Persistent Owl:
Heap Management and Integration with Mneme*

Vassiliki Christou *J. Eliot B. Moss* †

COINS Technical Report 88-78
August 1988

Object Oriented Systems Laboratory
Department of Computer and Information Science
University of Massachusetts
Amherst, MA 01003

Abstract

We present the integration of Trellis/Owl[1], a heap-based, object oriented language, with Mneme, a persistent object manager. In this document we discuss general issues of smooth integration of heap-based languages with persistent object managers, and eventually we focus on problems particular to Trellis/Owl, such as changes in the language from the user's point of view, heap management, modifications in the compiler and run-time system, and interface with Mneme.

*This project is supported by National Science Foundation Grants CCR-8658074 and DCR-8500332, and by Digital Equipment Corporation, Apple Computer, Inc., GTE Laboratories, and the Eastman Kodak Company.

†Authors' present address:: Department of Computer and Information Science, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA, 01003; telephone (413) 545-4206; Internet addresses Christou@cs.umass.edu and Moss@cs.umass.edu.

[1]Trellis is a trademark of Digital Equipment Corporation.

1 Introduction

In this paper we discuss the integration of Trellis/Owl [Schaffert *et al.*, 1986], a strongly typed, heap-based, object oriented language, currently under development at Digital Equipment Corporation, with Mneme [Moss and Sinofsky, 1988], a persistent object manager, currently under development at the University of Massachusetts. Although the work was specifically intended to integrate Trellis/Owl (hereafter referred to as Owl) with Mneme, and although parts of the design were necessarily affected by some specific features of the implementation of Owl, many of the design issues apply to all heap-based languages, and are independent of particular language features. For this reason, we will defer the discussion about Owl-specific issues until after the general approach to integrating heap-based languages with Mneme has been described.

The rest of this paper is organized in the following way: Section 2 offers some background information on heap-based languages, and provides a brief description of Mneme. Section 3 presents some of the objectives of Persistent Owl, and gives a brief scenario of a user session on the system. Section 4 addresses some of the problems that arise from the requirements we place on the persistent system, and essentially constitutes the design of the persistent heap manager. Section 5 discusses the issues that are particular to Owl Finally, the last section describes the steps that should be taken by the implementor of the Persistent Owl system to obtain a running prototype. The first three sections do not require familiarity with any particular heap-based language. In the fourth section we make the discussion slightly more Owl specific, since some design decisions are dictated by the internal format of Owl. In the last two sections we describe modifications to the existing Owl system, so some familiarity with the Owl language is assumed.

2 Background

Before we start the discussion about persistent systems we present some background information about heap-based languages, and a brief description of Mneme.

2.1 Heap-based Languages

Heap-based languages are programming languages organized around automatic storage allocation and reclamation. Objects are interconnected in the heap and can refer to each other in arbitrary ways. Directed graphs, both with and without cycles are allowed, although the language system has a set of root objects from which all other objects can be reached. Objects remain in the heap for as long as is necessary, and space is typically reclaimed by some form of garbage collection. Since the heap is a general graph, and since we need to traverse its nodes in arbitrary ways, following pointers is one of the most common operations of heap-based languages.

Figure 1 offers a pictorial model of some objects in the heap.

