The Persistent Object System MetaStore: Persistence via Metaprogramming

Arthur H. Lee

UUCS-92-027

Department of Computer Science
University of Utah
Salt Lake City, UT 84112 USA

June 10, 1992

Abstract

Object-intensive applications require persistence of complex objects. Many of these applications also use vast amounts of data, often exceeding a machine's virtual memory. *MetaStore* is a portable, persistent object system designed to solve these problems.

MetaStore uses the metaprogramming facilities of the metaobject protocol to add persistence to the Common Lisp Object System. This approach leaves the semantics of CLOS unchanged, requires only minimal syntactic changes to existing programs, and needs no compiler support. In the resulting language, programmers can define both classes and slots to be persistent. MetaStore then handles persistence at the metaobject level.

MetaStore focuses on the persistence of passive data by providing a virtual object memory. It tries to keep an appropriate number of objects in memory to maintain system performance at an acceptable level, and allows programmers to tune performance by specifying object and slot clustering. MetaStore currently supports only limited aspects of class evolution.

MetaStore addresses a range of issues in implementing persistence, including object identity, addressing mechanisms, shared objects and structures, dirty bits, queries, transaction management, version control, object base garbage collection, and object clustering.

Based on the initial implementation of MetaStore, we show performance measurements and propose improvements to the metaobject protocol of CLOS. CDRS CAD system, an object-intensive application at Evans & Sutherland Computer Corp., will soon be ported to MetaStore.

The Persistent Object System MetaStore: Persistence via Metaprogramming

by

Arthur H. Lee

A dissertation submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

Department of Computer Science

The University of Utah

August 1992

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a dissertation submitted by

Arthur H. Lee

This dissertation has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

June 23, 1992 — Chair: Joseph I. Zachary

June 25, 1992 — Robert R. Kessler

June 17, 1992 — Gary E. Lindstrom

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of The University of Utah:

I have read the dissertation of Arthur H. Lee in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to the Graduate School.

June 23, 1992
Date

Joseph L. Zachary
Chair, Supervisory Committee

Approved for the Major Department

Thomas C. Henderson
Chair/Dean

Approved for the Graduate Council

B. Gale Dick
Dean of The Graduate School

Abstract

Object-intensive applications require persistence of complex objects. Many of these applications also use vast amounts of data, often exceeding a machine's virtual memory. *MetaStore* is a portable, persistent object system designed to solve these problems.

MetaStore uses the metaprogramming facilities of the metaobject protocol to add persistence to the Common Lisp Object System. This approach leaves the semantics of CLOS unchanged, requires only minimal syntactic changes to existing programs, and needs no compiler support. In the resulting language, programmers can define both classes and slots to be persistent. MetaStore then handles persistence at the metaobject level.

MetaStore focuses on the persistence of passive data by providing a virtual object memory. It tries to keep an appropriate number of objects in memory to maintain system performance at an acceptable level, and allows programmers to tune performance by specifying object and slot clustering. MetaStore currently supports only limited aspects of class evolution.

MetaStore addresses a range of issues in implementing persistence, including object identity, addressing mechanisms, shared objects and structures, dirty bits, queries, transaction management, version control, object base garbage collection, and object clustering.

Based on the initial implementation of MetaStore, we show performance measurements and propose improvements to the metaobject protocol of CLOS. CDRS CAD system, an object-intensive application at Evans & Sutherland Computer Corp., will soon be ported to MetaStore.

To Joyce, Chris, and Jamie.

Contents

Figures

Acknowledgments

The long and winding road ...
The Beatles

Indeed, it has been a long and winding road! I am just happy that so many people have endured me long enough that I get to write the acknowledgements section of a doctoral dissertation.

First, I thank my research advisor, Joe Zachary. This thesis would not have come about without his patience, technical insight, and numerous yet thorough readings. His concern and encouragement kept me on track during the times when I was wandering around.

I also thank the other members of my thesis committee, Bob Kessler and Gary Lindstrom, for their focused questions and discussions above and beyond the call of duty. I will remember that one AMPS seminar that helped me realize there was indeed a bigger problem with *shared structures* than I first thought. This triggered a series of long discussions with them for months to come. Mark Swanson also gave much of his time on those discussions. I wonder if they would still be interested in "sharing" *anything.*

Evans & Sutherland gave me the opportunity to be persistent on persistence. Special appreciations go to Don Garner and Thomas Jensen for their support and, of course, to the CDRS *"peons"* for their friendship and technical discussions. Craig Steury deserves a special mention for those countless long "persistent" afternoons. Randall Krebs' installing LaTeX at E&S made my life much less miserable when it counted the most.

I owe special thanks to Andreas Paepcke for his generous time over the phone on many occasions to discuss his implementation of persistence and for sharing with me some of his sample code and papers. I also owe special thanks to Gregor Kiczales for the encouragement on using metaprogramming in implementing persistence, for giving "phone support," and for sending me a copy of the book *The Art of the Metaobject Protocol* even before it was published.

My fellow graduate students not only have helped me with technical discussions but more importantly have been friends. Thanks therefore go to Venkatesh Akella, Gilad Bracha, and Rok Sosič. Keep in touch!

I also thank the staff, past and present, at the CS department front office for all their help, especially Susan Jensen, Colleen Hoopes, and Stephanie Miya.

I am grateful for the information exchanges, verbal and electronic, with many good people over the years. The following deserve a special mention: Don Batory, Edwin Blake, Rick Cattell, Ken Kahn, Peter Lyngbaek (I am sad he is no longer with us), David Maier, Eliot Moss, John Rosenberg, Michael Stonebraker, Jon L. White, Peter Wisskirchen, and Jia-Huai You.

Let me thank my wife, Joyce, for her love, patience, nurture, and faith in me for *so long,* and my children, Chris and Jamie, for their love and for not complaining too much about the weekends without their dad. I will make it up guys—I promise! Finally, I thank my parents and grandpa on the other side of the ocean for their love and quiet yet enduring support. Grandma, I wish you were here to see this.

1

Introduction

Separate the transient from the permanent!
Brad Cox

This dissertation addresses the problem of object persistence in object-oriented programming languages. *MetaStore,* a persistent object system, uses the metaprogramming facilities of the metaobject protocol [23] to add persistence to the Common Lisp Object System [8,21]. This approach leaves the semantics of CLOS unchanged, requires only minimal syntactic changes to existing programs, and needs no compiler or run-time support. In the resulting language, programmers can selectively define both objects and slots to be persistent. MetaStore then handles persistence at the metaobject level.

Object-oriented programming is a convenient mechanism for organizing complex data and algorithms through encapsulation and inheritance. Applications such as computer-aided design, software engineering, scientific databases, medical and cartography systems, knowledge-based systems, multimedia systems, and office information systems require many complex objects, and thus are beginning to adopt object-oriented approaches.

There are two major issues to be solved in these so-called *object-intensive* applications. First, objects must persist so that they can be used again later. File-based or traditional database approaches are ill-suited for the many complex objects found in these applications. Second, many of these applications also use vast amounts of

data, often exceeding a machine's virtual memory. Thus, only a subset of all the objects may be kept in virtual memory.

Object-oriented programming languages can cleanly and elegantly be extended to support persistence at the granularity of objects and slots in such a way that not all objects and slots must be loaded into virtual memory or saved to disk together in batch mode.

MetaStore is a portable persistent object system designed to solve these problems. It supports the persistence of passive data by providing a virtual object memory. It tries to keep an appropriate number of objects in memory to maintain system performance at an acceptable level, and allows programmers to tune performance by specifying object and slot clustering. MetaStore currently supports only limited aspects of class evolution.

MetaStore addresses a range of issues in implementing persistence, including object identity, addressing mechanisms, shared objects and structures, dirty bits, queries, transaction management, version control, object base garbage collection, and object clustering.

MetaStore has two major components: (i) the language extension portion that relies on the *meta*object protocol of CLOS, and (ii) the database management portion that relies on a persistent object *store,* thus the name *MetaStore.* The focus of our research is on the *Meta* portion, although some level of database management support is required for persistence and is provided by *Store.*

Based on the initial implementation of MetaStore, we propose improvements to the metaobject protocol of CLOS in both design and implementation. We also propose changes in the design of MetaStore because performance measurements suggest that the current design of MetaStore overburdens object creation and accesses which are two of the most critical aspects of an object-oriented system.

With the proposed changes, the CDRS CAD system, an object-intensive application at Evans & Sutherland Computer Corp., will soon be ported to MetaStore.

In the rest of the chapter, section 1.1 describes the problem of object persistence based on our experience with a real world, object-intensive application. Section 1.2 surveys related work. Section 1.3 provides an overview of MetaStore. Finally, section 1.4 sketches a roadmap for the rest of the dissertation.

1.1 Object Persistence

There is no permanence.
The Epic of Gilgamesh

An object is *persistent* if it survives the termination of the process that created it. Thus, the object must be saved in secondary storage when the process terminates so that it can eventually be loaded and reused by another process at some later time. Unless stated otherwise, an *object* is an instance of a class throughout this dissertation. In MetaStore, persistence is supported on passive data with only a limited treatment of class evolution.

1.1.1 CDRS: An Object-intensive Application

Conceptual Design and Rendering System [26] is a geometric CAD modeler that is being used by designers in a dozen major automotive companies worldwide. CDRS is large[1] and sophisticated, and requires many complex geometric objects. It is written mostly in Common Lisp [46] with its object-oriented extension, CLOS [8,21]. The majority of data structures found in CDRS, especially the geometric data that must be persistent, are objects. An *object-intensive* application like CDRS is characterized as requiring:

- A large number of objects that may not all fit in virtual memory

[1]It is about 400,000 lines of Common Lisp code and 150,000 lines of C [22] and C++ [27,48] code. The disk image for CDRS is about 40 megabytes and the virtual image about 100 megabytes.

- A wide variation in the sizes of objects
- Complex data structures within objects
- Rich relationships, both semantically and structurally, among objects.

Supporting object persistence in object-intensive applications is particularly hard because of these characteristics. The problems we have had in CDRS with the file-based, batch-oriented approach of object persistence motivated this research. A persistent object system is essential for this kind of applications, and MetaStore is designed to meet this need. The design of MetaStore is general enough to support all object-intensive applications, although many design decisions are influenced by our experience with CDRS. Generality, practicality, elegance, and performance were all critical factors in the design decisions. We describe the problem of object persistence in detail and sketch our solution in the next section.

1.1.2 Understanding The Problem

The file-based, batch-oriented approach for object persistence is ill-suited for the many complex objects found in object-intensive applications. Our experience with CDRS shows that this approach is not acceptable since it is very inefficient in both time and space. With this approach, all the objects in a design session are saved in a model file in batch mode at the end of a modeling session. When a new session starts, all the objects in a model file are again loaded into virtual memory before any object is modified. This approach requires a huge amount of virtual memory, frequent and long garbage collections, and a long time to load and save the models. For example, CDRS usually uses 500 megabytes of swap space and 64 megabytes of main memory (often expanding to 128 megabytes). An automotive

model built by one of CDRS' corporate users takes about 24 minutes[2] to save or load. Users claim that anything more than three minutes of waiting is too much in a production environment. Users are usually noncomputer professionals and tend to save models frequently in fear of losing models due to reliability problems. Improving software reliability[3] and reducing user waiting time are critical for the success of an object-intensive system like CDRS. Our approach will reduce user waiting time by exploiting incremental saves and loads.

Traditional database approaches for object persistence are also ill-suited for the many complex objects found in object-intensive applications. Relational systems have successfully been used in many data-intensive [56] applications. They are well suited for data of a uniform or simple record format, but have difficulty handling objects of arbitrary sizes and complexities. They also suffer from what is known as *impedance mismatch* [4,15] between the data manipulation language (DML) and the host programming language of a database management system (DBMS). DMLs and programming languages do not match well: the impedance mismatch results mainly from the type mismatch between these two different kinds of languages. Because the type system of a host language and that of a DML for a database system are usually different, a programmer must take care of the type conversion when storing data in the database and when retrieving them from the database to operate on.

In contrast, a persistent object system supports object persistence using only one language, an object-oriented language. Instead of interfacing with a DBMS, a persistent object system usually interfaces with a persistent object store, which supports a basic subset of the DBMS features. Thus, persistent object systems

[2]It now takes about eight minutes after some optimizations are done with a binary file format. However, I am using the number before the optimizations were done since other numbers we will see with the current implementation of MetaStore are not based on the similar optimizations.

[3]Each new release is usually accompanied by one of the developers for a month or so.

can deal with complex objects without suffering from impedance mismatch and without having to support sophisticated database system features. A persistent object system is essential for a system like CDRS, and MetaStore is designed to support such applications.

Designing a persistent object system requires addressing a range of issues, the major ones being object identity, addressing mechanisms, shared objects and structures, dirty bits, queries, transaction management, version control, object base garbage collection, and object clustering. MetaStore focuses on these issues although other issues have to be handled at some level of sophistication since it is meant to support a production system, CDRS.

1.2 Related Work

How many good ideas can there really be?
Luca Cardelli

Saving the results of computation for a later use has been an issue from the very beginning of programming history. A comprehensive survey on this subject would be so vast that this survey focuses only on the ones that have led to the idea of MetaStore.

1.2.1 Interlisp

Interlisp [49] uses the simplest method of providing a language with a persistent store by just copying the whole heap out to a file. The system then resumes by loading the file. Simplicity is a virtue, but it carries a price. With this technique for instance:

- The heap must always be loaded onto the same location, or else pointers held on the heap become invalid. If the heap starts above the program, then any editing of the program which changes its size will invalidate all of the pointers.

In Lisp this is not a problem as program and data both sit in the heap, but this means giving up the freedom provided by ordinary filing systems of running one program against several files.

- It is impossible to store more data than can be loaded onto the heap at any one time. The maximum database is limited by the machine's physical or virtual memory.

- Programs which make only a few small changes to a large body of stored data pay a high I/O penalty because they must load the entire heap at the start and save it again at the end of a session.

1.2.2 PS-Algol

PS-Algol [13] is S-Algol [36] extended with persistence by applying the principle of *orthogonal* persistence. That is, any data type that can be handled internally within the language can be made persistent so that it can be stored and transmitted to other programs. It was done by an improvement in language implementation techniques rather than by linguistic changes. Thus, the implementation depends heavily on the language's run-time support.

The persistent heap of PS-Algol is implemented by the Persistent Object Management System (POMS) [11], which supports incremental loading and saving. The run-time system maintains two separate heaps, one in RAM and the other on disk. Thus, there are two types of addresses. Persistent IDs (PIDs) are represented as negative integers and local addresses as positive integers. A dereference instruction can therefore trap all negative addresses. The POMS then fetches the object from disk, loads it onto the heap and overwrites the negative address that had been dereferenced with the new local address. When a program has finished, the run-time system copies objects from the local heap in RAM to the persistent heap in disk.

As each object is copied from RAM to disk, all local addresses are converted to PIDs.

POMS uses the concept of *root* object. When a program is run against the database, a distinguished object is copied from the database to the local heap. This root object contains pointers to other items in the database.

With this approach, persistence is supported in the base language implementation level, where the persistence implementor can have flexible hooks to the internals of the base language implementation. The implementation in this approach wouldn't be portable though. Persistence can also be supported at the application level or at the metalevel. These different approaches which include the one MetaStore uses are described in section 2.4.

1.2.3 Mneme

The *Store* portion of MetaStore is most closely related to Mneme[4] [37]. Mneme is a persistent object store that can be used to support distributed persistent programming language implementations. It gives the illusion of a large, shared heap of objects, directly accessible from the programming language used to build the applications.

Mneme stores objects in a simple and general format, preserving object identity and object interrelationships. Given an object ID, each access to an object requires looking up the ID in the resident object table.

Mneme supports the minimal object semantics—every object has an ID, the object can contain IDs to describe its pointer relationships to other objects, and the object can contain any non-ID data desired.

---

[4]Mneme is the Greek word for memory; it is pronounced Nee-mee.

Mneme currently is being used to implement Persistent Smalltalk and Persistent Modula-3, which pose different problems. They are at opposite ends of the interpreted versus compiled and run-time versus compile-time type checking spectra.

Although *Store* is a necessary part of MetaStore, it was not the main focus of our research. If Mneme were available in Common Lisp, we could have used it in MetaStore by introducing the model file concept, which models a real world user "object," as an abstraction on top of Mneme.

1.2.4 PCLOS

Persistent Common Lisp Object System [38,39,40] is the closest relative of the *Meta* portion of MetaStore. PCLOS uses the metaobject protocol [23] of CLOS to add persistence to CLOS. It interfaces with a database management system through the virtual database layer for the database support.

PCLOS uses *husk* objects, in-memory data structures representing the objects in the database, for the purpose of message passing and in-memory referencing. The smallest grain size of persistence in PCLOS is an object, whereas in MetaStore it is a composite slot[5].

PCLOS has a very general system model with the following three layers:

- Language objects implementation
- Virtual database—conceptual interface between the other two
- Database management system

The general problem of impedance mismatch is bad enough with DBMS approaches. Implementing the system model of PCLOS requires dealing with one more mapping, thus requiring two overall:

[5]The notion of composite slot is introduced in section 3.5.

1. Objects and classes in the language must be described in terms of operations and concepts of the virtual database.

2. The virtual database operations and concepts must then be expressed in terms of various specific database managers.

In MetaStore we use a persistent object store, thus eliminating impedance mismatch altogether.

1.3 MetaStore

MetaStore is a persistent object system with a virtual object memory. This section gives an overview of MetaStore by describing our research goals, language design aspects via metaprogramming, the persistent object store used in MetaStore, and the architecture of MetaStore.

1.3.1 Research Goals

The main research goals of MetaStore are:

- Support object persistence so that it is transparent to application programs while maintaining system performance at an acceptable level.

- Design a general purpose persistent object store to be used by object-oriented programming languages that support object persistence.

- Design a portable persistent object system by extending an object-oriented language. This is done by using the metaprogramming facilities of a language system through which the persistent object store is used. Objects in this system will be incrementally loaded and saved. This system will be portable since it will not rely on any compiler or run-time support.

