder, Russell Atkinson, and Craig Schaffert, "Abstraction Mechanisms in CLU", *Communications of the ACM*, Volume 20, Number 8, August 1977, pp. 564–576.

[Liskov, et al. 1981] B. Liskov, R. Atkinson, T. Bloom, E. Moss, J. C. Schaffert, R. Scheifler, A. Snyder, *CLU Reference Manual*, Springer-Verlag, 1981.

[O'Brien 85] Patrick O'Brien, "Trellis Object-Based Environment: Language Tutorial", Version 1.1, Eastern Research Laboratory, Digital Equipment Corporation, Technical Report 373, November 1985.

[O'Brien, et al. 86] Patrick O'Brien, Bruce Bullis, and Craig Schaffert, "Persistent and Shared Objects in Trellis/Owl", *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Asilomar, CA, September, 1986, (IEEE Computer Society Order Number 734), pp. 113-123.

[Schaffert, et al. 85] Craig Schaffert, Topher Cooper, Carrie Wilpolt, "Trellis Object-Based Environment: Language Reference Manual", Version 1.1, Eastern Research Laboratory, Digital Equipment Corporation, Technical Report 372, November 1985.

[Schaffert, et al. 85] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, Carrie Wilpolt, "An Introduction to Trellis/Owl", *OOPSLA '86 Conference Proceedings*, Portland, OR, September-October 1986, pp. 9-16, available as *ACM SIGPLAN Notices*, Volume 21, Number 11, November 1986.

[Skarra, et al. 86] Andrea Skarra, Stanley B. Zdonik, Stephen P. Reiss, "An Object Server for an Object-Oriented Database System", *Proceedings of the 1986 International Workshop on Object-Oriented Database Systems*, Asilomar, CA, September, 1986, (IEEE Computer Society Order Number 734), pp. 196-204.

[Spector 87] Alfred Z. Spector, "Distributed Transaction Processing and the Camelot System", Carnegie-Mellon University, Technical Report CMU-CS-87-100, January 1987.