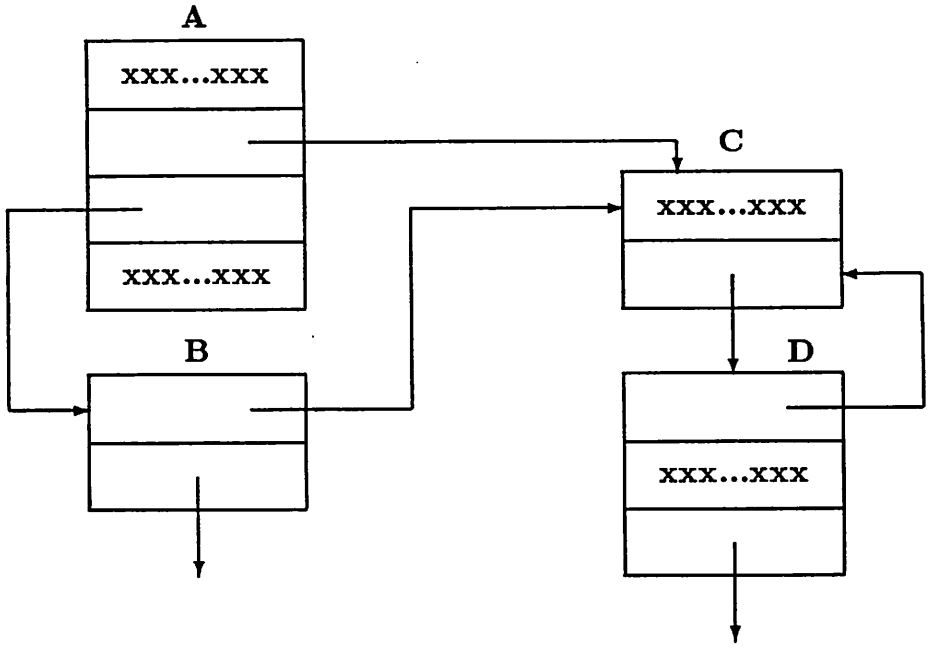

Figure 1: Objects in the heap

2.2 Mneme

Mneme is a storage manager that provides persistent object management for programming languages. The purpose of Mneme is to replicate the client language heap, but it must do that in a persistent way. However, unlike objects in the source languages it supports, Mneme objects are typeless. Each Mneme object consists of *slots* and *bytes*. Slots contain either references to other objects, or immediate values, such as small integer numbers. Mneme organizes its objects in *files*, and can operate on multiple files, each one having its own *root*.

One of the primary goals of Mneme is to provide support for a variety of programming languages, from traditional, such as Ada and C, to less traditional, such as Owl and Smalltalk-80 [Goldberg and Robson, 1983]. Another goal is to provide abstraction between the layers and not to reveal details of the internal representation of objects. Although we aim for an integration that is smooth for the programmer of the language, an object manager with such goals poses a restriction in the run-time system of any language that uses it: integration has to be loosely instead of tightly coupled, so that it will not compromise the abstraction of the layered design.

Figure 2 presents the overall architecture of Mneme and shows how client languages such as Owl relate to it.

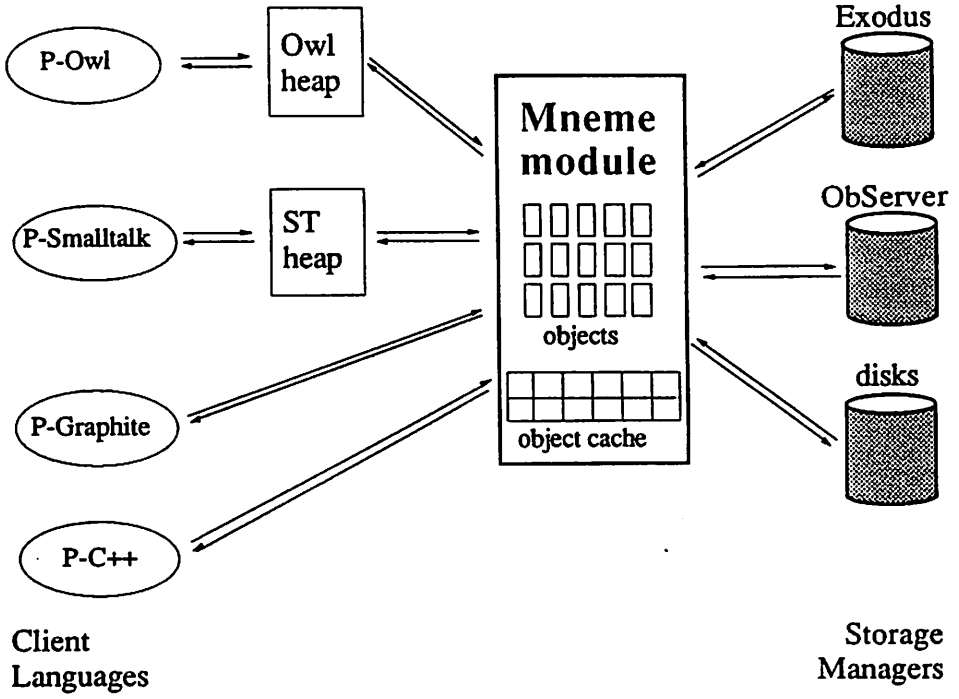

Figure 2: Mneme topology

3 Objectives of Persistent Systems

We begin by presenting the objectives of our persistent systems from the point of view of the user (functionality), and then we describe some language issues that arise from these objectives.

3.1 Orthogonal Persistence

In typical heap-based languages, such as Smalltalk and Owl the workspace is kept in main memory. Objects cannot be shared between users, and there is no persistence, other than saving snapshots of workspaces. We would like to to provide a shared space of persistent objects, as the one presented by Mneme. Our major goal is to make persistence and sharing as transparent to the user of the client language as possible. In that respect, our model resembles that of PS-Algol ([Atkinson *et al.*, 1981] and [Atkinson and Morrison, 1985]). Unlike Argus [Liskov and Scheifler, 1983] and Avalon ([Herlihy and Wing, 1986] and [Detlefs *et al.*, 1987]), where objects must be designated as persistent or shareable at creation-time, our persistent languages permit the user to decide at any point to make

objects persistent. Persistence independent of the type is *type-complete*. Since objects can become persistent at any time, the persistence offered is *dynamic*. Type-complete, dynamic persistence is *Orthogonal Persistence*.