- Preserve the semantics and minimize syntactic changes of the language chosen while supporting object persistence.

- Support a virtual object memory to monitor the amount of data in virtual memory and reduce it by flushing out some objects if there are too many objects.

1.3.2 Language Extension via *Meta*programming

This research was motivated by the problem of object persistence in CDRS. In solving this problem by supporting object persistence in a language system, the metaprogramming facilities of CLOS are used because:

- CDRS uses a commercial CLOS implementation with no access to the implementation source code.

- CLOS supports metaobject protocols.

- CLOS can be extended via its metaobject protocols to support object persistence.

- We want to evaluate the merits and drawbacks of the metaprogramming ideas in programming language design.

Thus, the metaobject protocol of CLOS handles all the persistence related issues that interact with the run-time system side of CLOS.

The smallest granularity of persistence in PCLOS [40] is an object, whereas in MetaStore it is a composite slot. Based on our experience with CDRS, on the average about 80% of an object is occupied by composite slot values, and it seems important to control persistence at this grain size. This grain size is particularly important for the virtual object memory idea to work. Supporting the grain size at the slot level is made possible by the use of the metaobject protocol.

1.3.3 Persistent Object *Store*

Supporting persistence requires some degree of database management support. A general purpose, language independent, persistent object store is designed and implemented to provide the necessary subset of database management features instead of using a DBMS for the following reasons:

- Unless an object-oriented DBMS is available, impedance mismatch would be a major problem.
- Even if it is available, we have to totally depend on the features and efficiency of the DBMS.
- Many object-intensive applications, e.g., CDRS, do not need all the sophisticated features of a DBMS.
- Being able to tune the module/system that provides the database management support is extremely important for an application like CDRS. Thus, it is essential to have our own implementation. There is no Common Lisp based object-oriented DBMS that can support CDRS with the required efficiency.

The persistent object store is designed to support object-intensive applications well. Moss [37], for example, states that:

- One of its goals was to be able to retrieve 10,000 "useful," "typical size" objects per second from external storage into memory.
- "Typical size" for languages such as CLU [28,29], Smalltalk-80 [16], and Trellis [42] appears to be in the range of 30 to 50 bytes.

Based on our experience with CDRS, however, the objects range anywhere from a few hundred bytes to 150,000 bytes. The decision to support the smallest persistence granularity at the slot level and to support a virtual object memory was

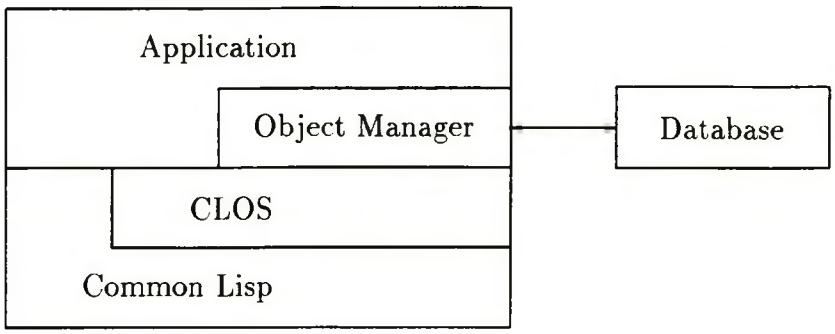


Figure 1.1. Object manager as an application module

influenced heavily by this characteristic of CDRS. We believe that most "design" applications share this characteristic.

1.3.4 MetaStore Architecture

If persistence is supported as a module in the application layer of a persistent object system, the system architecture would look like Figure 1.1. With this architecture the object manager handles the object management as well as persistence of the objects. Pre-MetaStore CDRS uses this architecture.

With persistence added to the metalevel of CLOS, the overall architecture of MetaStore looks like Figure 1.2. In Figure 1.2, `Store` is the persistent object *store,* and `Meta` is the language extension portion done via the *meta*programming facilities.

1.4 Roadmap

The remainder of this dissertation consists of five chapters.

In Chapter 2, several preliminary subjects that would help the readers understand the rest of the dissertation are discussed. They are the fundamental concepts of the object-oriented programming paradigm, a persistent object model, use of

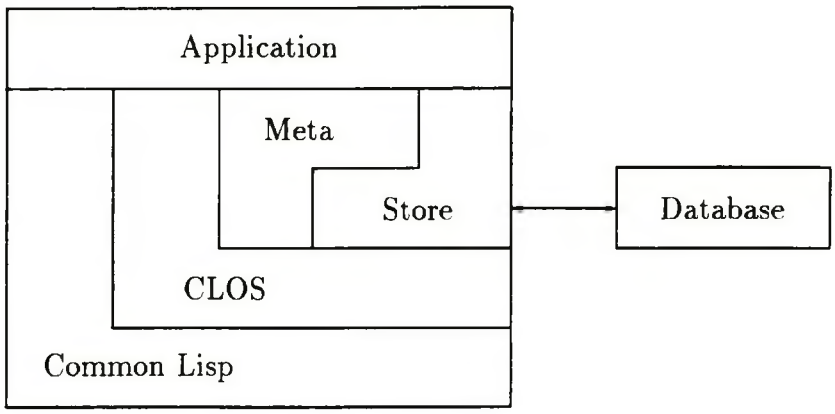


Figure 1.2. MetaStore architecture

the metaprogramming ideas in programming language design, and several different schemes of adding object persistence to a language system.

Chapter 3 describes the *Meta* part of MetaStore. That is, it describes how the metaobject protocol of CLOS is used to extend an object-oriented language, CLOS, to support object persistence. After an overview of *Meta* by describing the life cycle of a persistent object, various major issues are described in turn.

Chapter 4 describes the *Store* part of MetaStore. It describes a general purpose, language independent persistent object store that provides a necessary subset of DBMS features for object-oriented languages that support object persistence.

Chapter 5 reports the results of an implementation. They include our experience of using the metaobject protocol of CLOS as a language design tool, and various performance measurements on *Meta* and *Store* portions of MetaStore along with analyses of the measurements.

Finally, Chapter 6 sums up the dissertation by summarizing our contributions and sketching future work.

2

Preliminaries

Now, these are the foundations
II Chronicles 3: 3

In this chapter, several preliminary subjects that will help the reader understand the rest of the dissertation are discussed. First, the fundamental concepts of the object-oriented programming paradigm are summarized. The main purpose of this summary is to illustrate how these fundamental concepts help in cleanly and elegantly extending an object-oriented language with persistence. Second, an object-oriented database system model [3] is referenced, and we discuss how close MetaStore is to this model. Third, metaprogramming as a programming language design tool is discussed. Finally, three different possible persistence schemes are described.

2.1 Object-Oriented Programming

My cat is object-oriented
Roger King

Object-oriented programming is a programming methodology that encourages modular design and software reuse. There are many differences among existing object-oriented programming languages in terms of what is supported in the language. Describing the differences is beyond the scope of this section. Therefore, we describe the concepts that are considered fundamental in an object-oriented language. They are encapsulation, dynamic binding, and inheritance. While

concentrating on these fundamental concepts, we also describe why some object-oriented languages make extension for persistence more elegant and natural than other languages allow.

2.1.1 Encapsulation

Object-oriented programming languages support *data abstraction* by preventing an object from being manipulated except via its defined external operations. A class or more generally an abstract data type (ADT) has two components: specification and implementation. The implementation of a class includes the details of how it represents the data it contains, and which algorithms it uses for the actions it supports. Client programs then interact with class instances only by means of a well defined set of operations, i.e., the class' specification. The idea is to hide the implementation details of a class from the client programs. In object-oriented programming languages such as C++ [27,48] and CLOS [8,21], the separation between specification and implementation is enforced by the language, thus improving the modularity and reusability of the software.

An object can be viewed as an encapsulation. That is, an object has its internal state, its implementation, and its public interface. How an object implements its actions, and how its internal data are arranged, are encapsulated inside a procedural shell that mediates all public accesses to the object. Typically, the internal state is represented by the values of the slots[1] making up the object, the actions are implemented by methods, and the public interface is a subset of generic functions defined for a given class.[2]

---

[1] We borrow the terminology from CLOS. They include *slots, methods, generic functions,* etc.

[2] There are, in general, many generic functions defined for a class: some for the internal use and others for the public interface.

When an object-oriented language is defined and implemented via the use of metaobjects as is the case with CLOS [23], metalevel encapsulation is also possible in the form of metaobjects. Given a class, for example, all the instances, methods, generic functions, and the class definition itself of the class can be encapsulated in the metaobject of the class.

Given a class, because all the class related information that needs to be persistent is encapsulated in one place, namely in the metaobject of the class, it is easier to support object persistence in object-oriented languages than it is with non-object-oriented languages. The metaobject of a class is the only place to look to find all the information that needs to be saved in object-oriented languages that support metaobject protocols, whereas application programs must maintain all the information that needs to be persistent with a non-object-oriented language. There are other advantages. For example, there might be classes that are meant to be persistent while others are not. This can easily be supported in an object-oriented language like CLOS via the use of metaobjects as we will see in section 2.1.3. Controlling the grain size of persistence at the slot level can also easily be done with object-oriented languages that support metaobject protocols.

2.1.2 Dynamic Binding

Object-oriented languages support dynamic, or late, binding of method names to their implementations at run-time.[3]

Suppose we have a list of arbitrary objects: 2D points, 3D points, lines, circles, curves, raster images, students, etc. Given a list of these heterogeneous objects, suppose we want to display each object in the list.

---

[3]In CLOS, generic functions are bound to implementations of methods.

In a non-object-oriented language like Pascal [18], one routine, say `display`, would handle all the objects in the list.[4] `display` would be implemented using a `case` statement on the type of objects. Depending on the type of an object, it would execute different branches of the case statement. This one routine, thus, performs two functions: dispatching on types and displaying objects. Given this implementation of `display` and a list of objects, each object can easily be displayed.

Assuming that surfaces are not included in the initial implementation of `display`, suppose we now want to be able to display surfaces as well using the same mechanism. We would have to modify the existing `display` routine by adding another branch that handles surfaces. When the routine, `display`, is implemented by the user, this is manageable. However, if `display` was one of the routines that was provided by a module in a library to which we have no write access, then we would have to handle surfaces differently.[5]

In an object-oriented language like CLOS, each object type, i.e., class, would define its own method, `display`, to display that particular type of object only. Thus, the displaying part is handled by each class. The dispatching part, however, is handled by the language system itself. When an object is given as an argument to `display`, the language system at run-time finds the method that was specifically defined for the class the given object is an instance of, and executes the method with the given object as an argument. Therefore, the binding of the name `display` to an appropriate method implementation happens at run-time, thus the name

---

[4] Also in Pascal, the list has to be a homogeneous list of variant records.

[5] An alternative solution exists as well. We could implement one display routine for each type such as `display-2D-point`, `display-3d-point`, `display-line`, etc. This alternative puts the similar burden to the user of the routines. That is, given a list of objects to display, the user routine would have to check the type of each object in the list and call the right routine depending on the type. This seems even worse than the other solution because there would be in general many users of this kind of routines.

dynamic or *late* binding. With this extra support by the language system, all the difficulties that we mentioned above with other possible solutions disappear.

Along with inheritance, which we will describe in the next section, dynamic binding helps support object persistence as well. One of the routines we would need in supporting object persistence is `save-object`. Without the help of inheritance, each class would define a method with this name. Then, the module handling persistence would use these methods. However, this is not the best we could do with object-oriented languages because inheritance makes it even simpler, which we will now see in the next section.

2.1.3 Inheritance

Inheritance allows us to reuse and share the behavior of a class in the definition of new classes. Subclasses of a class inherit the operations and slots of their superclass and may add new operations and new slots.

For the purpose of supporting persistence, let us suppose that we defined a class named `persistent-class`. There would then be a method named `save-object` among others defined for this class. Each user class that needs to be persistent, for example `2D-point`, can now inherit `persistent-class` so that its instances can enjoy the persistent behavior with the help of `persistent-class`. We call a user class that inherits `persistent-class` a *persistable class.* An instance of a persistable class is a *persistable object,* which becomes *persistent* when saved. A *persistable slot* is a slot declared persistable in a persistable class. In the absence of its own method named `save-object`, if we send the message `save-object` to one of the instances of the `2D-point` class, the one defined for `persistent-class` is executed because the superclass `persistent-class` is inherited by a subclass `2D-point`.

Without inheritance in the previous section, we said that we would have to define a method named `save-object` for each class whose instances need to be persistent. With inheritance, all each user class has to do is to inherit `persistent-class` unless the user class has a reason to modify the behavior of the default method defined for `save-object` by `persistent-class`. With the help of the metaobject protocol in CLOS, the method `save-object` or any other persistence handling method in general does not need to understand the semantics of the class and its contents being saved. Thus, supporting persistence via the use of inheritance is quite adequate. MetaStore supports object persistence via inheritance as we will see in Chapter 3.

This section presents only a flavor of what inheritance is and how it helps in supporting object persistence. There is much more to inheritance than what is described in this section. Rather than describing it in detail, which is beyond the scope of this section, we refer the readers to [21], [27], [9], and [14] for more detailed coverage of inheritance. Jigsaw [9] and Cook [14] are particularly good for the formal semantics of inheritance.

2.2 Persistent Object Model

Indeed, there are at least 64 different object models!
Carl Hewitt

There are many object-oriented programming languages in use today. Different object-oriented languages support different sets of language features and there seems to be no consensus as to what must be included in a language for it to be considered object-oriented. The situation with persistent object systems is similar. For an object-oriented programming system to be considered a persistent object system, it must support some database system related features, and there seems to be no consensus here, either.

The authors in [3] define a model for *object-oriented database systems.* It describes the main features and characteristics that a system must have to qualify as an object-oriented database system. These characteristics are separated into three categories: mandatory, optional, and open.

Although persistent object systems have slightly different requirements than object-oriented database systems, they are in general quite similar. Persistent object systems, not being a database management system, require fewer of the features that are related to the database management aspects. MetaStore is designed with the model defined in [3] as a guide. In this section, the aspects in which MetaStore deviates substantially from the model defined in [3] are described. Only mandatory and optional features are considered since the open features are considered beyond the scope of the initial design of MetaStore.

2.2.1 Mandatory Features

As mandatory features for object-oriented database systems, the authors in [3] list complex objects, object identity, encapsulation, types or classes, inheritance, overriding combined with late binding, extensibility, and computational completeness on the language side; persistence, secondary storage management, concurrency, crash recovery, and ad hoc query facility on the database management side.

The things listed on the language side are already supported by most object-oriented languages in which MetaStore would be implemented, including CLOS in which the initial implementation of MetaStore is done.

Among the things listed on the database management side, secondary storage management is a classical feature of database management systems. It is usually supported through a coordinated set of mechanisms. For object-oriented database systems, these include index management, data clustering, data buffering, access path selection, and query optimization. In MetaStore, only a simple form of index

management, data clustering, and access path selection are supported; data buffering and query optimization are not treated. Supporting data buffering is beyond the scope of this dissertation; the one provided by the operating system would be good enough for our purposes in MetaStore. Supporting query optimization is not necessary because all the queries we support in MetaStore are simple, e.g., query by object identifier, query by name, and query by key.

2.2.2 Optional Features

Some features that improve the system, but that are not mandatory to make it an object-oriented database system are classified as optional. Some of these are of an object-oriented language nature and others are simply database system related. Most of these are targeted at serving "new" kinds of applications such as CAD and office automation, and are more application oriented than technology oriented.

The authors in [3] list multiple inheritance, type checking, and type inferencing on the language side; distribution, design transaction, and versions on the database management side.

Multiple inheritance is supported by CLOS. MetaStore supports transactions in design environments and a simple form of version control. Distribution, type checking, and type inferencing are not included in the design of MetaStore because they are beyond the scope of this dissertation.

2.3 Metaprogramming

Reflection's broader implications:
deepest hopes and darkest fears.
Gregor Kiczales

This section briefly describes what metaprogramming is, gives an example of a language system that supports it, and explains how one, the metaobject protocol of CLOS, is used in MetaStore.

Some computational systems are designed to have two levels: the base level (computational level) and the metalevel. Programming at the metalevel is called metaprogramming. *Reflection* is the fundamental concept that makes the metaprogramming ideas possible, and is defined to be:

> an entity's integral ability to represent, operate on, and otherwise deal with itself in the same way that it represents, operates on, and deals with its primary subject matter [2].

A detailed description of reflection and reflective techniques is beyond the scope of this section. Thus, the reader is directed to [31] for a comprehensive treatment of reflective techniques, and to [45] for an elaborate description of how reflection is used in describing the semantics of a dialect of Lisp called 3-Lisp [45]. The workshop report [2] describes the recent research activities on the subject.

One popular design of an object-oriented programming system that supports metaprogramming is CLOS.[6] Its metaprogramming facilities, the metaobject protocol, is of immediate interest to MetaStore. The metaobject protocol of CLOS is used in MetaStore to extend CLOS with object persistence.

The authors in [23] describe how CLOS was designed and implemented with reflective and object-oriented techniques as the underlying technologies to support metaprogramming. It is stated in [23][pg. 1] that:

> This book is about a new approach to programming language design, in which these two demands of elegance and efficiency are viewed as compatible, not conflicting. Our goal is the development of languages that are as clean as the purest theoretical designs, but that make no compromises on performance or control over implementation. ... The way in which we achieved elegance and efficiency jointly is to base language design on *metaobject protocols*. Metaobject protocols are interfaces to the language that give users the ability to incrementally modify the language's behavior and implementation, as well as the ability to write programs within the language. ... The metaobject protocol approach, in contrast, is based on the idea that one can and should "open languages up," allowing users to adjust the design and implementation to suit their needs.

[6] Many industrial strength implementations of this design are also available.

In CLOS, the language implementation itself is structured as an object-oriented program, which includes an implementation of reflection. This allows users to exploit the power of object-oriented programming techniques to make the language implementation adjustable and flexible, using standard techniques of subclassing and specialization in object-oriented programming.