3.2 Language Impact

Adding persistence to any programming language usually requires changes to the language. Some of the changes have more and some have less of an impact at the level of the user/programmer. One of the goals in our persistent systems is to make the persistent language as close in efficiency to the non-persistent as possible. To obtain better performance, we believe that it is better to create copies of the persistent objects in the heap in a way that is compatible with the previously existing language data structures. By replicating the language heap from persistence storage, and vice-versa we also achieve the goal of modifying the client language minimally. Although the method chosen requires more space, we believe that in a space-efficiency trade off, we can always trade space to obtain a faster running system. Another goal in providing transparent persistence is to shield the user from as many of the effects as possible. In the following discussion, we divide the persistence features into three categories: features that can be completely hidden from the user, features that are presented to the user as options, and features that the user must be aware of and use to obtain persistence.

3.2.1 Transparent Operations

There are several persistence operations that can be hidden from the user completely:

- Reading persistent objects.

  Accessing of data is the most common operation in any system. In non-persistent systems all objects reside in main memory and are readily accessible to the user. In a persistent system, however, requested objects are not always available. The heap manager must identify the presence or absence of an object in memory by a faulting mechanism, similar to page-faulting in operating systems, which is transparent to the user. In our system, faulting is based on individual objects instead of pages, and it is called *object-faulting*.

- Writing persistent objects.

  Objects become persistent when they are written from main memory to stable storage through Mneme. Not all heap objects are written back however; some describe a volatile state of the run-time system, and cannot be reached at write-back time (they are garbage), or, even if they could, they would not be useful for future references, so they should not be stored. Newly created objects must also be identified and written into permanent storage. The heap manager takes care of the write-back details without user intervention.

- Reclaiming space.

  Several data structures must be added to the heap manager to obtain persistence. Since the space in the heap is reclaimed by garbage collection, there is clearly a need to reclaim these new structures as well. The garbage collector should make the process of reclamation transparent to the user.

3.2.2 Options Open to the User

Since requested objects are not always in the heap, but may be faulted in from storage, it is highly advantageous to minimize the number of secondary storage accesses. Mneme supports a wide range of object management policies, and allows each user application to choose its own policy. Users wishing to take advantage of these features must be aware of some aspects of the underlying object management system to be able to use them. Some of the options available to the user might be as follows:

- Prefetching

  Objects might be prefetched instead of brought-in on demand. If the application uses a set of data in a predictable way (e.g. accesses a lot of data linearly), then some of these data can be brought in by Mneme while the heap manager is working on previously fetched data. This policy could improve the performance of the system in terms of accessing time per object.

- Clustering

  Another way that an application can take advantage of Mneme policies is by grouping objects according to declarative rules expressed in the source language. A whole cluster of objects can be brought into the heap at the same time, according to application-specific clustering rules. We intend to pursue the clustering option in the implementation of persistent Owl. Our decision is based on the fact that unless the way objects are going to be accessed is simple and known in advance, prefetching would not be very effective. Clustering, on the other hand, is controllable by the user, and every application program may have a different way to cluster objects. However, we recognize that this grouping technique will require more extensive changes in the source language than we would like to make at this point. For that reason, we have not used clustering in our prototype design.

3.2.3 Non-transparent Operations

Sharing and persistence add some additional requirements to the part of the system that interacts with the user. Objects should not be updated, or even accessed, without some control mechanism that enforces consistency. Persistent objects must be atomic and durable. They must either be completely updated, or not modified at all, and once updated the objects must survive system crashes. Also, for consistency, objects in the process of being updated by one user must not be accessed by another. A *transaction* is introduced as

the basic unit of sharing and reliability. Objects can only be accessed during transactions. If the transaction commits, then all the updates become permanent. If it aborts, then none of the intermediate states become visible. An object held and updated by one transaction cannot be accessed by another. Users must be aware of the semantics of transactions and in some cases they may have to explicitly begin, commit, and abort transactions in their application programs.

3.2.4 Example of an Interactive Session

To illustrate the previous discussion of transparent and non-transparent operations, and to give to the reader an idea of the operations performed when objects are accessed, we present a simple example. In our example the application program adds an integer number into a set of integers called **ints**. We avoid any language specific-details, but since this is part of an interactive session, we assume that some preliminary work has been done. Depending on the transaction semantics in the client language, some of the statements, such as **begin** and **commit** may or may not have to be explicit.