In the design of CLOS, the basic elements of the programming language—classes, methods, and generic functions—are first made accessible as objects. Because these objects represent fragments of a program, they are given the special name *metaobject.* Individual decisions about the behavior of the language are encoded in a protocol operating on these metaobjects—a *metaobject protocol.* For each kind of metaobject, a default class is created, which delineates the behavior of the default language in the form of methods in the protocol.

In the CLOS metaobject protocol, the rules used to determine the implementation of instances are controlled by a small number of generic functions. This makes it possible to change those rules by defining a new kind of class, as a subclass of the default class, and by giving it specialized methods on those generic functions. By doing this, the user is making an incremental adjustment in the language. Most aspects of both the language's behavior and implementation remain unchanged, with just the instance representation strategy being adjusted.

Using the metaobject protocol, MetaStore incrementally adds object persistence to CLOS. There are structural adjustments and behavioral adjustments. In structure, each persistable class is augmented with extra slots that MetaStore uses. This is done by using the inheritance mechanism at the metalevel. Behavioral adjustments are made by specializing generic functions defined in the metaobject protocol. As long as the protocol is complete enough to do the things that need be done to add persistence, MetaStore will not have to know anything about the internals of how CLOS is implemented. When a deficiency is found in the

protocol, it can be extended to handle the deficiency. In Chapter 6 we propose some improvements to the metaobject protocol of CLOS based on our experience with MetaStore.

2.4 Persistence Schemes

I offer thee three things: choose one of them
I Chronicles 21: 10

In this section, we discuss at what level in an object-oriented language system object persistence can be best supported. It can conceptually be supported at one of several different levels of such a language system:

- *Base level persistence:* Persistence is supported at the base language implementation level with a full compiler and run-time support of a language system. This is the same level at which, for example, garbage collection is supported. Persistence in PS-Algol [13] is implemented at this level.

- *Application level persistence:* Persistence is supported at the application programming level as an application module, say an object manager, designed to support the persistence of user objects. Pre-MetaStore CDRS is implemented in this way.

- *Metalevel persistence:* Persistence is supported at the metalevel via the metaobject protocol of a language system. At this level, the persistence implementor can support object persistence without any help from either user programs or the language compiler and run-time support system. This is possible because the metalevel interface provides enough hooks to the underlying language system internals for user programs. Some limitations of this scheme, encountered in implementing MetaStore is detailed in section 5.1.

Let us examine different issues involved in supporting persistence in each of these approaches.

2.4.1 Addressing Mechanisms

The question of addressing mechanisms is at the basis of all implementations of persistent object systems. In persistent object systems we store objects while preserving referential structure. Different addressing mechanisms are available depending on which scheme we use.

In the base level persistence scheme, we can adopt an addressing mechanism particularly suited for persistent data with full freedom. In PS-Algol [13], for example, disk addresses were represented as negative numbers and virtual addresses as positive numbers by dividing address space in two halves. Segmented virtual memory used by the Smalltalk system [19] is another example of addressing done at this level.

Both metalevel and application level schemes are similar in the degree of access they have, which is essentially none, to the base language implementation as far as addressing mechanism is concerned. This is because addressing mechanism is not a property of object-orientedness but of the base language implementation techniques. Therefore, we have to choose mechanisms appropriate for these levels. Associative persistent ID addressing and pointer swizzling are two mechanisms that are available at these levels, and they are described in detail in section 3.7. Of course, these mechanisms can also be used at the base level.

2.4.2 Incremental Saves

Supporting incremental saves of objects requires keeping track of which objects, structured data, and atomic data have been modified since the last incremental save. By adding a dirty bit to each data item, the persistence implementor can intercept all the data mutating operators in the language to update the dirty bit. Because the values of slots in an object can be changed without going through the

public interface of an object, the problem of keeping track of dirty bits is hard, especially with CLOS because of the flexibility available to user programs.

In the base level persistence scheme, it is easy to maintain the dirty bits because the persistence implementor has easy access to the internals of the language implementation. The persistence implementor also has unlimited access to all the data mutating operators such as `(setf aref)` for arrays and `rplaca` for lists in Common Lisp, for example.

In the metalevel persistence scheme, it is easy to add dirty bits to objects and intercept the object mutating operators, e.g., `(setf slot-value)`, but there is no way to do this for data that are not objects. For instance, adding a dirty bit to an array and intercepting the array mutating operator, e.g., `(setf aref)`, is impossible to do using only the interface provided by the metaobject protocol. The persistence implementor must resort to a mechanism that an application level persistence implementor can use, as described next.

In the application level persistence scheme, the persistence implementor has no access whatsoever to the internals of the language implementation. The implementor has to maintain the dirty bits in his or her own data structures although they can be maintained in slots in the case of objects, and the implementor must rely on user programs telling him or her what is being modified.

Therefore, the base level persistence scheme would be the best one to use for dealing with incremental saves of modified objects. If metaobject protocols are available for all persistable data types supported in a language, then the metalevel scheme would be as good as the base level scheme.

2.4.3 Shared Structures

Preserving language semantics on structure sharing in the context of persistence is a difficult problem. By structure, we mean structured data other than objects.

Thus, arrays and lists are examples of structures. Suppose the array a1, a slot of the object O1 in Figure 2.1 is ready to be saved, and also suppose that a1 has another array a2 as one of its elements. Further suppose that a slot of another object O2 also has a2 as its value through a third array a3. Thus, a2 is shared by O1 and O2.

This sort of sharing is perfectly legal in Common Lisp. Assuming only objects have dirty bits,[7] and also assuming both O1 and O2 are dirty, if both O1 and O2 are saved, two copies of a2 will be saved: once by O1 and another by O2. When O1 and O2 are both loaded at some later time, b of O1 and c of O2 will have their own copies of the original array a2, say a2-1 and a2-2.

In the base level persistence scheme, the persistence implementor has several options. One would be keeping back pointers from an array to each of its references.

[7]Unless we have access to the base language implementation level, there is no easy way to add a dirty bit to a data item unless it is an object.

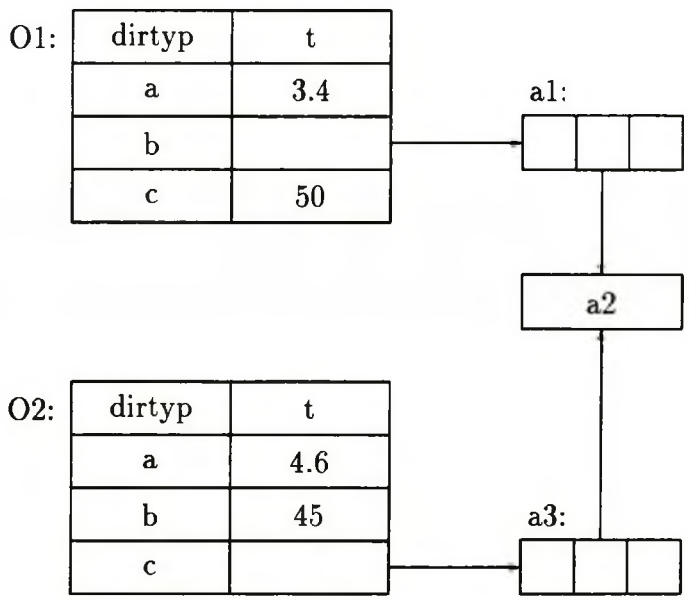


Figure 2.1. An array shared by two objects

Another would be keeping one level of indirection on each array. Still another would be representing each array as an object. Whichever the implementor chooses, he or she has the full freedom as to how arrays are represented and how they are accessed in the context of persistence.

Both the metalevel and application level schemes have no access to the internals of the nonobject, structured data of a language system. Therefore, they must rely on their own mechanism to deal with sharing of these structures. One possible solution at these levels is to keep back pointers in the form of a hash table with the virtual address of a referenced structure as the key and the things that make references to the structure as the value of each hash entry. This algorithm is described in detail in section 3.11.2.

Therefore, structure sharing in the context of persistence can best be supported in the base level persistence scheme. If metaobject protocols are available for all persistable data types supported in a language, then the metalevel scheme would be as good as the base level scheme.

2.4.4 Orthogonal Persistence

Orthogonal persistence requires supporting persistence of any data type that is supported in a language. Based on the analyses done in the previous subsections dealing with incremental saves and shared structures, therefore, it is obvious that the base level persistence scheme would be the best one to use, and it would be difficult to support orthogonal persistence in the other two schemes. If metaobject protocols are available for all persistable data types supported in a language, then the metalevel scheme would be as good as the base level scheme.

2.4.5 Selective Persistence

The issue here is separating the transient from the persistent. In a typical object-intensive application, the majority[8] of data items that are created, including objects, are meant to be transient, i.e., are never meant to be saved. This is so partly because some are meant to be transient for their entire life cycle, and partly because others, whether contained in an object or not, evolve many times before they finally reach a savable state. Because one of the main concerns of persistent languages is performance, treating all data being created as persistable would not be very desirable. Even for a persistable object, it is not desirable to save the object on every update because it would in general be updated many times before it reaches a savable state.

In the base level persistence scheme, given an array for example, it would be difficult to tell if it is meant by user programs to be transient or persistent. That is, the base level is too low a level to easily distinguish what is meant to be transient and what is meant to be persistent. If the language is modified in such a way that user programs are required to tell which arrays are transient and which arrays are persistent, for example, it would be too big a change in the language although it would solve the problem of selective persistence. An alternative at the base level would be to support two different kinds of arrays: transient arrays and persistent arrays, which again requires changes in the language syntax and semantics.

Separating the transient from the persistent at the object level seems appropriate. Then, any data reachable from a persistable object via persistable slots can be treated persistable.[9] Distinguishing persistable objects from transient objects can

---

[8]Our educated guess is over 90%. Generational garbage collectors, e.g., [53], are particularly effective because of this life pattern of program data.

[9]Therefore, a persistable object only reachable from a transient slot will in effect not be persistent. This situation can easily be fixed by declaring the slot to be persistable.

easily be done by using the inheritance mechanism of an object-oriented system. That is, a persistent root class can be defined to handle persistence, and any class that is intended to be persistable inherits the persistent root class. Therefore, both the metalevel and the application level would equally be better than the base level in supporting this feature.

2.4.6 Granularity of Persistence

In the base level persistence scheme, the grain size of persistence may be too small because each datum of data types supported by the language would be treated as a separate unit during a save or a load operation.

Because an object seems to be a good choice for the grain size, the application level persistence scheme would be a good choice. As with the selective persistence in section 2.4.5, the inheritance mechanism would be used.

If supporting the grain size at the slot level as well as the object level is desired (as it is in MetaStore), then the metalevel persistence scheme would be the best fit. The metaobject protocol of CLOS supports the interface to the slot level metaobjects, thus making it possible to support the grain size at that level.

2.4.7 Portability

Because the compiler and the run-time support system in the base level persistence scheme is likely to be customized to meet the needs of a persistence implementor, it would not be portable across different implementations of a language.

The metaobject protocol of an object-oriented language is designed to be portable, thus persistence implemented using the protocol would certainly be portable.

The persistence implemented via the application level persistence scheme would also be portable.

2.4.8 Summary

The metalevel persistence scheme is particularly good at supporting persistence of objects. It is quite natural to support selective persistence via inheritance mechanisms. It is easy to support slot level persistence by augmenting the slot accessing mechanisms. Because metaobject protocols are portable, so is the persistence implemented via them.

As for addressing mechanisms, a pointer swizzling mechanism, which is one of the most popular ways, can easily and naturally be used by the metalevel scheme.

Orthogonal persistence is hard to support without the support of the base language implementation level. If the base language itself is implemented with reflective and object-oriented techniques, thus supporting the metaobject protocols at that level, then orthogonal persistence would easily be supported at the metalevel. With this design, each data type would be implemented as an object and some selective aspects of the implementation internals would be accessible via the metaobject protocols. However, the current metaobject protocol of CLOS does not go this far: only objects are implemented this way, thus a limited access to the internals by user programs via the metaobject protocol. In Chapter 6 we propose some improvements to the metaobject protocol of CLOS based on our experience with MetaStore.

3 Language Extension via Metaprogramming

Language design is reminiscent of Ptolemaic astronomy
– for ever in need of further corrections.
Jean-Yves Girard

This chapter describes the *Meta* portion of MetaStore. That is, it describes how the metaprogramming facilities, the metaobject protocol, of an object-oriented language system, CLOS, is used to extend the language with object persistence. As we saw in section 2.4, object persistence can be implemented at one of several different levels of a language system. In MetaStore it is done at the metalevel.

Although the design of MetaStore can be implemented in any object-oriented language that supports metaprogramming facilities, CLOS was chosen in this research as an implementation language for the following reasons. CDRS, which motivated this research, is written in CLOS. CLOS supports a metaobject protocol, and can be extended with object persistence via the metaobject protocol.

After an overview of *Meta*, various major issues are described in turn.

3.1 Overview of *Meta*

The overview of *Meta* is given by describing the life cycle of a persistent object. At each stage in the life cycle, we describe what kinds of language extensions are necessary and how they are achieved via the help of the metaobject protocol of CLOS. When the protocol is not expressive enough to handle any issue, this is also described.

We distinguish between persistable and persistent as follows. A *persistable object* is an instance of a persistable class, where a *persistable class* is a subclass of the *persistent class* `persistent-root-class`. Thus, persistence in MetaStore is via inheritance. A persistable object becomes a *persistent object* when it is eventually saved to the object base. A *persistable slot* is a slot declared persistable in a persistable class. The value of a persistable slot also becomes persistent when saved.

Figure 3.1 contains two persistable objects, `O1` and `O2`. `O1` has five slots: `oid`, `a`, `b`, `c`, and `d`. `oid` is added by MetaStore and the other four are from the user defined class of which `O1` is an instance. `O2` has four slots: `oid`, `e`, `f`, and `g`. `oid` is again added by MetaStore and the other three are from `O2`'s user defined class.

A persistable object, just as any other object, is created by a method call such as `make-instance` in CLOS. In MetaStore, when a persistable object is created, it is assigned a unique object identifier (OID) (section 3.3). The OID of the object `O1` in Figure 3.1 is `22`. A unique OID is necessary to map virtual addresses to persistent IDs as objects are saved to the object base, and to map from persistent IDs to virtual addresses as objects are loaded from the object base (section 3.7).

Upon creation of a persistable object, an intermediary data structure called a *phole*[1] (section 3.5) is added between a persistable object and each of its persistable composite slot values (section 3.5). In Figure 3.1, slots `b` and `d` of the object `O1` and `f` and `g` of `O2` are composite slots, and each has its own phole. A composite slot is a slot whose value is not of a primitive type.

The use of pholes in MetaStore is a novel idea, which makes the following possible:

- Maintaining dirty bits for incremental saves

[1] Phole stands for a *p*ersistent *hole;* it is pronounced "Fole."

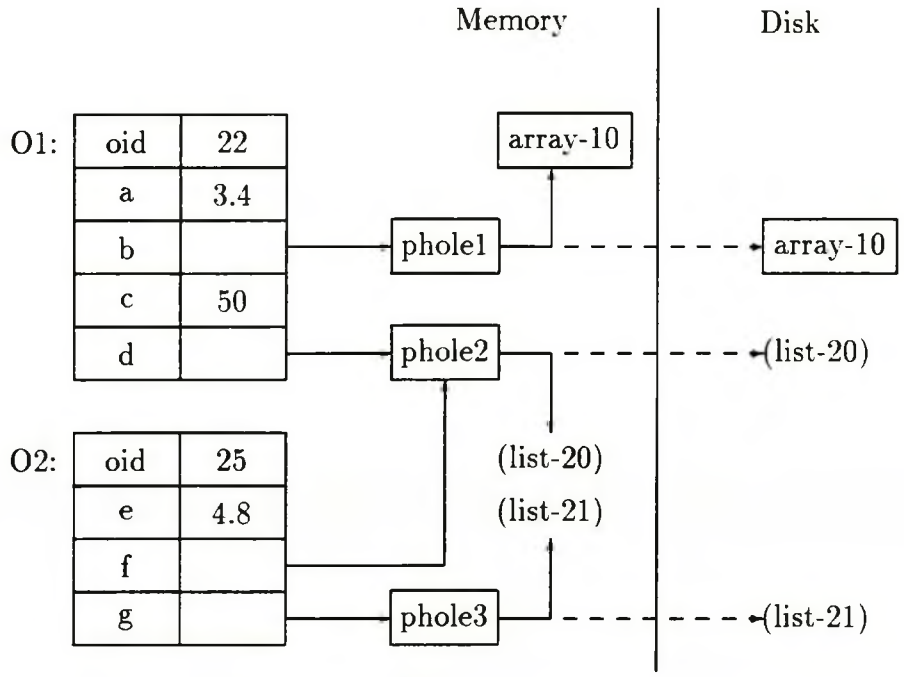


Figure 3.1. Persistent objects in MetaStore

- Supporting persistence granularity at the slot level
- Lazy loading of composite slot values
- Supporting a virtual object memory
- Handling shared structures

A phole contains the identifier of a slot and provides one level of indirection, which is necessary to support the features listed above.

Once a new persistable object is created, it can be repeatedly accessed, read or written. For now, a read access can be viewed as being much like a read on a transient object. A write access is more interesting. The slot accessed for a write is marked dirty in the phole associated with the slot if it is a persistable composite slot of a persistable object. Otherwise, the object itself is marked dirty. Thus, a two level dirty bit scheme is adopted in MetaStore (section 3.10).

At some later time, a repeatedly accessed persistable object is saved if dirty when there is a request for a save, a new checkpoint (section 4.4.2), or a new version (section 4.5), thus becoming persistent. Even after an object is saved, the copy in virtual memory remains until the process that created the object terminates or the object is deleted upon a user's request. When a persistent object is deleted, both copies—the one in virtual memory and the one in the object base—are deleted.