The user might have been interacting with the system previously, so some of the initial details are omitted.

```
...
...
transaction_begin (...)
insert (ints, 101)
```

> At this point the set **ints** is updated. If it is not resident, then it should be faulted into memory, without notifying the user. The user does not have to explicitly lock the object. If another transaction is using it, then it will become available by the end of the other transaction.

```
transaction_commit(...)
```

> Updates are made permanent at this point without any explicit write-back commands. All the objects are released.

```
...
```

4 Design

We present a discussion of the specific problems we have encountered in the integration of heap-based languages with Mneme, and lay out the proposed solutions to these problems. Finally, we describe briefly the heap management algorithms.

4.1 Problems

4.1.1 Arbitrary interconnections

In non-persistent systems, all objects are created and manipulated in main memory, so the complex interconnections between objects do not affect the performance of the system. In persistent systems, however, objects must be brought in from secondary storage. We need a mechanism to cause faults when objects are not in the heap. Also, once an object is faulted, we do not want to follow its references to other objects, or we would end up bringing in potentially unnecessary parts of the store. This is neither desirable nor practical. We would like to be able to fetch objects on demand, since in a typical session many of the existing objects will not be referenced.

4.1.2 Format Conversion

Objects in both Mneme and its client languages need to be identified in a a way that guarantees uniqueness. Mneme uses two kinds of identifiers: Persistent IDentifiers (PIDs), which are unique for an object within a Mneme file, and Client IDentifiers (CIDs), which are unique for an object in the heap during one transaction. Mneme automatically translates from PIDs to CIDs, and the user of the language is never obliged to see PIDs. The need for translation arises because PIDs are unique only within files, but there can be more than one file in use at the same time, and so objects in different files may have the same PID. Mneme provides unique identifiers, and the translation between PID and CID is relatively cheap.

Another level of translation is required between language references and CIDs. The client languages use language references, instead of CIDs, and, depending on the language they may be different. In the case of Owl the language references are addresses in the heap. We need a way to go from language references to CIDs and vice-versa. This translation is harder and more expensive than the PID-CID conversion, and since it is language specific it must be done by the client language rather than by Mneme.

4.2 Forwarders

In this and the following subsections we discuss a solution to the above mentioned problems and the heap manipulation algorithms as they were dictated by the implementation of the Owl language. For that reason the discussion here is particular to Owl, but some of the ideas presented, with appropriate modifications, should apply to most heap-based languages.

4.2.1 The Concept of Forwarders

Forwarders are invented to provide information to the faulting algorithm and help solve the problems of format conversion. A *forwarder* is a memory resident data structure that provides information about a persistent object, such as whether the object is resident, the

address and the CID of the object, etc. When an object is brought into the heap, we create a forwarder reference for each of its components. The *forwarder reference* is an index into a table of forwarders. A forwarder points to the target object if it is in memory, or contains information needed for its retrieval.

Figure 3 shows an example of some objects accessed through forwarder references.