When an object is saved and the process that created it terminates, the life of the object is not terminated, but goes into a dormant state.[2] When another process similar to the one that initially created the object loads the saved object on a user's request, the object is revived and continues its active life. When an object is loaded from the object base, it is treated as clean until modified. Once a loaded object is modified, it is almost like a newly created object. The only difference is that the loaded object has a version stored in the object base, whereas a newly created one does not. Thus, the life of a loaded object continues as if it were a newly created one.

An object is usually loaded as a husk (section 3.6). A husk has pholes for its persistable composite slots without their values instantiated initially, although the persistable atomic slots are always instantiated. The persistable composite slot values of a husk are loaded when they are read accessed, instantiating the pholes. Thus, pholes make lazy loading possible. In Figure 3.1, for example, when `O1` is first loaded as a husk, the loading of `array-10` and `list-20` is delayed until their corresponding slots are read accessed.

When the number of persistable objects in virtual memory reaches a certain limit, some objects (determined by the virtual object memory algorithm described in section 3.12) are flushed to free up some space in order to improve the system

[2]The limited treatment of class evolution currently being supported by MetaStore does not include modified methods.

performance. When an object is flushed, it turns into a husk. A flushed object is exactly like one just loaded as a husk. A husk will never be attempted for flushing.

In Figure 3.1, `list-20` is shared by two objects through slots `d` of `O1` and `f` of `O2`. Handling the persistence of shared structures is also made possible by the use of pholes (section 3.11).

Addition of OID, addition and removal of pholes, slot level persistence, keeping track of dirty bits, lazy loading, virtual object memory, handling the persistence of shared structures, and language syntax extension are all done, most exclusively and others partially, by using the metaobject protocol of CLOS (section 3.8).

Building on this brief overview, the major issues involved in the life cycle of a persistent object are described in detail in the remainder of this chapter. The *Store* portion is detailed in Chapter 4.

3.2 Programmer's View of MetaStore

Some programming conventions are added for the users of MetaStore to follow. A few minor semantic changes to CLOS were also made for performance reasons.

3.2.1 Programming Conventions

A user program must follow some programming conventions and interface with the library provided by MetaStore to use the persistent behavior of objects.[3]

1. Observing a minor extension in syntax of the **`defclass`** macro in CLOS when defining a class.

2. Calling functions to initialize MetaStore, to define keyed classes, to name objects, to save objects, to load objects, to delete objects, to deal with check-

[3]By a user program, we mean the application that uses MetaStore, e.g., CDRS. Thus, a user programmer is one who develops an application using MetaStore, e.g., one of CDRS implementors.

points, to signal dirty objects,[4] to deal with model files, to control the high water mark for the virtual object memory, to deal with clustering, and to deal with versions.

In this section, only item 1 is described. The ones in item 2 are described in sections where the related subjects are described.

The syntax of the **defclass** macro in CLOS must be extended if we want the instances of the class being defined to behave as persistable objects. Let us look at an example of a user defined persistable class to see how one is extended:

```
(defclass student ()
  ((name  :initform "")
   (id    :initform -1)
   (major :initform 'undecided)
   (hobby :initform 'guitar :TRANSIENT T)                    (1)
   )
  (:METACLASS PERSISTENT-METACLASS))                         (2)
```

In the example above, two things are added to deal with persistence: one is mandatory and the other optional. First, the mandatory keyword :METACLASS in

```
(:METACLASS PERSISTENT-METACLASS))                           (2)
```

declares that the class **student** is persistable, thus making all of its instances persistable objects. Second, the optional slot option :TRANSIENT in

```
(hobby :initform 'guitar :TRANSIENT T)                       (1)
```

declares that the slot **hobby** is not persistable, thus making its value nonpersistable even though the rest of the object is. An object is transient if it is not persistable. The longevity of a transient object ends at the termination of the process that created it.

CLOS supports the built-in slot options **:type** and **:allocation** for slot definitions and in MetaStore another one, **:transient**, is added with the semantics given above.

[4] An ideal solution would not require this, but a solution chosen in MetaStore for pragmatic reasons requires it.

3.2.2 Semantic Changes

Semantics of CLOS is preserved with a few minor exceptions. In supporting persistence, the sharing of composite values is restricted to the direct values of composite slots for performance reasons. For example, an array that is referenced by another array that is the value of a composite slot cannot be shared unless it is wrapped as an object (section 3.11.3).

As we will see in section 3.10, we support incremental saves of modified objects. To do this, we maintain dirty bits in each object. Although an algorithm to handle the dirty bits (the ideal solution) that preserves the semantics of the language with minor help from the compiler exists, we chose in MetaStore one that requires some help from user programs, thus not preserving the semantics, because:

- The ideal solution would not be efficient enough for an object-intensive application to use.
- The ideal solution requires some help from the compiler, which is not feasible under our implementation.
- The required help from user programs is not very substantial.

3.3 Object Identity

To support object persistence and sharing, each object must have a unique identity (ID). It is a persistent, or logical, ID, as opposed to a volatile ID, which is the virtual address of an object. Volatile IDs cannot survive the termination of the process that created the object, whereas a persistent ID can when the object is saved in the object base. In MetaStore, a reference in virtual memory from one object or a variable to another object is done via a volatile ID, and the persistent ID is stored as a piece of state information inside the object. The persistent IDs are then used for addressing persistent objects.

In MetaStore, a true object ID (OID) is implemented as a pair of a file ID and an object ID. Because the object base consists of a collection of design/model files,[5] each model file is assigned a unique file ID. Each object in a model file is assigned a unique ID also. Thus, given an OID (a file ID/object ID pair), any object in the entire object base can be identified. A user program has a read access to an OID, but a write access is not allowed.

In the current implementation, a model file can have up to 2^{29} objects, and the object base can have that many model files. This limitation, imposed by the hardware, should not be a problem dealing with a CAD application such as CDRS. A model file considered large in CDRS consists usually of about 50,000 objects.

3.4 Object Persistence

An object begins its life as a persistable object when it is first created as an instance of a persistable class. A persistable object then becomes persistent when it is eventually saved to the object base by one of the following causes:

- It is explicitly requested by a user program to be saved.
- It is saved because a checkpoint (section 4.4.2) is created.
- It is saved because a version (section 4.5) is created.
- It is reachable from a persistable or persistent object.

An object `B` is *reachable* from another object `A` if there exists a chain of references starting from `A` to `B`. Reachability is computed using only the persistable slots of a persistable object.

---

[5] See section 4.1 for the architecture of object bases.

3.5 Structure of Memory Objects

The representation of a normal CLOS object can be viewed as a record structure. For example, an object with four slots: `a`, `b`, `c`, and `d` with the values `3.4`, `array-10`, `50`, and `list-20` respectively, has the structure shown in Figure 3.2.

A persistable CLOS object with the same set of slots and slot values as the one in Figure 3.2 is represented differently in MetaStore. We distinguish atomic slots and composite slots. An *atomic slot* is a slot whose value is of a primitive type. For example, a slot whose value is a number, a symbol, or a string is an atomic slot. A *composite slot* is a slot whose value is not of a primitive type. For example, a slot having an array, a list, a structure, or an object as its value is a composite slot. In Figure 3.2, `a` and `c` are atomic slots, and `b` and `d` are composite slots.

Because the smallest granularity of persistence in MetaStore is a composite slot, we add a placeholder,[6] which is termed a *phole* in MetaStore, between an object and each of its composite slot values if the slot is declared persistable. Thus, a persistent counterpart of the object in Figure 3.2 looks like the one in Figure 3.3. A phole contains information such as dirty bits for incremental saves, reference counts to deal with sharing, version information, slot ID, and the value of the slot which would be `nil` if it is not yet loaded.

[6]MultiScheme [35] uses placeholders as the primary vehicle connecting the scheduler with the underlying support for parallel processing.

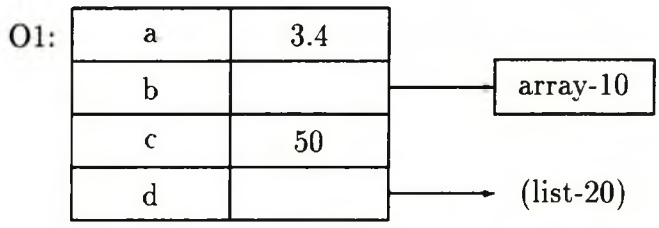


Figure 3.2. A transient CLOS object

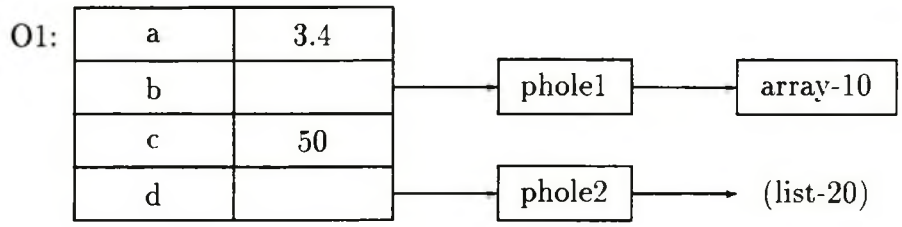


Figure 3.3. A persistable CLOS object

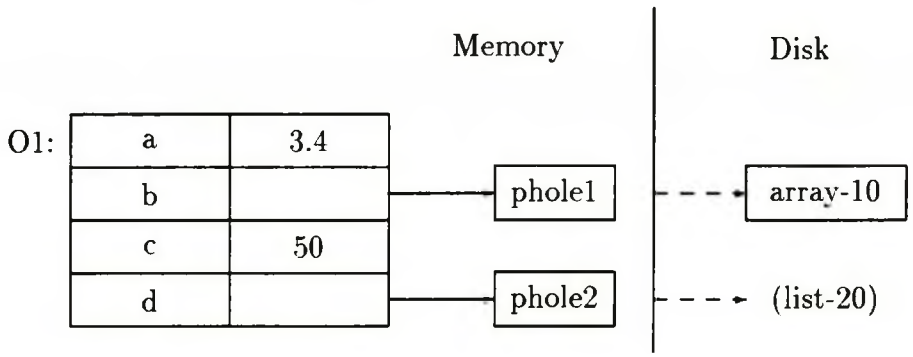


Figure 3.4. A husk object

3.6 Husks and Granularity of Persistence

When an object is to be loaded from disk, only the *husk* of the object is first loaded. A husk object has all of its persistable atomic slots instantiated, but each of its persistable composite slots points to a phole. A phole at this point is empty except that it knows where in disk the value of the slot is located. When the object in Figure 3.3 is first loaded as a husk object, it would look like the one in Figure 3.4. When a persistable composite slot in a husk is not yet loaded, the phole corresponding to the slot has enough information on how to get the value of the slot, resident in disk. The notion of a phole is much like *future* in Multilisp [17]. As a loaded persistent object (a husk) is accessed, it eventually becomes a fully instantiated memory object.

PCLOS [40] also uses the notion of a husk object, but with a rather different

meaning. A husk in PCLOS is a placeholder for an object, an instance of a class, and it is used to make memory-resident references to the instance, that is not yet loaded, work properly. Thus, a husk there is much like a phole in MetaStore, except that a phole is used for a composite slot, whereas a husk in PCLOS is used only for objects. The notion of a husk object as used in MetaStore does not exist in PCLOS.

One of the main differences between PCLOS and MetaStore is that of the granularity of persistence: the smallest grain size in PCLOS is an object, whereas that in MetaStore is a composite slot.

Based on our experience with CDRS, on the average about 80% of an object is occupied by composite slot values, and it seemed important to control persistence at this grain size for the following reasons. With the file-based, batch-mode object persistence in CDRS, a realistic car model would occupy so much of virtual memory that there were times when the system could not even complete a garbage collection, resulting in a system crash. Even when it managed to continue without a crash, the performance of the system was usually unacceptably bad because of frequent garbage collections. With a large car model loaded, each garbage collection takes several minutes.

To keep the amount of data in virtual memory at a reasonable level to achieve acceptable performance, MetaStore was designed in such a way that some of the objects, which might not be immediately needed, can be removed from virtual memory. Removing objects entirely from virtual memory causes problems with objects being shared by multiple objects or program variables. These problems with shared objects happen because the removed objects would be loaded again when they are accessed again. If a program variable is hanging on to the object that MetaStore assumes removed, then (i) the space that MetaStore assumes reclaimed has not been reclaimed and (ii) sharing is not consistent any more, i.e., there are

at least two copies of the same object in virtual memory. Because the majority of the space occupied by an object is occupied by the composite slot values, we decided to control the persistence granularity at the composite slot level. This way, we can restore most of the space occupied by an object by flushing it, thus turning it into a husk (section 3.12), without disturbing the sharing. Detailed discussion and solutions for shared objects and structures are in section 3.11.

3.7 Addressing Mechanisms

One of the fundamental issues in implementing persistence is that of addressing mechanisms. The goal is to support object persistence with referential transparency. There are two different address spaces where an object can exist at any given time: virtual memory and/or disk. When an object moves from one space to the other, the address that identifies the object has to adapt to the new address space. An address in memory is called a *virtual address,* and an address in disk is called a *logical address.* A virtual address is a volatile ID, whereas a logical address is a persistent ID.

For example, a reference from the object `A` to another object `B` in memory is done via a virtual address. When `B` is saved to disk and loaded back in again, it is not likely that `B` will restore back again the same virtual address as `A` would still be using in referencing `B`. Thus, we have to provide a mechanism by which virtual addresses are translated to logical addresses on the way out to disk, and logical addresses are translated to virtual addresses on the way back in from disk while maintaining referential transparency.

Although there are many ways to deal with the addresses of persistent objects as described in [12], we only consider the mechanisms that are appropriate at the metalevel since MetaStore is supporting persistence at that level. Here, we focus on two most feasible candidates:

- *Associative OID Addressing:* Keep the logical addresses at all times and perform an associative lookup of a logical address to find the virtual address on every object reference.

- *Pointer swizzling:* Translate addresses back and forth as objects travel between the two spaces.

Let us look at these two schemes in detail.

3.7.1 Associative OID Addressing

In this scheme, one persistable object refers to another in terms of an object identity (OID). An associative lookup table is maintained to map an OID to the virtual address of an object. Whenever a reference is made to an object, which is in the form of an OID, the OID is looked up in the table to find the location of the referenced object in memory. Thus, translation of OIDs are done repeatedly.

Because no reference translations are done while an object is being saved or loaded, there is smaller cost at the time of loading and saving. However, there is a constant cost on every object reference while the object is in memory. This may be too inefficient for the object-intensive applications that MetaStore supports. A fast access to an object and a fast reference from one object to another are a few of the more critical aspects in an object-oriented language, and this scheme would add a constant overhead to these.

3.7.2 Pointer Swizzling

When an object is ready to be saved, all the object references contained in the object are converted to logical IDs. In the process of loading an object, all the logical IDs contained in the object are converted back to virtual addresses[7] by

[7]A variation of this algorithm is to load without any translation during the load. Reference translations are then done when it is first referenced, thus lazily. Once a translation is done,

using an object table. The object table contains the mapping from persistent IDs to virtual addresses. Between loading and saving, the loaded objects will usually be modified repeatedly. During this middle phase, an object reference is done at the usual virtual address reference speed.

It is possible that an object that is referenced is not yet loaded. That is, the object that is making the reference is pointing to a phole, whose value is not instantiated. When the referenced object is accessed, it is then loaded. Thus, loading of objects is delayed until it is required.

In a design environment with object-intensive applications, a typical transaction tends to load a set of objects and work for a while on them before the objects are saved again. It tends to spend more time during the "workForAWhile" phase compared with the loading and saving phases. During the middle phase, the design is repeatedly modified, at which time access patterns are unpredictable, and a fast response time is more important than the system throughput. Thus, it is important to minimize the overhead during the middle phase.

Because MetaStore is designed to support this kind of design environment, pointer swizzling is chosen over associative lookup scheme in MetaStore.

3.8 Metaobject Protocol in Action

Supporting object persistence in CLOS without any help from the compiler or run-time support system requires dealing with the following:

- Separating the transient from the persistable.
- Adding and removing *pholes* at the time of a creation of or an access to persistable objects.

---

keep the translations in place. This saves some time at the loading phase, but still burdens the "workForAWhile" phase that we will describe below.

- Intercepting read and write accesses to objects to support a virtual object memory.
- Supporting persistence via inheritance.
- Intercepting read accesses to objects to support lazy loading.
- Intercepting write accesses to objects to update dirty bits.
- Adding the slot option `:transient` to control persistence at the slot level.
- Realizing the notion of a *husk*.

In MetaStore, these are handled via the metaobject protocol of CLOS. CLOS consists of five basic building blocks: classes, slots, generic functions, methods, and method combinations. The structure and behavior of each building block are defined and implemented by the metaobjects of the basic building block at the metalevel. For example, the semantics of slot accesses and the internal structure of slots are defined by the `slot-definition` metaobject.

In MetaStore, the structure and behavior of some of the metaobjects in the metaobject protocol are modified in such ways that each of the requirements listed above is accomplished. In this section, we describe the modifications that are made to those metaobjects. Both structural and behavioral aspects are described, using some sample code extracted from the working implementation.

3.8.1 Persistent Class Metaobject Class

First, a *specialized class metaobject class* [23], `persistent-metaclass`, is defined as a subclass of the standard metaobject class, `standard-class`, as shown below.

```
(defclass persistent-metaclass (standard-class)
  ())
```

The new class specifies the same structure and behavior as its superclass at this point. Although the structure will remain the same, if we were to want to keep extra information at the class metaobject class level, we would add some slots to this class. An instance, a metaobject say `M`, of this class defines the behavior of a user class, say `student`, whose metaobject is `M`. Thus, the behavior of an instance of `student` is defined by `M`. By modifying `M` using the mechanism established by the metaobject protocol, we can achieve the extension of object persistence. These modifications are described in the rest of this and coming subsections.

3.8.2 Creating Persistable Objects

At the time a persistable object is created by a call to `make-instance`, several things are taken care of: (i) Pholes are added to each composite slot if the slot is declared persistable. (ii) The necessary information for the virtual object memory (section 3.12) is stored in the data structure that is used by the virtual object memory. (iii) The necessary information for handling structure sharing (section 3.11) is also stored in the appropriate data structure.