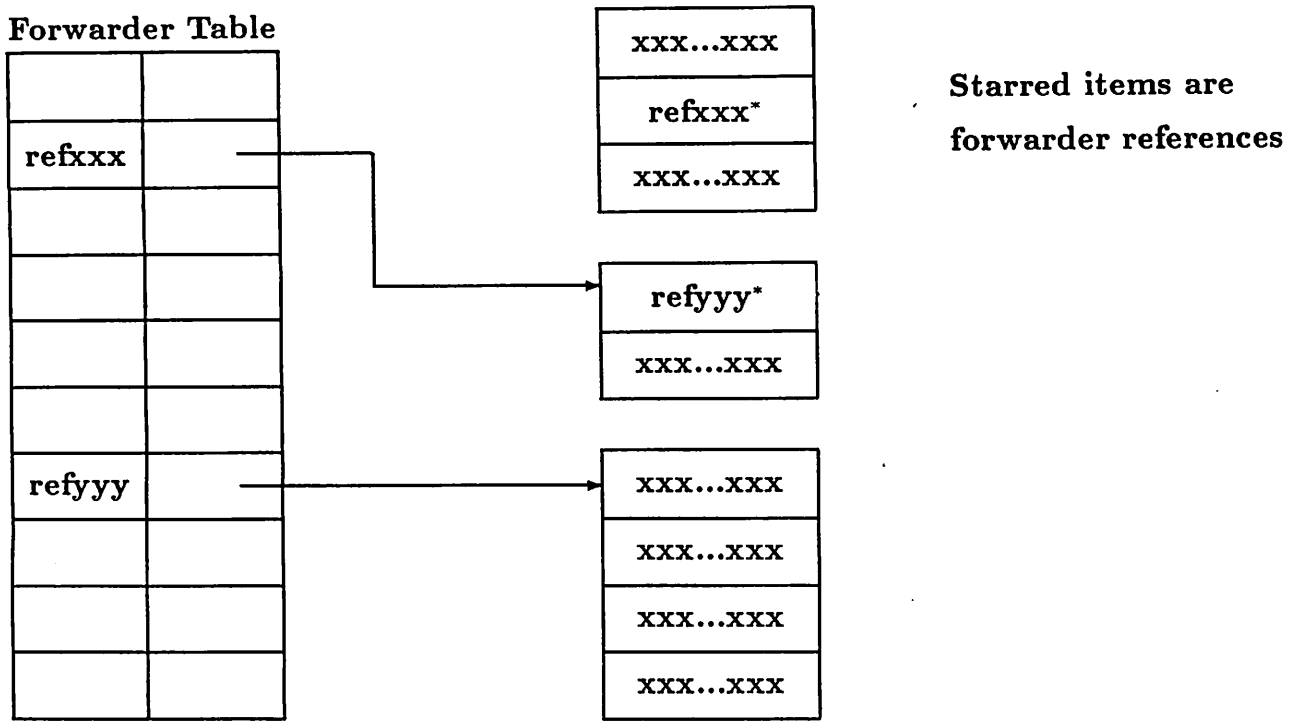

Figure 3: Use of Forwarders in Persistent Heap

4.2.2 Physical Description of Forwarders

Each forwarder consists of two long words. The first long word is the address of the object in the heap (language reference) if the object is resident, or zero otherwise. The first two bits in the second long word are reserved to indicate information about the object. The first is the Resident bit which is used to indicate whether the object is known to be in the heap, and the second is the Redundant bit. The importance of this piece of information is described in the next subsection. The rest of the second long word holds the CID of the object.

Figure 4 illustrates the format of a forwarder.

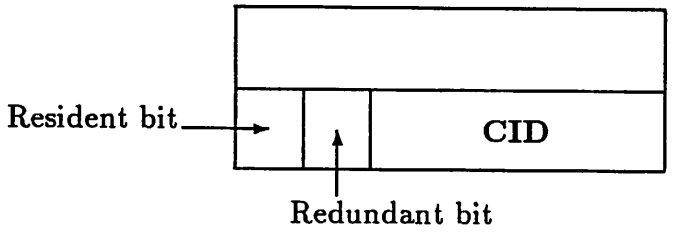

Figure 4: Example of a forwarder

4.3 The Use of Forwarders

A point of interest in the design of fetching and writing back is that one target object can have more than one forwarder that points to it. Since the object interconnection graph is arbitrary, objects can be reached via more than one path. For the semantics to be correct, however, it is essential that we have only one copy of an object in the heap. We have decided to allow non-unique forwarders to be created because we believe that, on the average, the operation of creating forwarders will be more frequent than the operation of accessing the forwarder target objects. Therefore, we would like to make the forwarder creation operation as fast as possible. Creating forwarders without testing for uniqueness is very fast, so we have decided to create forwarders without testing, and test for uniqueness of the target object only when the forwarder reference is followed. The extra space consumed by the redundant forwarders is not excessive, and we prefer to sacrifice some space in order to increase efficiency.

Figure 5 shows an example of a case where multiple forwarders are created. In this case when object A was retrieved the forwarder references to all its component objects were created, including the one to object C. The same thing happened when object B was retrieved. Thus, there are now two forwarders to object C.

According to the discussion above, when an object is referenced, the search for it might result in one of the following three situations:

1. The object is not resident. In this case the Resident bit in the forwarder is zero.
2. The object is resident, but became resident through a different forwarder. Just by looking at the forwarder, however, we cannot distinguish this case from the previous one, because the Resident bit is still zero.
3. The object is resident, and we know it as such (because the Resident bit is set).

Case 3 is the simplest, because the address of the object is already in the appropriate forwarder word, and we can simply use this address to access the object. Cases 1 and 2 cannot be distinguished by examining the forwarder. In order to determine which situation we have encountered we require some extra information. For this purpose Mneme provides

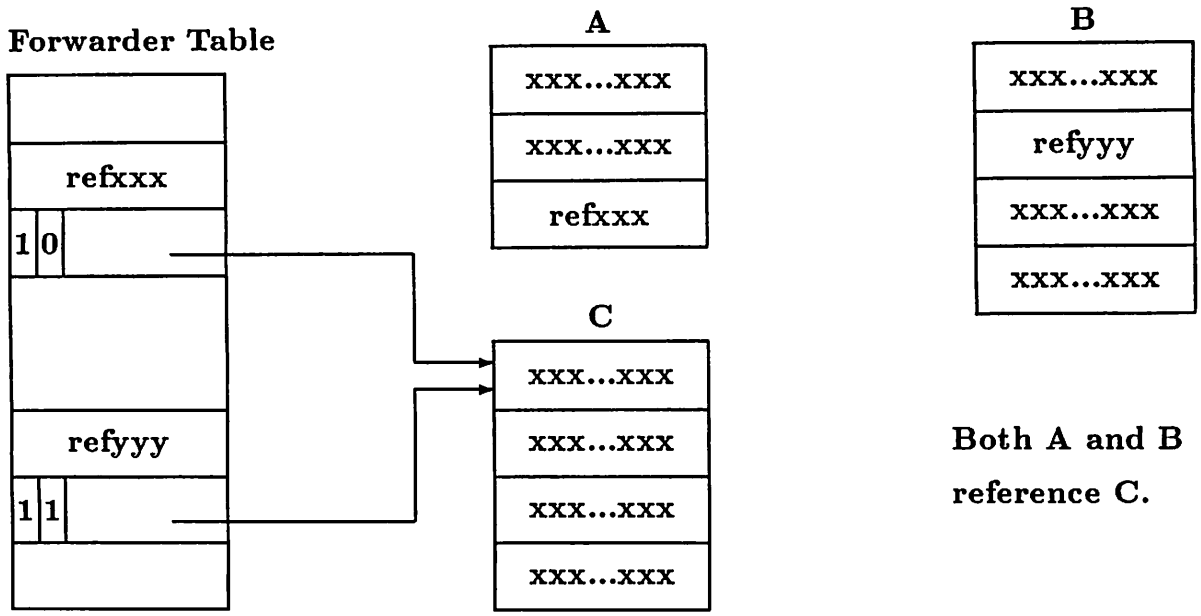

Figure 5: Multiple Forwarders

its clients with *transaction information* (TnxInfo) for every object that is accessed during a transaction. Mneme users set and use this field according to their needs. TnxInfo is zero if the object has not been retrieved, or it is the address of the forwarder if it has been retrieved through another forwarder. If the object was not brought into main memory, then we have case 1, and we ask Mneme to bring the object in. The TnxInfo field of the object is then set to the object's forwarder address, the Resident bit is set to 1, and the Redundant bit is set to 0. If the object was already resident, then we do not update the TnxInfo field since it already contains the information we need. In that case the Resident and the Redundant bit are both set to 1. When the Resident bit of the forwarder is set to 1, we say that the link is *snapped*. The Redundant bit indicates whether the forwarder to the object is the first to be snapped This piece of information is not important in the fetching phase, and it is not absolutely necessary later either, since we can always examine TnxInfo to find out whether the forwarder is redundant. It is however useful during write-back. In the write-back phase we wish to write an object only once, so only resident objects referred to by non-redundant forwarders are written back. If we did not use the Redundant bit we would have to examine TnxInfo in processing each forwarder. As it is, we substitute a simple bit test for a Mneme call, and the set up has negligible cost.

Another interesting situation occurs when new objects are created. When a new object is created, it is pointed to directly, and not through a forwarder. At the end of the transaction, references to new objects can be distinguished from forwarder references because they have different tag bits. New objects may also be referenced by multiple objects, so again, we must be able to ensure uniqueness. Mneme cannot help us in this case, because we need to translate from language references to CIDs, and not vice versa. New object

references are therefore inserted into a *new object table*, and the test for uniqueness is done by examining this table.

4.4 Algorithms

There are three different actions supported by the heap manager: fetching, updating, and writing back. At the beginning of a transaction a forwarder is created for the root object. Once objects are referenced we fetch the root object itself and create forwarders for all the references to its components. Forwarders are stored linearly in the *forwarder table.*

4.4.1 Fetching

The following sequence of actions explains how requests to objects are handled:

```
An object is requested.
The forwarder reference to it is used to index the forwarder table.
/* The forwarder is examined. */
If the resident bit is 1, Then
  follow the reference and reach the object.
Else
  Give the CID to Mneme and get the TnxInfo for the object.
  If TnxInfo is a forwarder reference, Then
    /* The object has been reached before. */
    Copy the forwarder reference into the
      appropriate forwarder word.
    Update the forwarder to indicate that:
      1. The object is now resident.
      2. The forwarder is redundant.
  Else
    /* The object is not in the heap. */
    Ask Mneme for the object and write the reference in the
      appropriate forwarder word.
    Construct the object in the heap requesting object data from Mneme.
    Update the TnxInfo field.
    Update the forwarder to indicate that:
      1. The object is now resident.
      2. The forwarder is not redundant.
```

4.4.2 Updates

Updates are performed as usual. New objects might be created, but they do not become persistent until the transaction commits, and the heap is written back to Mneme.

4.4.3 Writing back

The write-back algorithm is basically a linear scanning of the forwarder table. There are two important points that the algorithm must handle correctly: first, only non-redundant forwarders should be processed, and second, new objects should be identified and written back.