3.8.3 Persistence via Inheritance

Persistence of program data in PS-Algol [13] is done via the principle of orthogonal persistence. In MetaStore, we chose to do it via inheritance. Treating all the data in a program as persistable as it is done in PS-Algol is impractical since about 90% or more of data that a program deals with are never meant to be saved. Persistence via inheritance can easily partition all the objects of a program into two groups: persistable and transient. Even for the objects that are instances of a persistable class, only a small subset of them ends up being saved. `persistent-root-class` is defined in MetaStore and it is inserted into the *class precedence list* [23] of each persistable user class. Insertion of the root class is done

at `initialize-instance` phase of class definition [23]. This way, a user class does not explicitly have to include the persistent root class as one of its superclasses. The persistent root class is meant to be invisible to user programs. Because the root class handles the object persistence, the implementation of persistence is localized to one class and can easily be modified if necessary.

The purpose of `persistent-root-class` is two-fold: (i) Structurally, it adds extra information such as an object ID (`oid`) and a dirty bit (`dirtyp`) to each object as in:

```
(defclass persistent-root-class ()
  ((oid    :initform (make-oid :oid (get-next-osid)
                               :mid *current-model-id*))
   (dirtyp :initform t :initarg :dirtyp :accessor dirtyp)
   )
  (:metaclass persistent-metaclass))
```

(ii) Behaviorally, by defining a method on `persistent-root-class`, the following are supported by the persistent root class.

- It provides the default method for checking the consistency of objects before they are saved. In general, the default method would be a dummy routine. Its role is to provide a method name that both MetaStore and application programs know about so that MetaStore can send this message just before an object is saved. Although MetaStore currently does not perform any consistency checking, it could if necessary. One possibility would be to check to see if an object has a valid object ID. A user program would either overwrite the default method or define an `:after` method if it desires to do some checking before an object is committed for saving. It is important to make sure each object being saved is consistent, so that it will be useful when it is loaded back for a later use. What it means for an object to be consistent is left to be defined by user programs although MetaStore could augment the meaning if necessary. It is quite common for an object to have "wrong" data in an application like

CDRS which can be fixed by this routine before it is saved. It is possible to continue from a bug by ignoring it, thus making the data in virtual memory inconsistent. When these inconsistent data are saved by MetaStore, they must be fixed. Because MetaStore is not smart enough to fix inconsistencies caused by application programs, it is important for an application program to have the chance to fix any anomaly of an object before it is saved.

- It handles flushing out objects if the virtual object memory algorithm decides that an object is to be flushed out (section 3.12).
- It handles encoding of objects for saving and decoding of them during loading. Address translations are done as a part of this process.

3.8.4 Accessing Objects

Each access, read or write, is intercepted by using the metaobject protocol so that appropriate persistence related actions can be handled.

3.8.4.1 Read Access

On a read access, if the accessed slot is a persistable composite slot which is not yet loaded, then the value of the slot is read in from disk. If the slot is a transient slot or an atomic slot, then the value should already be in memory and is returned. All this is handled by modifying the behavior of `slot-value-using-class` method, which is the workhorse of the user accessible routine `slot-value`. It is done by defining an `:around` method to `slot-value-using-class`.

3.8.4.2 Write Access

A write access is more complicated than a read access. On a write access, in addition to the default behavior, the following are taken care of by MetaStore with the help of the metaobject protocol:

- If a transient slot is accessed, do nothing extra. Otherwise, do the following.
- If an unbound[8] slot is accessed, then set the dirty bit of the object accessed. In addition, if the new value coming in is a composite value, then add a phole with its dirty bit set.
- If both the current and new values are atomic, set the dirty bit of the accessed object.
- If the current value is atomic and the new value is composite, add a phole to the slot with the dirty bit set and the reference count updated. Reference counts are used to handle the sharing of composite slot values (section 3.11). Also set the dirty bit of the accessed object.
- If the current value is composite and the new value is atomic, update the reference count of the current value, and set the object's dirty bit. This is the case where a phole is removed.
- If both the current and new values are composite, update reference counts of both, and set the dirty bit of the new value.

All this is handled by modifying the behavior of `(setf slot-value-using-class)`, which is the workhorse of the user accessible routine `(setf slot-value)`. It is done by defining an `:around` method to `(setf slot-value-using-class)`.

3.8.5 Creating Husks

When an object is created via a call to `make-instance`, it happens in two steps:

- An instance is allocated by the routine `allocate-instance`.

[8] A slot is said to be *unbound* if it has no value at all.

- The allocated instance is initialized with values specified for `initforms` in slot definitions.

When an object is to be loaded from disk, we could create an instance, say `O1`, by calling `make-instance` and then replace the slot values of `O1` with the saved values to fully recover the original state of the saved object. This wastes some time and space because the initialized slots by the values specified for `initforms` will immediately be replaced by the values being loaded from disk.

To eliminate this waste, we create a husk. A husk is also created in two steps:

- An instance is allocated by the routine `allocate-instance`.
- Only the transient slots are initialized with values specified for `initforms` in slot definitions.

The atomic persistable slots of a husk are instantiated by the saved values and the composite persistable slots are instantiated with empty pholes. These pholes will then be instantiated with their values when the slots are accessed.

There is one minor concern with skipping the execution of `initforms` of persistable slots if they are meant to create some side effects besides initializing the slots. Our experience with CDRS shows that this has not been done, and I believe this is not a good programming practice. If a user program must use, or abuse, the `initforms` to create side effects, it can easily be done in some other way. For example, an `:after` method for `make-instance` would be a preferred way of accomplishing the same.

3.8.6 Persistent *Slot-Definition* Metaobject Class

While a class definition is processed, a *slot-definition* metaobject class is created for each slot. We must add an extra slot option, `:transient`, so that each slot can be declared as transient or persistable. We must do this in two different places:

- *standard-direct-slot-definition:* Instances of this class hold intermediate, not fully processed slot-related information from the class definition form. We define *persistent-standard-direct-slot-definition* as a subclass of *standard-direct-slot-definition* with an extra slot, `transientp`, and its *:initarg*, `:transient`.

- *standard-effective-slot-definition:* Instances of this class hold slot-related information that has been fully processed, *finalized*, using inheritance rules, thus ready to be used at run-time. We define *persistent-standard-effective-slot-definition* as a subclass of *standard-effective-slot-definition* with an extra slot, `transientp`, and its *:initarg*, `:transient`.

Thus far, we have taken care of the static parts. We also have to tell the system which *slot-definition* metaobject class should be instantiated to implement each persistable slot. We do it for both `persistent-standard-direct-slot-definition` and `persistent-standard-effective-slot-definition`. These are used by two generic functions: the former is used by *direct-slot-definition-class* and the latter by *effective-slot-definition-class*.

Both initialization and reinitialization of instances are funneled to the generic function `shared-initialize`. Here, we first make the value of slot option, *:transient*, be available for use.

There is one more thing to take care of. A rule for inheritance regarding transience of slots has to be enforced. A slot is treated transient only if all classes in the inheritance chain that define a slot with that name has the same declaration. This is done at the time effective slot definitions are computed by the generic function, `compute-effective-slot-definition`.

3.9 Incremental Loading

MetaStore supports incremental loading. An application using MetaStore can load a set of objects by queries using names or keys (section 4.2). Once an object

is loaded as a husk, the values of the unloaded composite slots will be loaded on demand. An object can be loaded either as a pure husk or as a husk with some of its composite slots instantiated.

An object-intensive, graphical user interface based application like CDRS has a rather difficult problem, compared with an application that interfaces, e.g., an airline database. When a user of CDRS is ready to edit an existing car model, he would have to load enough data to see on the screen what is already there in the design. That is, we have to preload a subset of objects at the time of opening a model file. In CDRS with MetaStore, some subset of *keyed* objects (section 4.2) will be loaded as husks with the composite slots, that make sense to be loaded initially, instantiated. Some of these composite slots are needed for displaying and others are data that are commonly needed for most of user operations. The objects that are intended to be preloaded are clustered (section 4.7) so that they can be loaded efficiently. Once the preloading is done, other objects and composite slot values are loaded lazily.

3.10 Incremental Saving and Dirty Bits

Objects become persistent, i.e., saved, by one of requests such as the ones listed in section 3.4. In all cases, only *dirty* objects and slots are saved. Because the smallest grain size of persistence is a composite slot, it is possible that a composite slot of an object is saved, but not the object that owns the slot. That is, each persistable object and persistable composite slot value has its own dirty bit.

When an object is ready to be saved, the saving algorithm works in three phases:

- All the dirty persistable composite slots of the object are saved first.
- The object itself is then saved if dirty.
- The object table is finally saved to activate the changes in the object base.

Let us introduce the notion of a *network*. Given an object, a collection of all the objects that are *reachable* from the given object is called a network.

When a user program requests an object be saved, the network rooted at the given object is saved. In this case, the user program may only be interested in saving an object, i.e., network, even if there might be other objects that are dirty. Just because a user program requests an object be saved, we can not save all the other dirty objects at the same time. Cyclic references are also handled.

When a user program requests a checkpoint (section 4.4.2) or a version (section 4.5) be created, however, the situation is different. The semantics of creating a checkpoint or a version is that all the modified persistable objects are saved. We could save all the dirty objects on a checkpoint or a version by handling each object in the same way as when a user program requests an object be saved, in which case we would have to traverse each composite slot values of each dirty object to compute the networks. However, we can do much better. We can bypass all the traversing of composite slot values and treat each object as a leaf node. We can do so because we can easily find all the dirty objects and composite slots in the object table, which is memory resident at all times. The object table contains all the pholes for both objects and composite slots. Each entry in this hash table is a pair, where the key is an OID and the value is a phole.

How do we keep track of dirty bits? We have to maintain dirty bits for each persistable object and its composite slots. There are different algorithms with different characteristics: some positive and others negative. In the sections below, we describe the problem in detail, possible solutions, and the design decisions we made for MetaStore.

3.10.1 The Problem

A read access to a persistable object is not a concern. Write accesses are the ones that concern us. Even for a write access, modifying slot values via the public interface, i.e., via `(setf slot-value)`, is not a problem. The problem is with changing a slot value through the back door. For example, the following piece of code demonstrates the problem:

```
(let ((arr1 (slot-value object1 'slot1)))
  (setf (aref arr1 3) 4.5))
```

Here, the value, an array, of the slot `slot1` is read accessed and locally bound to `arr1`. The array is then modified. However, this modification is not registered in the phole of `slot1`, thus the dirty bit in the phole of `slot1` is not current. To make sure this change is reflected in the phole of `slot1`, the user program could do the following:

```
(let ((arr1 (slot-value object1 'slot1)))
  (setf (aref arr1 3) 4.5)
  (setf (slot-value object1 'slot1) arr1))                    (1)
```

The extra call, labeled `(1)`, would solve the problem since `(setf slot-value)` part of the metaobject protocol in MetaStore is keeping track of the dirty bits. However, this extra call is normally not necessary in a CLOS program. That is, requiring this extra call changes the semantics of CLOS.

Although we used an array as an example here, all composite values except objects pose this problem.

3.10.2 An Ideal Solution

The goal is to support incremental saving without requiring application programs to deal with dirty bits at all. That is, we want dirty bits to be transparent to user programs. We describe an algorithm that satisfies this goal, and point out why this solution may not be practical.

Although all nonobject composite values pose this problem, we will concentrate on arrays, which typify the problem, for describing this algorithm. If each composite value has its own dirty bit, then the problem becomes simpler. However, that would be too expensive, so we are adding one to a composite value only when necessary—lazily. This solution requires a few extra things:

- A hash table containing dirty composite values.
- Overload the write access routine for arrays, i.e., `(setf aref)`.

When an array `a1` is write accessed as in:

```
(setf (aref a1 3) a-new-value)
```

`a1` is added to the hash table, say `ht1`, containing the dirty composite values. Note that only dirty ones are added to the hash table, so the number of entries in the hash table would stay small most of the time because of frequent incremental savings.

Now, we can overload `(setf aref)` as follows:

```
(defsetf aref (arr &rest subscripts) (new-val)
  `(progn (setf (gethash ,arr ht1) t)
          (setf (sysaref ,arr ,@subscripts) ,new-val)))
```

This new version of `(setf aref)` adds the array being modified to the hash table in addition to the original functionality of changing an element of an array. This assumes that the workhorse system routine for `aref` is `sysaref`. If we don't have access to this routine,[9] then we will have to require our application programs to use a new name for `aref`, say `paref`, which would require the following definitions instead:

```
(defmacro paref (arr &rest subscripts)
  `(aref ,arr ,@subscripts))

(defsetf paref (arr &rest subscripts) (new-val)
  `(progn (setf (gethash ,arr ht1) t)
          (setf (aref ,arr ,@subscripts) ,new-val)))
```

[9]We don't have access to this routine in the context of implementing MetaStore for CDRS since we are using Lucid Common Lisp [30].

Thus, a user program would have to use `paref` and `(setf paref)` for any array that can potentially be persistent. For arrays that will never be persistent, `aref` and `(setf aref)` can still be used. This has trade-offs though. On the positive side, for arrays that will never be persistent, we can save some time and space since these arrays will bypass the hash table interface. On the negative side, it is possible that `(setf aref)` can mistakenly be used for an array that is potentially persistent, in which case what is saved in the object base would be inconsistent with what is in the virtual memory. To avoid this possible inconsistency, we can require all user programs use `paref` and `(setf paref)` at the expense of some overhead.

When a persistable object is ready to be saved, we save each persistable composite slot value in two phases as follows: (i) Since the smallest granularity of persistence is a composite slot, we first traverse the value of the given composite slot to see if any of the composite values that is a part of the slot value is dirty. If we find a dirty one, remember the fact that there was a dirty one, thus the entire slot value can be treated as dirty, and continue the traversal to mark all the rest clean. Marking one clean means removing one entry from the hash table. (ii) If found dirty, then save the entire slot value as one entry in the object base. If none was dirty, then skip to the next slot value. Once all the composite slot values are saved, we finally save the husk itself if dirty.

3.10.2.1 Feasibility

Let us examine the potential problems with this solution. Some are performance related, and others are related to using a Common Lisp system that does not allow source level changes.

- Each write access to a composite value is burdened by adding an entry to the hash table, `ht1`.

- The entire slot value for each persistable composite slot of an object to be saved has to be traversed to see if any is dirty. For each composite value encountered during the traversal, at least one look up in the hash table is also required; twice if a dirty one is found.

- When a checkpoint (section 4.4.2) or a version (section 4.5) is created, all the persistable objects have to be traversed to locate dirty objects and dirty composite slot values.

- Dealing with an application that is already built and needs to be ported to MetaStore, although minor, there is resistance from programmers on changing the syntax of the language. For example, we would have to change from `aref` to `paref`.

Although the hash table in general would contain only a small number of entries, the cost of adding an entry to and looking up an entry in the hash table on each write access and at saving time concerns us somewhat. Having to traverse each composite slot values given a short list of objects to be saved also concerns us somewhat. However, what concerns us the most is having to traverse *all* the persistable objects on the creation of a checkpoint or a version, which not only would take a lot of time but also may be done quite frequently in the case of checkpoints.

3.10.3 A Practical Solution

What about reality?
Jeannette Wing

Extra cost associated with the ideal solution described in the previous section forces us to use a compromise solution that seems more pragmatic, although it puts more burden on application programmers.

3.10.3.1 User Programs' Responsibilities

User programs are required to tell MetaStore about what is becoming dirty. If a slot value is modified via the public interface, i.e., `(setf slot-value)`, then there is no extra responsibility on the part of an application program. However, if a user program modifies a slot value through a back door, then it must inform MetaStore that there is a modification being made, using one of the following ways:

- `(mark-dirty object1 &rest slot-names)`: For each slot in `slot-names`, `object1` is marked dirty if the slot is a persistable atomic slot, and the slot itself is marked dirty if the slot is a persistable composite slot. If no slot name is given, i.e., `slot-names` is `nil`, then `object1` and all the persistable composite slots are marked dirty.

- Instead of the usual `with-slots` macro of CLOS, user programs can use an alternative, `with-pslots`, with the following syntax and semantics. In fact, every occurrence of `with-slots` can be replaced by a `with-pslots`.

  ```
  (with-pslots (slot1 (slot2 :dirty) (slot3 :dirty)) object1
    (body-of-usual-with-slots))
  ```

  This says that somewhere in the body of this construct, the values of `slot2` and `slot3` will be modified, possibly through a back door. `with-pslots` will signal to MetaStore that these slots are dirty and then call `with-slots`.

3.10.3.2 Dangers and Remedies

Writing a new application using MetaStore with these extra responsibilities is not too bad, but porting an existing code is rather worrisome. If a mistake is made by not informing MetaStore correctly on dirty bits, there is no guarantee that what is saved in the object base is correct. Furthermore, the user of the system will not find it out until it is too late. That is, you save a model, restart the system, load

it, and find out that it is not the same as what you thought you saved. Another danger is possible hidden modifications on composite values by a library routine. A user program calling a library routine should understand the possible modifications and mark the dirty bits appropriately.

To remedy these dangers, MetaStore provides tools that detect possible programming errors. These tools would be used while the application is being developed and tested before delivery. The following are the possibilities:

- Regardless of what the dirty bit shows, before each object or a composite slot is saved, compare the value in memory with the one in disk. If they are different and the dirty bit was set to clean, then a programming error is found. Although this is a slow process, it will catch the programming errors.

- Implement the "ideal solution" and see if there is any discrepancy between what the "ideal solution" shows and what this "practical solution" shows about dirty bits.

At least one should be provided as a tool for application programmers, and we chose to provide the first one for the initial implementation of MetaStore.

3.10.4 Another Practical Solution

The main disadvantage of previous practical solution was the danger of missing a dirty persistable composite slot in case it is not marked correctly. Furthermore, we will not be able to find out about an incorrect save until it is too late. In this section we slightly modify the previous practical solution to correct this danger.

3.10.4.1 User Programs' Responsibilities

Instead of requiring the programmer to mark what is dirty, we request him to identify what is clean. With this algorithm, the default is that all persistable

composite slots are assumed dirty unless identified by a user program as clean. A user program would use:

```
(mark-clean object1 &rest slot-names)
```

to tell MetaStore that each slot in `slot-names` is clean for `object1`. If no slot name is given, i.e., `slot-names` is `nil`, then all the persistable composite slots are marked clean. If a persistable atomic slot is included in `slot-names`, then `object1` itself will be marked clean.

3.10.4.2 Difficulties

This algorithm lessens the danger described in the previous solution. It is not likely to miss any dirty slot as long as application programmers stay conservative. That is, unless they are absolutely sure that a slot is clean, they do not mark it clean. This way, diligent programmers will get better performance than lazy ones. This seemingly elegant solution has its own problems and they are:

- It is likely that a fair amount of clean slots will be saved because the application programmer intentionally or unintentionally misses marking them clean.
- There is no good communication channel available between application programmers and MetaStore so that the marking can be done. When an object is accessed, read or written, in a body of a method, some of the slots can be marked clean. For a class with many slots, it would be hard to know which slots are for sure clean. Because it would be hard to know which methods mark which slots clean, fewer slots would tend to be marked. An even more serious problem is the case where an object is loaded and never accessed, read or written. Because of the preloading (section 4.3.1) done on many objects, it is quite likely that some of these objects are never accessed, so are never given the chance to mark their slots clean. The metaobject protocol can not

help for marking slots clean on slot accesses since it is not correct to mark all slots other than the one being accessed to be clean. Because of these reasons, we predict that application programmers would end up marking only a small fraction of all that can and should be marked clean, escalating the time spent on saving objects.

Because of these reasons, we chose to implement the first practical solution for MetaStore with the tool described in the previous section.

3.11 Shared Structures

We have seen better days.
William Shakespeare

There is almost no limit on what can be shared by what in Common Lisp. That is, a data structure in Common Lisp can have arbitrary relationships with other data structures. For example, structures such as arrays or lists can freely be shared by variables, arrays, lists, objects, etc. This unlimited freedom adds much difficulty to dealing with these structures in the context of supporting persistence.

One of our research goals was to support persistence without changing the semantics of the language with no compiler or run-time support. This turns out to be a rather lofty goal.

There is a solution that fully supports sharing in the context of persistence, although the cost of doing so may be too high for an object-intensive application. There is also a solution that limits the sharing of persistent structures, but the limitations may not be too severe to use. After illustrating the problem, we present an ideal solution, and a compromise solution that may be practical enough to use in object-intensive applications.

3.11.1 The Problem

When an object or a structure has a unique ID, sharing of that object or that structure is easy to support. When an object or a structure has a unique ID,

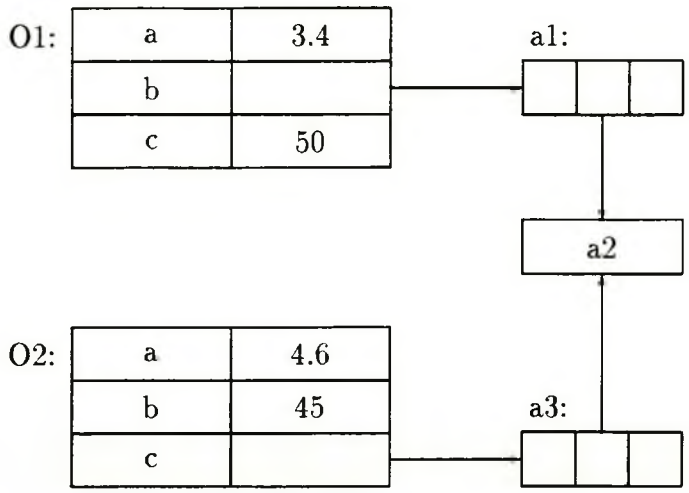


Figure 3.5. An array shared by two objects

one level of indirection such as a phole can be used, thus being able to handle the persistence of shared structures correctly. Because each object has a unique ID, sharing objects is not a problem. Sharing structures such as arrays and lists however is a problem since they normally don't have their own IDs. In one solution we describe below we actually add a unique ID to each composite value. We will also examine the cost of doing so. As with dealing with dirty bits (section 3.10.2), we will concentrate on arrays while describing the algorithm since they typify the problem.

Suppose a composite slot value, the array `a1`, of the object `O1` in Figure 3.5 is ready to be saved, and also suppose that `a1` has another array, say `a2`, as one of its elements. Further suppose that a slot of another object `O2` also has `a2` as its value through a third array `a3`. Thus, `a2` is shared by `O1` and `O2`.

This sort of sharing is perfectly legal in Common Lisp. Assuming only objects have dirty bits, and also assuming both `O1` and `O2` are dirty, if both `O1` and `O2` are saved, two copies of `a2` will be saved: once by `O1` and another by `O2`. When `O1` and `O2` are both loaded at some later time, `b` of `O1` and `c` of `O2` will have their own

copies of the original array `a2`, say `a2-1` and `a2-2`.

This is a rather simple example of sharing. To handle the sharing of composite values in general, we have to design an algorithm that handles all the cases in full generality.

3.11.2 An Ideal Solution

Supporting persistence for composite values in full generality requires a unique ID for each composite value that is being shared by different objects. Rather than blindly adding an ID to each potentially sharable composite values, we only add one when necessary—lazily.

Here again, we use arrays as an example to describe the algorithm. This solution requires a few extra things:

- An associative lookup table, say a hash table. This is an *"eq"*-test hash table, where a key is a virtual address for an array and a value is either an array or a phole.
- Overload the write access routine for arrays, i.e., `(setf aref)`.

With this algorithm, Figure 3.5 would now look like Figure 3.6. When the array `a2` becomes a value of another array `a1` by a call:

```
(setf (aref a1 1) a2)
```

the following analysis, in addition to what happens dealing with dirty bits, is undertaken:

1. If `a2` is in the hash table, say `ht2`, then:
   (a) If the value part of `a2` entry in `ht2` is a phole, say `phole4`, then add `a1` to the reference count, `referrers`, of `phole4`. This means that there are at least three structured data that share `a2` after this update.

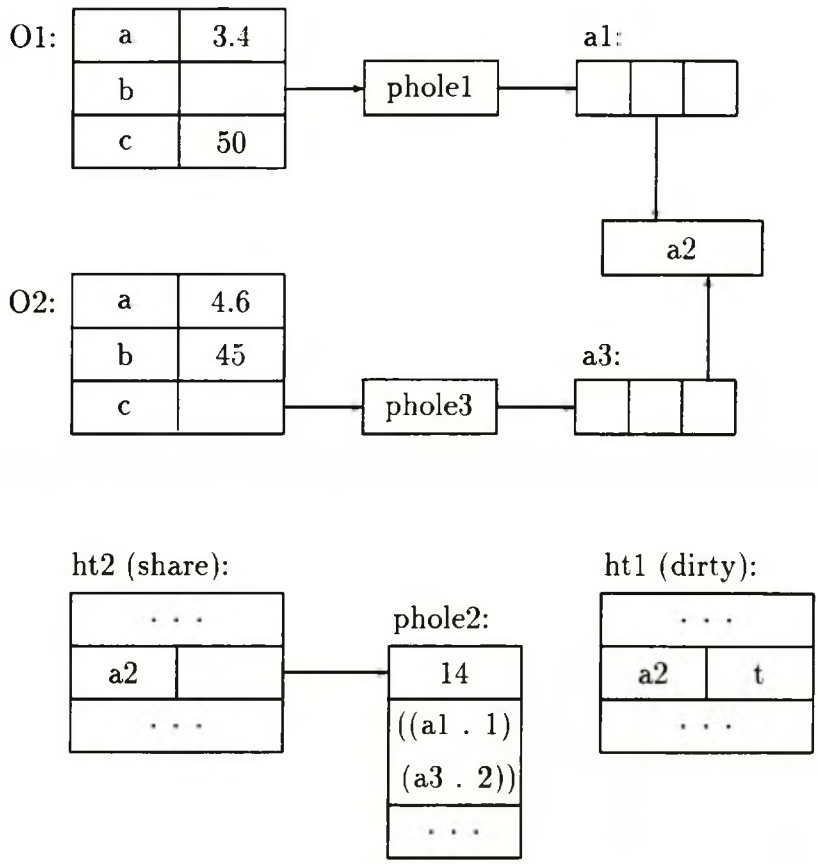


Figure 3.6. A persistable array shared by two persistable objects

(b) If the value part of `a2` entry in `ht2` is not a phole, but another composite value, say `a3`, then create a phole, say `phole2`, and add `a1` and `a3` to `referrers` of `phole2`. This means that there are exactly two structured data that share `a2` at this point. This is where the lazy creation of pholes happens, and a phole is not created until at least two things are sharing an array. This case is illustrated in Figure 3.6.

2. If `a2` is not in `ht2`, then add a new entry to `ht2` with `a2` as the key and `a1` as the value. This means that `a2` is not shared yet, but potentially could be. This entry is added without creating a phole because sharing has not happened, yet.

This algorithm adds a phole to each composite value when it is shared by at least

```
(defsetf aref (arr &rest subscripts) (new-val)
  `(progn
     (setf (gethash ,arr ht1) t)            ;for dirty bit
     (when (composite-p ,new-val)           ;for sharing
       (let ((in-ht2 (gethash ,new-val ht2)))
         (if in-ht2
             (if (phole-p in-ht2)
                 (push ,arr (phole-referrers in-ht2))
                 (let ((phole2 (make-phole)))
                   (push in-ht2 (phole-referrers phole2))
                   (push ,new-val (phole-referrers phole2))))
             (setf (gethash ,new-val ht2) ,arr))))
     (setf (sysaref ,arr ,@subscripts) ,new-val)))
```

Figure 3.7. Modification to array updates for sharing

two others. The added phole, however, does not affect read accesses of composite values. It only affects write accesses.

`phole1` and `phole3` in Figure 3.6 are there since each composite slot has its own unique phole unless it is sharing a slot value with one or more other slots. `phole2` is there because of this algorithm. That is, `phole2` is added because `a1` and `a3` are sharing `a2`. `phole2` contains the information about sharing.

Figure 3.7 shows how the overloading of a write access routine is modified. Arrays are used in the example code, thus modifying the array mutating operator (`setf aref`). The routine in Figure 3.7 assumes that the workhorse system routine for `aref` is `sysaref`. If we do not have access to this routine, then we will have to require our application programs to use `paref` instead of `aref` which would require similar definitions instead as we saw with dirty bits (section 3.10.2).

Suppose now we want to save `O1` and `O2`, both dirty. Suppose also that `a1`, `a2`, and `a3` are dirty. First, by saving `O1`, we will save `a1`, `a2`, and `O1`. At that point, they will also be marked clean. Second, by saving `O2`, we will save `a3` and `O2`, and mark them clean. Note that by the time `O2` is saved, `a2` is already clean, thus not

saved multiple times. Also note that this algorithm saves each composite value as a separate entity in the object base.

Suppose now `O1` and `O2` are loaded in that order by a different process at some later time. When the value of `b` of `O1` is loaded, the value of `phole1` will be set with `a1`. The second element of `a1` will then get `a2` loaded by using the slot ID, `14`, in the object table. Note, however, that `a1` references `a2` directly rather than the phole, `phole2`. When the value of `c` of `O2` is loaded, `a2` would already be loaded and `a3` will reference `a2` directly again by going through the object table.

When an object is declared deleted by a user program, MetaStore releases all the handles that MetaStore is keeping track of. They include both hash tables: one for dirty bits and the other for sharing. That way, what a user program considers garbage will be guaranteed to be garbage as long as user programs do their share of clearing unneeded references.

3.11.2.1 Feasibility

Let us analyze the cost involved with this ideal solution. Because this solution depends on the "ideal solution" for tracking dirty bits, supporting this ideal solution incurs the following extra cost on top of the cost of dirty bits (section 3.10.2):

- Each write access to a composite value is burdened by the case analysis done in the presented algorithm along with dealing with the hash table `ht2` for sharing.

- With this solution, each composite value is saved as a separate entity in the object base, thus the granularity of persistence is a composite value rather than a composite slot value. This algorithm would suffer much more dealing with disk I/O because of the small grain size.

- The hash table, `ht2`, will in general be quite large because of many small arrays, e.g., arrays of length three for point data in an application such as CDRS as described in section 3.10.2, thus affecting each write access potentially severely.

- Although many small arrays are referenced by a parent array, sharing is not very common in CDRS. Therefore, lazy creation of pholes is very important. However, if an application that uses lots of sharing were to use MetaStore, the amount of space occupied by the pholes would affect the system performance noticeably.

Although this "ideal solution" would preserve the semantics of Common Lisp for sharing, supporting it in the context of persistence would be too expensive in a production quality application like CDRS. Therefore, in the next section we present a compromise solution that seems practical enough to use and is implemented in MetaStore.

3.11.3 A Practical Solution

Extra cost associated with the ideal solution described in the previous section forces us to use a compromise solution that seems more pragmatic in terms of performance, although it limits sharing composite values to some degree.

3.11.3.1 Limitations and Remedies

Supporting persistence of any composite value that owns a unique ID is not a problem. Because the smallest grain size of persistence in MetaStore is a composite slot value, each persistable composite slot has a unique ID. Thus, using the already added unique IDs for composite slots, we can easily support structure sharing at the composite slot level. Therefore, we limit persistable sharable values to objects and persistable composite slot values.

Because any persistable object can freely be shared, any composite value which is not a direct slot value can be made sharable by wrapping it as an object. Thus, the array a2 in Figure 3.6 can be made sharable by making it a slot value of an instance of the sharable composite class, say sharable-composite-class:

```
(defclass sharable-composite-class ()
  ((sharable-composite :initform nil
                       :initarg :sharable-composite
                       :accessor sharable-composite))
  (:metaclass persistent-metaclass))
```

With this class defined, a2 now can be replaced by an object created by calling,

```
(make-instance 'sharable-composite-class :sharable-composite a2)
```

Using the accessor sharable-composite and its dual (setf sharable-composite), one can conveniently perform read and write accesses respectively. The cost of using this class is one extra slot access on both read and write accesses. However, this would give much better system performance than the "ideal solution" presented in the previous section. Because the amount of sharing done in an application like CDRS was so small, this is considered an acceptable alternative.

3.12 Virtual Object Memory

The Tao is an empty vessel;
it is used, but never filled.
Lao Tsu, Tao Te Ching

It is analogous to the virtual memory concept. An object resides in virtual memory because it is either created anew or loaded from the object base. Objects in virtual memory will live there until they are deleted. As users create more objects or more objects are loaded due to lazy loading, sooner or later the virtual memory of the program will be filled and reach a point where the system performance is not acceptable. When the system is garbage collection based, as is with Common Lisp, it occasionally reaches a point where a dynamic garbage collection can not even be

performed, resulting in a system crash. For example, this problem has happened many times with CDRS.

The *virtual object memory* idea is to handle this problem. As the number of objects or the amount of data increases in virtual memory, MetaStore monitors the objects in virtual memory and flushes some of them out to the object base. When an object is *flush*ed out, it becomes a *husk.* At some later time, if the object is accessed again, the accessed slot will be instantiated again as usual. Because on the average about 80% of an object is occupied by composite slot values based on our experience with CDRS, flushing out some of the objects that will not be accessed for some time will improve the system performance. This was one of the reasons why we decided to control the persistence at the composite slot level.

In MetaStore, we chose to flush out an object and keep the husk rather than letting the whole object go. Because objects can be shared by variables, other objects, etc., it is important to keep at least the husk portions of objects when they are flushed out. Suppose we did not. That is, MetaStore throws away an object, say `O1`, which is referenced by a user variable, say `v1`. Later, if `O1` is requested again by a user program, `O1` will be loaded again because `O1` does not exist as far as MetaStore is concerned. However, this new `O1` is different from the one `v1` is still holding on to, thus resulting in two different copies of `O1`. We could solve this problem by keeping one level of indirection on objects, but it is not as efficient as keeping husks around.

For this scheme to work properly, we restrict on how a composite slot value is used. When a composite slot value is obtained, a user program can only store it in its short lived local environment, e.g., a `let` binding, or in a slot of an object, but nowhere else. Suppose it was stored in a global variable, say `g1`, and subsequently the object is flushed. If the same slot is read accessed and the newly loaded value is stored to another global variable, say `g2`, then there are now two different copies

of the slot value: one held by g1 and the other by g2. Short lived environments are fine because the lifetime of a local environment will not outlive the time taken by flushing and a reaccess because an object is never flushed unless it has not been accessed for a long time. When a slot value is returned, it is usually stored in a local environment in practice. Storing it in a global variable can always be omitted because the object itself can be considered a global environment. Storing the value in a composite value can always be done by wrapping it as an object as was done for structure sharing. Storing a returned composite slot value in an object is also fine because of the way MetaStore supports structure sharing with pholes.

How do we decide how many objects to keep in virtual memory at any given time? How do we decide which objects to flush out? We answer these questions in the following subsections.

3.12.1 High Water Mark

MetaStore maintains a programmable *high water mark*, which shows how many objects are to be kept in virtual memory at any given time. Because it is hard and expensive to keep track of the amount of data in virtual memory in terms of, say, the number of bytes or words, MetaStore uses the number of objects as a gauge.

The number is obtained initially from the experience of using an application like CDRS, and later by tuning with different numbers. Given an application and a machine configuration, the factors that affect this number would include: the average object size, the typical size of a design model, the amount of available swap space, the size of physical memory, the size of dynamic area available for garbage collection assuming the system is garbage collection based, the efficiency of this algorithm, etc.

Although it would be ideal to find an optimal number, it would be almost impossible to do so. Thus, we should aim at finding a rather conservative number.

Because it is impossible to tune an application to find a good number without an application fully ported to MetaStore, we delay this part of research until one such system is available. CDRS will soon be ported to MetaStore.