The following sequence of actions is performed:

```
/*  We scan the forwarder table. */
 For every entry in the forwarder table do
   If it is redundant or not resident then
     do nothing
   Else
     For every field in the object do
       If the field is a forwarder reference then
         /* This field does not point to a new object. */
         Give the CID in the forwarder back to Mneme as the
           value for the field. (Mneme converts the CID to PID.)
       Elseif the field is an immediate value then
         Give field to Mneme for write-back.
       Else
         If the field names an object in the New Object Table then
           Give the CID to Mneme.
         Else
           Create a new forwarder for the new object and put
             the forwarder at the end of the forwarder table.
           Mark the forwarder as resident and non-redundant.
           Create a corresponding Mneme object.
           Fill-in the forwarder CID slot with the object CID
             obtained from Mneme.
       End if;
     End for;
   End if;
 End for;
```

Notice that there is no need to create more than one forwarder for a new object because we ensure uniqueness when we create the object. The reason that new forwarders are marked non-redundant is because it is necessary for the algorithm that searches the forwarder table to process the new object. By adding the new forwarders at the end of the table we insure that the forwarder will be processed eventually and thus, that all new objects will be given to Mneme. At the end of a write-back (i.e., when a transaction commits), the forwarder table is invalidated, and all the heap information in terms of language references and CIDs becomes garbage. When a new transaction begins, all the steps of fetching and updating are repeated with a new table. We would like to be able to re-use

some of the table information and avoid refetching all the objects from Mneme for every transaction, but at this point we do not have any satisfactory technique of doing that, and this matter is still under investigation.

5 Design of Persistent Owl

In the design that is particular to Persistent Owl we have added some language specific characteristics. Also, it is worthy to note that some of the features in the implementation of Owl dictate solutions that will differ from those of other languages, such as Smalltalk. Although we strive for a minimal number of modifications to the client language, there is still the need to modify Owl at two different levels: the user level and the compiler/run-time system level.

5.1 Owl Library Additions

Owl, like most object-oriented languages consists of some base types, and allows the users to build their own types according to their needs. A predefined Owl library was built by the language designers to provide some of the base types and allow the creation of new more complex types. It is convenient to introduce a few more new Owl library types for the application programs that use the persistent system.

We define a new type Repository with which database semantics are associated. All persistent objects reside in repositories. In our design, repository contents are stored and retrieved from files, but repositories are active objects containing not only data, but transaction information as well. Although Mneme can operate on multiple files, the design for Owl assumes that one user can have only one repository open at a time, so files and repositories are in a one-to-one correspondence. Objects are organized in repositories in the following way: every repository contains a Root object from which all the persistent Owl objects can be reached. The Mneme file also contains a root object, which has the user root (Owl root) in one of its slots. The Owl root can thus be obtained from the Mneme root. In general the Mneme root contains more information, but we do not wish to make this information visible to the language user.

Users can define their own root objects according to their applications. However, since objects have to be referred to somehow, it seems reasonable to have named objects in the root, and be able to reach them by their name. In this case the root object should be, or at least should contain as a component, one or more dictionaries, where the lookup is done by object name. Users can insert and delete named objects in the dictionary. They can also perform other dictionary operations, such as getting the number of entries, looking up an object, etc.

In a future design, repositories will also contain default clustering information. If objects do not carry their own clustering information, then the placement mechanisms will search in the repositories for default clustering rules.

5.1.1 Description of the New Types

We present the specifications of the new library objects. This description is by no means exhaustive. It is only intended to clarify some of the concepts mentioned before, and as a means to describe briefly some of the operations that take place when the application programs use Persistent Owl.

In the description of the new Owl objects the keyword Mytype indicates a class operation, while the keyword me indicates an instance operation. For brevity we have not shown error conditions in the description of the new library types.

- Repository
  Type_module Repository[Root: Type]

```
component me.name: String;
component me.root: Root;
component me.transaction_active?: Boolean;
operation create (Mytype, name: String) returns Repository;
operation destroy (Mytype, name: String);
operation open (Mytype, name: String) returns Root;
operation close (Mytype, name: String);
operation transaction_begin (me);
operation transaction_commit (me);
```

  end type_module;

- Root

  The Root object is defined by the user. A create operation will certainly be required.

- Dictionary
  User_dictionary[ValueType: Type]

```
component me.valseq: Vector[ValueType];
component me.capacity: Integer;
component me.size: Integer;
operation create (Mytype) returns (Mytype);
operation insert (me, key, val);
operation remove (me, key: String);
operation lookup (me, key: String) returns ValueType;
operation getsize (me) returns Integer;
operation getcapacity (me) returns Integer;
operation indictionary? (me, key: String) returns Boolean;
```

  end type_module;

Our decisions about the new library types came after experimenting with Owl programs that provided the desired functionality in the non-persistent system. We feel that the programmer should be free to select the root object, because this does not impose any specific database model on the users of the system. Also, in providing a dictionary with the type of the object as a parameter we allow the users to select any type of objects they wish, but still enforce homogeneity in the entries, and enforce a more strict error checking than if we merely provided a typeless dictionary of generic objects.