3.12.2 Least Recently Used Flushing Policy

MetaStore decides which objects to flush out using the *least recently used* (LRU) flushing policy. The primary purpose of the virtual object memory is to free up some virtual memory space occupied by objects that has least chance of being accessed again. Therefore, it seems most appropriate to free up the space occupied by objects that have not been accessed the longest.

When an object is accessed, read or written, the order of objects being accessed is maintained by MetaStore. This is done by intercepting `slot-value` and `(setf slot-value)` via the metaobject protocol. Along with the high water mark, MetaStore monitors the number of objects in virtual memory using an abstract data type, *high water watch,* which supports the following operations:

- *hww-init:* Initializes it to an empty list.

- *hww-add:* Given an object, if it is already in the list, it is moved to the end of the list. If it is not, it is added to the end of the list. If the length of the list is longer than the `high-water-mark`, then the system flushes all the objects beyond the value of `high-water-mark` at the front of the list. We have an option of flushing out one or some number of objects at a time whenever the number of objects in virtual memory reaches `high-water-mark`. We decided to flush out only clean objects for a few reasons: (i) It will be rare to find a dirty object at the front of the list because of the incremental savings done frequently and because there are in general many objects loaded into virtual memory at any given time. When the number is small, there is no reason to

worry because it will be much smaller than the value of `high-water-mark`, the minimum of which is set sufficiently large. (ii) It makes other algorithms such as checkpointing much simpler. An already flushed object, a husk, is never attempted for flushing.

- *hww-remove:* Removes an object from the front of the list and flushes it.

- *hww-delete:* When an object is deleted, it is deleted from the data structure used by the virtual object memory. This is different from *hww-remove* in that this does not worry about flushing the object.

3.12.3 Feasibility

It is not unusual to have about 20,000 to 40,000 objects in a typical design model, say of a car. The concern is the fact that this long list may have to be traversed rather frequently. With a list implementation, the efficiency analysis for each operation of `high-water-watch` would be as follows:

- *hww-init:* $O(1)$.

- *hww-add:* To optimize the add operation, MetaStore keeps a small buffer of most recently accessed objects.[10] The size `hww-buffer-size` of the buffer is programmable, too. This optimization[11] improves the add operation substantially, especially when the list becomes large. With this buffer, if the object being added, i.e., the object just accessed, is in the buffer, the adding operation

[10]Another less efficient alternative would be to maintain the `tail-pointer` to the end of the list. This is to optimize the case of repeatedly accessing the same object, which tends to happen a lot within a method, the object being added is first checked against the last one in the list. This still happens with the implemented buffer algorithm. With this optimization, repeated accesses to the same object run with $O(1)$ efficiency.

[11]This idea is used in many areas of software systems. To name a few, caching in paging algorithm and ephemeral garbage collection are implementations of this idea in different disguises.

is skipped altogether. Because `high-water-mark` is always substantially larger than the buffer size,[12] this approximation would not affect the correctness of order in the list measurably. A typical operation in a design environment such as CDRS touches only a small number of objects and a number slightly larger than this number would be a good candidate as the size of the buffer. It is currently set at 20.

However, it still requires an $O(n)$ search to see if the object is already in the list or not, and to delete it from the list if it is in the list. Rather than blindly performing a delete operation, it only deletes one if the object is actually in the list. That way, this $O(n)$ search to delete is performed only when necessary. This is made possible by keeping an extra slot, `in-hww-p`, in each persistable object. This slot is updated at the time an object is accessed by either `slot-value` for a read or `(setf slot-value)` for a write.

Checking the length of the list also takes $O(n)$ and it has to be checked on every call to *hww-add* in a naive algorithm. In our design, we keep the length in a variable and update it as *hww-add, hww-remove,* and *hww-delete* are performed. Thus, computing the length of the list is $O(1)$.

Thus, this operation overall is $O(n)$, but quite efficient one because of the optimizations done.

- *hww-remove:* Theoretically, it is $O(n)$ because of the possibility that the first one in the list is a dirty one. In practice, however, it is $O(1)$ since the chance of finding a dirty one at the head of the list is almost nil. When we do find a dirty one at the head of the list, we relocate it to the end of the list as if it

[12]MetaStore prohibits a user program from setting the buffer size to a small number. That is, there is a default minimum set by MetaStore.

was just accessed. That way, we will not find the same dirty one at the head of the list continuously affecting the performance.

- *hww-delete:* It is $O(n)$ because the one being deleted could be anywhere in the list. This operation is not performed very frequently and `in-hww-p` is used when done.

Another possibility would be to keep objects for the high water watch in a sorted binary tree using the time an object is accessed as the sort key. This would give an $O(\log n)$ search for each object. It is doable since a simulated time in the form of an integer can be used, or universal time values are available in Common Lisp, but getting the time and maintaining a sorted binary tree may outweigh the benefit of faster search time.

Another possibility would be to keep time, but sort the objects only when asked. Thus, flushing happens only when a user program requests. If solutions, that are transparent to user programs, are impractically slow, then this would be an alternative.

3.13 Portability

Strange is your language and I have no decoder ...
Peter Gabriel

MetaStore is designed to be portable because both *Meta* and *Store* are. It is currently implemented using Portable Common Loops (PCL) [7] with Lucid's Common Lisp 4.0 [30]. The current implementation was based on the CLOS Specification [8,46] and the metaobject protocol described in [23]. Therefore, MetaStore is portable as long as the CLOS implementation is compatible to these. A few deficiencies in the current specification of the metaobject protocol of CLOS are found in extending CLOS with object persistence thus making the current implementation of MetaStore less portable, and they are discussed in section 5.1.

PCL was chosen because Lucid's CLOS version 4.0 was not complete to the CLOS specification.

As far as Common Lisp is concerned, MetaStore is portable as long as the implementation of Common Lisp is compatible to the specification in [46].

4

Persistent Object Store

Oh, hidden so deep yet ever present!
Lao Tsu, Tao Te Ching

Store in MetaStore is a persistent object store that provides a database management system for object-oriented languages that support persistence. It is general purpose and language independent. The current implementation supports languages that can interface with Common Lisp [46] programs. In this chapter we describe the essential and performance enhancing features of *Store*.

We begin in section 4.1 by describing the architecture of the object base in MetaStore. It consists of two levels: model files and objects within model files. Objects are stored to and retrieved from the object base via queries. In section 4.2 we describe three different ways of querying the object base. To enhance the loading time on queries, we support a few indexing schemes, which are described in section 4.3.

Because the object base can be shared by multiple processes, concurrency control with atomic updates is critical in maintaining the consistency and integrity of the object base. A crash recovery mechanism must also be provided so that the object base can be recovered to a consistent state on a crash. These issues are described in section 4.4.

Objects in the object base evolve over time and it is important to maintain different versions of an object. We describe a version control scheme in section 4.5.

Because of the shadow paging algorithm used to support crash recovery, atomic updates, and version control, a nontrivial amount of garbage exists in model files. An algorithm for collecting the garbage is described in section 4.6.

As a means to enhance performance on loading objects, indexing schemes are described in section 4.3. Another way of enhancing performance is to cluster objects in model files, thus reducing disk seek time on loading. We describe several clustering algorithms in section 4.7.

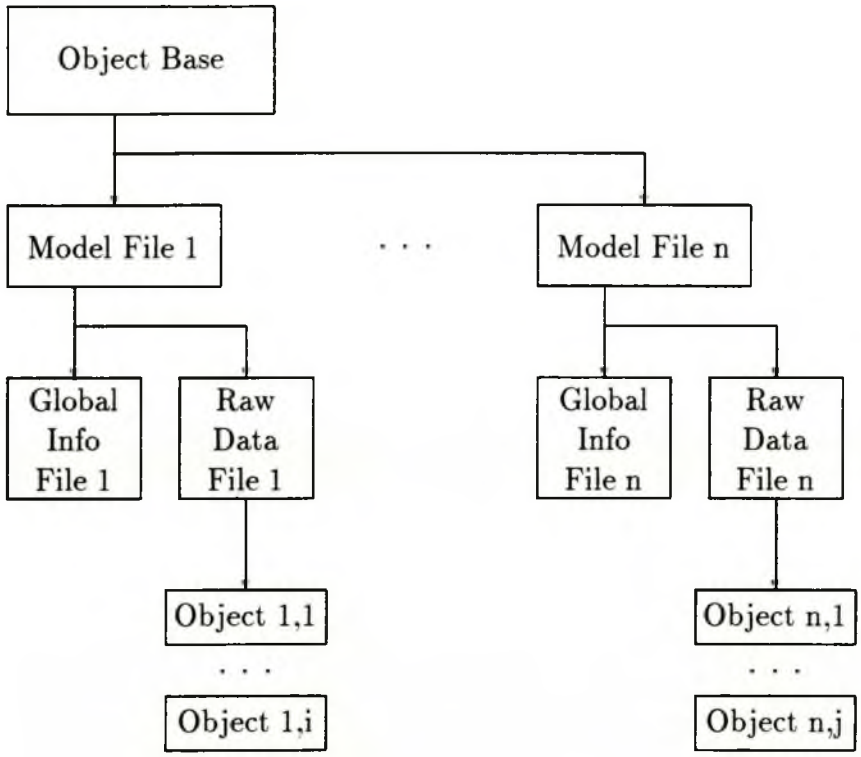
An application program interfacing the object base could use multiple model files at a time, possibly merging them into one. An algorithm for dealing with multiple open model files is described in section 4.8.

4.1 Architecture of Object Base

The object base in MetaStore has two levels as shown in Figure 4.1. It consists of one or more model files. A model file is represented by two files: one containing the global information about the model (indexes, keys, versions, structure sharing, etc.), and the other containing the raw data for each object. A model file in the object base of MetaStore can be regarded as a micro database as each model file is similar to a flat structured database.[1] In the current implementation of *Store*, each model file can have up to 2^{29} arbitrarily large objects and composite slot values. The object base can have 2^{29} such model files. Each model file usually contains one or more design objects, e.g., a door or hood of a car. Each model file is also self-contained, thus objects cannot be shared among model files. The reasons for not allowing objects being shared among files are detailed in section 4.8.3.

Model files seem to be the right granularity for partitioning the object base. The user's perception of an "object" in a design environment is a real world "object," not an instance of a class. A user's "object" would be represented by a collection

[1] An object base can have a flat structure, where the entire object base is a collection of objects.



Note: Each arrow in this figure means "consists of," thus an object base consists of n model files.

Figure 4.1. Architecture of an object base

of networks. A model file models a user "object." For example, a model file could contain a wheel or a door of a car. With this architecture, a typical design session would open one or more model files for editing, thus the object base interface of a user would be quite localized. Because of the localized access and the lack of demand for concurrent accesses to a model file, we adopt a simple transaction management scheme as we will see in section 4.4.

Store keeps track of all the model files making up the object base, and supports a set of interface routines for application programs to use in modifying the object base.

4.2 Queries

In a design environment that MetaStore supports, objects are loaded from the object base and modified in virtual memory. Modified objects are then saved back to the object base. The smallest unit of an update operation thus is a composite slot value. With this design a user program only needs the interface of loading and saving objects and composite slot values. Any modification is done by using the host programming language in use. In this regard, the interface to the object base in MetaStore is simpler than a general purpose database system, which normally supports a general purpose query language. The object-oriented database system Orion [5], for example, provides its own custom query language, the syntax of which is very similar to Smalltalk [16]. Orion also supports query optimization by automatically determining an optimal path to execute a given query. GemStone supports an object-oriented language called OPAL [33] which is also used as GemStone's query language.

In MetaStore objects are loaded from the object base by one of three ways:

- *Preloading:* Some objects are preloaded at the time a model file is opened for editing partly because we want to reduce the time spent on queries by user program or on lazy loading, and partly because some objects must be loaded to display what is in a design so that a user can see even before he can start editing the model (section 4.3.1).

- *Lazy loading:* On a read access to a persistable composite slot of an object, the value of the slot is loaded if not already. This kind of loading happens transparent to the users.

- *User program queries:* MetaStore supports three different ways of querying the object base, which we will see in detail below.

Objects in MetaStore are saved by one of three ways:

- *Save an object request:* A user program can request for an object to be saved. MetaStore treats the given object as a network by saving the network rooted at the given object.

- *Checkpoint request:* On a checkpoint request, all the modified objects are saved (section 4.4).

- *Version request:* On a version request, all the modified objects are saved along with the new version information (section 4.5).

In this section, we focus on the user program queries. An application program can request for a retrieval of an object by one of three ways that *Store* supports: query by ID, query by name, and query by key.

4.2.1 Query by Object ID

Each object has a persistent ID, unique within the object base, via which an object can be retrieved.

4.2.2 Query by Object Name

A user program can assign a name to an object. A name for an object is unique within the object base. A name is represented as a pair: a model file name and a name unique within a model file. Once an object is named, the name can be used to retrieve the object. For users of the system implemented on top of MetaStore, a meaningful name such as `door` instead of an ID such as 22 is much easier to deal with. Either MetaStore or the application program could map user's meaningful names to IDs for retrieval of objects from the object base. We support the mapping in *Store*.

4.2.3 Query by Object Key

In an object-oriented design systems, there are two categories of objects: top level objects and subordinate objects. A subordinate object without its top level object, i.e., owner, is useless. Thus, there is no use loading a subordinate object without its owner. For example, a top level object `curve` has a subordinate object `math-curve` in CDRS. If a student is represented as an object and the student's address is also represented as another object, the student object would own the address object.

Store provides a way for a user program to declare *keyed classes*. The store then creates an index for each object of a keyed class for fast retrievals. Once a keyed object is loaded, its composite slot values are lazily loaded when read accessed. Thus, keyed objects in *Store* are top level objects. A subordinate object of a keyed object would be a part of a composite slot value of the keyed object. From our experience with CDRS, only about 4% of all the objects in a model file are keyed objects.

When retrieving a keyed object, a user program can optionally supply a set of slot name/slot value pairs. The store then retrieves only the objects that meet the constraints supplied by the user program. This way, only a subset of all the objects of a keyed class can selectively be retrieved. For efficiency reasons, we restrict the user supplied slot name/slot value pairs to be those of atomic slots.

4.3 Index Management

answer me speedily
Job 31: 37

Indexes are used to speed up object retrievals from the object base. In Orion [25] a single-class index is defined to be an index that is maintained on an attribute of a single class, and a class-hierarchy index to be an index on an attribute of all classes of a class hierarchy rooted at a particular class. Orion supports only single-class

indexing. In GemStone [32] collection objects such as sets and bags may be indexed with values of slots being used as keys. It supports two schemes: indentity indexing and equality indexing.

Store supports a simple indexing scheme different from both Orion and GemStone. In *Store* indexes are maintained on named objects and keyed objects in an index table, which maps a persistent ID to a location of a model file. When a named object or a set of keyed objects is requested, the store retrieves them efficiently using this table.

Named objects are simple to handle with a name table and an index table because the name table maps a name to a persistent ID.

Retrieving keyed objects selectively using slot name/slot value pairs as described in section 4.2.3 may be quite inefficient as the number of keyed objects becomes large. There are at least two different ways to handle this difficulty. One is by preloading keyed objects and the other by a more elaborate indexing scheme similar to the single-class index scheme of Orion. We now describe both schemes and explain why we chose the preloading scheme in *Store*.

4.3.1 Preloading Objects

In the preloading scheme, keyed objects are preloaded at the time a model file is opened for use. Given objects of a preloadable class, a user program can even specify what portions of the objects are to be used. Depending on what kind of session is intended, different preloading schemes can be executed. For example, opening a file to edit a model by modifying curves and surfaces in CDRS requires quite a different set of slots to be instantiated than opening a model file to view with photorealistic rendering.

Named objects can also be preloaded. An application program may in some cases want to preload nonkeyed, unnamed objects, i.e., subordinate objects described in

section 4.2.3. It can do so by calling a function that defines a preloadable class, whose instances would then be preloaded. A class defined to be preloadable by this call is different from a keyed class in that no index is created for a class that is defined as preloadable.

All preloadable objects, including their persistent composite slot values, are separated at the top of the model file so that they can be sequentially read in when the model file is opened. Each design application would have a small number of *modes* of preloading. In CDRS, for example, we would use only two modes: modeling and rendering. When a model file is opened for a session, it can be opened for one or a combination of the defined modes. When the number of modes becomes large, we must handle complicated semantics of partitioning a model file into preloadable sections. There would be a section allocated for each mode in a model file in addition to a section that contains portions of objects that are common to all modes. For example, a model file in CDRS would have four sections:

- A common section that is preloaded in any editing session
- A section that is loaded only for a modeling session
- A section that is loaded only for a rendering session
- A section that will be lazily loaded.

Clustering objects and composite slot values into different sections in a model file is done with the modes defined by a user program while the model file is garbage collected. Detailed description on how mode information is used in clustering preloadable objects is deferred until section 4.7.1.

The amount of preloading an application chooses to do has a direct effect on the amount of lazy loading that will be done. The goal is to maximize preloading without compromising system performance too much, so that lazy loading can be minimized.

4.3.2 Elaborate Indexing

This scheme, which is similar to the single-class indexing scheme of Orion [25], would extend the idea of keyed classes to slots. A user specifies slot name/slot value pairs at the time a keyed class is defined. For example, a user program can tell *Store* the following:

```
(define-keyed-class 'curve 'type '(:selected :general))
```

to mean that indexes are to be created for the slot name `type` for two possible slot values: `:selected` and `:general` for the class named `curve`. With this definition, all the curves whose `type` is `:selected` or `:general` can efficiently be retrieved.

If many slot name/slot value pairs are required for an application, supporting this scheme would add much overhead in index management, not in the complexity of algorithm but in the number of indexes that must be maintained. Based on our experience with CDRS, only a few pairs would be sufficient. CDRS, for example, uses only one pair. Even with other CAD systems, the number of pairs would in general be quite small.

Implementing this scheme would add extra overhead to object creation and write accesses to objects. Slot value modifications must be monitored for changes such as from `:selected` type to `:general` of a `curve` in CDRS. This scheme is easy to support via metaobject protocols, but adds extra cost to each object creation and slot access.