5.2 Modifications to the Owl System

In addition to the compiler and run-time system functions that need to be added or modified, some modifications in the representation of Owl objects are also required, since distinguishing one kind of an object from another by examining the object representation plays an important role in the persistent system.

Owl currently uses some of the bits in every slot to tag values in slots. Small integers are immediate values 31 bits long. Combinations of some reserved bits indicate whether the object is an integer or a reference. In the persistent system we will have integers, references, and forwarders. We plan to shrink the range of integers by one bit, obtaining thus an extra bit for our tag bits. The first few reserved bits are used to determine what kind of object we have encountered. In case of a forwarder reference, the rest of the bits are the index in the forwarder table.

The forwarder table is simply an array of forwarders and can be implemented either as a fixed array with some predefined maximum size or it can be extensible. The new object table is a hash table where the Owl IDs of new objects are stored. Both of these tables are in the heap, but they are not Owl objects.

6 Implementation issues

In this section we offer a brief technical discussion addressing some implementation questions. The steps needed to implement the design are described and some performance measurement strategies are proposed.

6.1 Additions to the Owl Library

The implementation languages chosen are both Owl and C. Owl is used for the implementation of the Owl library types, although the internal parts of library types (builtins) are written in C. C is also used as the heap manipulation language, since both the Owl system and Mneme are written in C, and we have flexibility and smoother integration this way. Most of the Owl features mentioned in the previous section are already implemented in Owl. We also expect to use both Owl and C in the implementation of clustering rules, because as declarative rules in the Owl application program they need to have an Owl part, and as object placing mechanisms in the lower level they need to have a C part.

6.2 Heap Management and System Modification

Currently, both the compiler and the run-time system operate under the assumption that all objects are in the heap. Modifications must be made in the compiler to generate code for the object faulting. When a call to an operation is made the class of the object needs to be checked. In some cases type checking is static, but in some others it is dynamically bound. In the dynamic case we need to know the class of the object, which means that we should fault and bring the object in to examine the class. We consider having two entries in the compiler generated code, one for static class checks and one for dynamic class checks.

The garbage collector needs to be modified as well. It must know about the new data structures we are introducing so that it can reclaim space once they become garbage. Another useful operation we consider adding to the garbage collector is to snap the links in redundant forwarders. The garbage collection routines might examine the transaction information for every CID and if the object has been reached via another forwarder they can add all the information we need in the forwarder of the object.

Another task to be done is to separate user from system data. We would like to make only useful data persistent, but the run-time system creates instances of several classes we can not store successfully. Many of these objects keep track of volatile information, such as activities, locks, and duration times. Saving the status of these objects between invocations is meaningless. A tag bit combination may be reserved in the class field to indicate whether the object is appropriate for storage. In this case the write-back algorithm should be modified to use these bits when deciding which objects to write back.

6.3 Performance Measurements

One way to take some performance measurements is to add an application program to use the newly provided functions. The experience of developing and using a test program will be a reasonable indication of the reaction of programmers and users to such an environment. At the same time, the store manager can be instrumented to keep track of the number of times objects are accessed, the number of objects fetched and written back, the time it takes for resident and non-resident objects to be accessed, etc. Other interesting measurements are: the number of forwarders created versus the number of objects actually used, the time to create a forwarder versus the time to ensure uniqueness for a given object, and finally, the total run time of the system, as an indication of the overhead of the persistent heap. Eventually, when clustering mechanisms are added it will be useful to cluster data in more than one way in the application level, and test the performance of Mneme and the object server in fetching and writing back blocks of objects.

References

[Atkinson and Morrison, 1985] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems 7,*

4 (Oct. 1985), 539–559.

[Atkinson *et al.*, 1981] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a peristent heap. *ACM SIGPLAN Notices 17*, 7 (July 1981).

[Detlefs *et al.*, 1987] David Detlefs, Maurice Herlihy, and Jeannette Wing. Inheritance of synchronization and recovery properties in Avalon/C++. Tech. Rep. CMU-CS-87-133, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Mar. 1987.

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

[Herlihy and Wing, 1986] Maurice Herlihy and Jeannette Wing. Avalon: Language support for reliable distributed systems. Tech. Rep. CMU-CS-86-167, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, Dec. 1986.

[Liskov and Scheifler, 1983] B. Liskov and R. Scheifler. Guardians and actions: Linguistic support for robust distributed programs. *ACM Transactions on Programming Languages and Systems 5*, 3 (July 1983), 381–404.

[Moss and Sinofsky, 1988] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Issues and application of a reliable, shared object interface. COINS Technical Report 88-30, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Apr. 1988.

[Schaffert *et al.*, 1986] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 9–16.