4.3.3 MetaStore's Choice

We chose to use the "Preloading" scheme instead of the "Elaborate Indexing" scheme in *Store* because of the following efficiency concerns:

- Based on our experience with CDRS, a user would be more willing to wait a few extra minutes due to preloading at the time a model file is opened than

putting up with excessive lazy loading on the top level keyed objects on the fly as he is using the system. A well-tuned preloading will make the system much more efficient.

- The "Elaborate Indexing" scheme adds a constant overhead to each object creation and slot access although the performance of an object-oriented application depends heavily on the speed of object creations and slot accesses.

4.4 Transaction Management

many shall run to and fro
Daniel 12: 4

Simultaneous, conflicting modifications can occur when multiple concurrent transactions execute in an uncontrolled manner. Therefore, concurrency control is necessary in order to maintain database consistency and integrity. In section 4.4.1 we describe how atomic updates on transactions are achieved with the concurrency control scheme adopted in *Store*. Recovering from a crash is described in section 4.4.2.

4.4.1 Concurrency Control

In *Store* the granularity of concurrency control is set at the model file level. Both Orion [24] and GemStone [33] support more elaborate mechanisms for concurrency control than the one we support in *Store*. Mneme [37], although not implemented yet, also has a more elaborate design than *Store* for concurrency control. Concurrency control in these other systems are more elaborate in the sense that they use a smaller grain size, namely an object. We chose the simpler scheme in *Store* since:

- Each model file is really an "object" (a real world "object" such as a `hood` of a `car`) to the end user's perception. A user "object" consists of one or more networks. There are usually rich relationships among objects within a

network. All the objects making up a network must be saved successfully in a save transaction. Therefore, the object level concurrency control scheme may be not only an overkill but also complicated to support for design applications. Based on our experience with CDRS, we did not see any real reason to support concurrency control at the object level. For instance, a `point`, a `curve`, or a `surface` would not mean much to be retrieved alone with a write lock based on the reasons given for the architecture of the object base for *Store* in section 4.1.

- Each model file is kept self-contained (section 4.8).

Store uses the standard transaction model with read- and write-locks [52], where locking is done on a model file. With this model multiple read-locks can be applied to a model file, whereas no other lock, read or write, can be applied to a model file if a write-lock is already applied to the file. In *Store* an update transaction would:

1. obtain a write-lock on a model file,
2. preload all the appropriate objects that are meant to be preloaded (section 4.3.1) after loading the object table information,
3. modify some objects,
4. save the modified objects followed by the save of the object table,
5. and finally release the write-lock.

This would repeat until a user session is done. Because step 2 above would in general take a long time, we modify this transaction model somewhat as follows for efficiency reasons. We augment the model with the nested transaction idea [6]. With this augmentation, a process would:

1. obtain a write-lock on a model file at the start of a session (We call this lock a checkout-lock.),

2. perform a series of nested transactions,

3. and finally release the write-lock (checkout-lock) when the session is finished.

Each nested transaction would simply perform an update transaction without having to worry about locks at all. Let us call the top level transaction that obtains a checkout-lock a checkout transaction. Then, a checkout transaction obtains and releases locks, whereas nested transactions are the ones that update a model file in the object base. These nested transactions are important because a crash recovery can be made to the start of a nested transaction rather than to the beginning of a checkout transaction as we will see in the next section.

A model file is modified by a series of atomic transformations of the model file from one consistent state to another, where each transformation is done by a safe update transaction. A transaction is said to be *safe* if it transforms the object base, i.e., a model file, from one consistent state to another via an atomic update without a crash.

An atomic update in *Store* is achieved by using the shadow paging algorithm [51]. With this algorithm when a write of an object is to be done, it is done to a different place than the place where the original value of the object is located. Both PS-Algol [13] and GemStone [33] also use shadow paging for their atomic updates.

When a modified object in a network for an update transaction is ready to be saved, it is written to the end of the model file that is currently checkout-locked. It would be ideal to write a modified version of loaded objects back to the same location where the originals are located. It is in general not easy to do so because a newer version of an object may be different in size from the original. We chose to write objects at the end of the model file in *Store* because we want to (i) support

multiple versions (section 4.5) of objects in a model file, and (ii) be able to recover back to a coherent and consistent state of the object base in cases of system failures or programming errors.

Each nested transaction (incremental save) would involve one or more objects. As each object[2] is saved, the *phole*, associated with the object, in the object table remembers the new location in the file where the new write is done. After all the dirty objects participating in an incremental save are saved with the corresponding pholes updated along the way, the object table is written out to a temporary file.[3] If the object table is successfully written out to the temporary file, then it is renamed to the object table file, of the currently checkout-locked model, that the object base knows about. Thus, the atomic update of an incremental save depends on this last step of renaming the temporary object table file, the "commit" step. If the system crashes during the commit step thus not correctly renaming the temporary file, we would lose the editing that has been done since the last such commit.[4] If the system crashes before the commit, then all the write accesses done to the raw data portion of the model file are not reachable from the saved object table and become garbage, which will later be collected (section 4.6). In GemStone collecting garbage generated by failed transactions seems a major overhead because the entire database must be collected [33], whereas collecting garbage in *Store* is a trivial task since only the model files that are currently open need to be treated and each model file can be garbage collected in isolation due to the "self-sufficient" property of each

---

[2]A composite slot is handled similarly and we only describe objects for the simplicity of describing the algorithm.

[3]One possible improvement to this design would be saving the object table incrementally.

[4]In fact, we could augment our algorithm so that after each such crash, we can check to see if there is a such temporary object table file. If so, go ahead and rename the temporary file. We did not include this into the initial design since (i) The chance of this happening is very slim. (ii) The temporary file may not be complete either, which is not easy to check. Even if a crash happens, the stake is not very high because of frequent incremental saves.

model file in *Store*.

One disadvantage of the shadow paging algorithm is that it in general generates nontrivial amount of garbage in a model file, thus requiring object base garbage collection. Although this is a nuisance, we decided that supporting version control and being able to recover from system failures or programming errors far outweigh being a nuisance. Even without garbage collections, objects and composite slot values in a model file are rearranged for object and slot clustering anyway (section 4.7). Thus, combining these two seems reasonable especially because they are done during off-hours in batch mode.

4.4.2 Crash Recovery

The most common tool for protecting against loss of data in the face of system failures is the log or journal, which is a history of all the changes made to the database, and the status of each transaction. Although the resilient protocol via the "Redo" recovery algorithm [52] is a common one used in many database systems, we decided to use a simple log based crash recovery protocol via the "checkpointing" algorithm in *Store*. Thus, the system using *Store* would be able to recover to some coherent and consistent state of objects if the system crashes.

While using the system, a user can request checkpoints along the way. A checkpoint is a consistent state remembered of a model file so that it can be recovered if necessary due to a system crash or a user request to support features such as an *undo* operation.

A checkpoint in *Store* is a snapshot of a model file at a given time. When a checkpoint is created, an incremental save of all the modified objects is done as described in section 3.10. When a dirty object is saved, the previous location of the object in the model file is remembered. Thus, a snapshot is a list of logical ID/file position pairs corresponding to the objects and composite slots which have been

modified since the last checkpoint. When two checkpoints are made along the way, say `c1` at time `t1` and `c2` at time `t2` in that order, the checkpoint `c2` is basically remembering the state of the model file at time `t1`.

Checkpoints are done incrementally to save some space, although it would require extra processing time to restore an old checkpoint. Suppose four consecutive checkpoints, say `c1`, `c2`, `c3`, and `c4`, have been made. Now, if we want to go back to the checkpoint `c2`, then we must go through `c4` and `c3` in that order to get to `c2`.

It is algorithmically easy to support an arbitrary number of checkpoints. However, for practical reasons, the limit is set to a small number that can be configured by a user program.

The operations supported by *Store* dealing with checkpoints are as follows:

- *Create a checkpoint:* A new checkpoint with an optional name can be created. If a name is passed, it will be remembered and returned at the end of this operation. Otherwise, a *Store* generated name is used and returned. In any case, a name is guaranteed to be unique within a model file.

- *Goto a checkpoint:* A previously defined checkpoint can be restored given a name. Going back to the most recent one can be done easily by a special syntax.

- *Delete a checkpoint:* A checkpoint can be deleted given a name. Deleting the most recent one is simple. However, deleting one in the middle of the checkpoint history is a little tricky. If four checkpoints have been created, say `c1`, `c2`, `c3`, and `c4` in that order, and `c2` is to be deleted, then `c3` is replaced by the union of `c2` and `c3`. If there is a common entry in both checkpoints, then the one in the earlier one, `c2` in this case, is kept.

When multiple models are merged, checkpoints are always applied to the base model (section 4.8). This is consistent with how merged models are handled since only the base model is write accessible.

Checkpoints are not persistent, i.e., not saved in a model file because they are perceived to be temporary. In comparison, versions as described in the next section are meant to be permanent and are saved to disk.

4.5 Version Control

Version management is important for engineering databases because design is often an experimental process, the scope of which changes with time. It is necessary to keep track of the evolution of design objects, and the changes made to a design by various transactions. It is also important for concurrent cooperative work because different clients may work simultaneously on different versions of the same object, rather than wait for each others' transaction to complete.

The granularity of version control can be done either at the model file level or at the object level. Both Orion [10] and ObServer [55], for example, use the object level version management, but selectively. That is, a versionable object is an instance of a versionable class or that of a subclass thereof.

The model file level scheme is algorithmically simpler and semantically easier than the object level scheme. The key difference to note is the degree of involvement on the part of end users. With the grain size at the object level, a user has a finer control over objects' versions. It is not without its cost though. A user must constantly worry about which objects are at what version level and which others are at what other level, and so on. Because a model file is a good grain size of partitioning the object base as described in section 4.1 and each model file would contain the design of a real world object, version control at the model file level is more meaningful than one at the object level for the end users. If, on the other

hand, the object base is flatly structured with objects as direct targets of queries, then supporting version control at the object level would be more appropriate.

In *Store* we chose the model file level scheme, in which a version is an attribute of a model file. When a new version is created, each object or composite slot value in a model file acquires a new version. Each version contains two pieces of information: `version-name` and `file-position`. Thus, given a version name of an object, the given version of the object can be retrieved using the associated file position.

Different versions of an object could very commonly share the same file position because only a small subset of all the objects in an editing session would be dirty when a version is created. The only time two different version names of an object get different file positions is when the object is modified and saved between those times when those two versions are created.

A naive implementation would have each version name be associated with a file position. Then, there would be many file positions shared by multiple versions of an object. Let us assume for example that the following is the version information contained in the phole for an object `O1`:

```
((v1 . fp1) (v2 . fp2) (v3 . fp2) (v4 . fp3)),
```

where the file position of a version `v1` is `fp1` and that of `v2` is `fp2`, etc. The file position `fp2` is shared by both versions `v2` and `v3`. To eliminate the space occupied by the duplicate file positions between different versions of an object, we allow two formats for a version: a version name/file position pair or a version name alone. With this optimization, the same version information for the object `O1` would now look like:

```
((v1 . fp1) (v2 . fp2) v3 (v4 . fp3)).
```

Although only one integer value for each of some selective pairs is eliminated, this optimization may actually save quite a lot of space. The creation of each new

version would add one integer to each phole in the entire model file. If a user chooses to create versions often with only a few objects modified, then conceivably a lot of space overhead will be incurred. To make this optimization work, the list of pairs in a version is kept ordered. Thus, operations such as deleting a version or adding a new version must respect the order in the list. Although this optimization adds some overhead in processing time, it is considered an important optimization for space efficiency, both in virtual memory and in model files in disk.

4.5.1 Operations

Store provides the following interface for version control in the model file level scheme with the given semantics.

- *Opening a model file:* When opening a model file, a version name can optionally be specified to use a specific version. If one is not given, the latest version is the default. Preloading of keyed objects (section 4.3.1) factors this into consideration by appropriately ignoring unnecessary versions.

- *Create a version:* A new version can be created by adding the new version information to each phole in the model file and by saving all dirty objects.

- *Select a version:* A different version can be selected to be the currently active one. This operation would visit each phole in the model file and set the value `value` of each phole to `NIL` if appropriate, also taking care of the shared file positions. The next read access to a nullified phole would then retrieve a new value for the phole.

- *Delete a version:* A version from an open model file can be deleted. This is handled in two steps: For each phole in the open model file, (i) nullify the value `value` of a phole if necessary, also taking care of the shared file positions,

and (ii) remove a version name/file position pair again also taking care of the shared file positions.

- *Extract a version:* A version of the open model file can be extracted into a separate model file. This is done by creating a new model file whose name is specified by the user. With this operation, a user is given the option of keeping the version being extracted in the original model file or not.

4.6 Object Base Garbage Collection

Because of the shadow paging algorithm used, there is in general a nontrivial amount of garbage contained in a model file. The pattern of use that creates garbage the most is one where a user loads objects and repeatedly modifies and saves them.

The garbage collection algorithm used here is basically that of *stop and copy* [1]. As described in section 4.1, each model file internally consists of two files: one for the raw data and the other for the global information including the object table. Given a model file, therefore, the object table is used as the root for this garbage collection algorithm. All the versions of all the objects and composite slot values in the raw data portion of a model file that are reachable from the root are copied into a temporary file. When the copying is finished, the temporary file now containing only "good" raw data is renamed back to the original file name in the object base. This renaming is the step that guarantees an atomic update of the entire garbage collection algorithm.

Garbage collection is done during off hours in batch mode. The object base keeps track of model files that have not been garbage collected so that they can be collected at some off hours designated by the system manager. If a model file, which has not been garbage collected, is requested for editing, the user will be warned, at

which time the user can decide to use the uncollected file with the risk of slower loading time or ask it to be opened after being collected first.

During a garbage collection, objects and composite slot values would in general be reorganized dictated by the order of pholes in the object table. Object and slot clustering is also performed during a garbage collection and we now describe the clustering scheme for *Store* in the next section.

4.7 Object Clustering

To reduce the disk seek time during the loading of objects and composite slot values, they are kept clustered in the model file. Different clustering algorithms give different benefits. An optimal clustering for one application could be drastically bad for another, so clustering almost always involves tradeoffs between different modes of use for the same collection of objects. Because *Store* is general purpose for different kinds of applications, it is important to provide means for application programs to use to configure different modes of clustering.

The term "clustering" in this context refers to how closely objects are located to one another in model files for efficient retrievals. *Store* does not concern itself with dynamic clustering at the save time because of the perceived overhead of doing so. It instead applies the clustering algorithm to a model file in batch mode. It would incur a tremendous overhead to cluster objects being saved with the objects that are already in disk dynamically during saves. Furthermore, saved objects are not likely to be loaded again in the same session unless they are flushed out by the virtual object memory algorithm (section 3.12) because objects still remain in virtual memory even after they are saved. It would be better to increase the *high water mark* a little higher to reduce lazy loading of unclustered objects than worrying about clustering during the save time. *Store,* therefore, currently supports static clustering rather than dynamic clustering.

Because of "preloading" (section 4.3.1) that *Store* uses, we separate clustering into two different kinds: one for objects to be preloaded, and the other for objects to be lazily loaded.

4.7.1 Objects for Preloading

There are two issues to consider in the context of preloading: (i) deciding which objects to preload, and (ii) deciding what portions of an object to preload given an object to be preloaded.

A graphical user interface based application like CDRS requires loading at minimum some subset of slots of geometric objects for viewing. Thus, the slots or objects whose mission is to display an object would typically be preloaded. Based on our experience with CDRS, we thought that it is extremely important to support preloading of a certain collection of objects, i.e., named objects, keyed objects, and objects that are defined to be preloadable by the `define-preload-class` call. These preloadable objects are separated into a specific region of a model file, and are sequentially read in at the time a model file is opened. Sequential reads are more efficient than random reads or clustered reads. It may seem to make sense to cluster objects of the same class together at the object level as does LOOM [20], but it really does not because of the sequential reads.

Each session of an application can potentially use different aspects of objects. In CDRS, for example, a user can start a session for modeling, rendering, or both. Thus, it is important to provide a user configurable means for each mode of use.

We support the following mechanism to satisfy this need. This is similar to the *semantic clustering* idea of [44]. For each persistable class, a user can optionally specify a list of operation mode/slot name pairs. For each session, then, the husks of preloadable objects are loaded along with the appropriate portions of the objects depending on the mode. That is, only some subset of composite slot values of an

object is loaded. The value of a composite slot not mentioned in any specification is always preloaded by default. This way, diligent users get better performance, whereas lazy users get the default performance. This is one way of passing the domain specific knowledge down to a low level tool. Suppose, for example, the following specification is given:

```
(pos-preload-cluster class-name '((:model slot1 slot3 slot5)
                                  (:render slot2 slot6 slot8))),
```

where `slot1`, `slot2`, `slot3`, `slot4`, `slot5`, `slot6`, and `slot8` are composite slots and `slot7` and `slot9` are atomic slots. With this specification, if the model file containing this object is opened for the `:model` mode, then the husk and the values of `slot1`, `slot3`, `slot5`, and `slot4` are preloaded. Note that the value of of the composite slot `slot4` is loaded by default because it is not included in the specification. If the same model file is opened for the `:render` mode on the other hand, then the husk and the values of `slot2`, `slot6`, `slot8`, and `slot4` are preloaded. Again, the value of `slot4` is preloaded by default.

We support a special mode, `:all`, to mean that all the modes defined for an application are to be preloaded, thus overriding all the other modes defined.

The notion of *horizontal* and *vertical* clustering [50] is in reference to the class hierarchy. Thus, clustering objects among which one object is making a reference to another is considered vertical, whereas clustering objects because they all belong to the same class would be considered horizontal. Horizontal clustering is good for the cases where the objects of the same class are to be loaded together as in a graphical user interface based design system that requires displaying what is in a design even before any editing is done. Vertical design is good for the cases where an object is to be loaded together with others that reference it. Therefore, in *Store* we chose to do horizontal clustering on preloadable objects and vertical clustering on lazily loadable objects.

4.7.2 Objects for Lazy Loading

Lazy loading of objects or composite slot values happens because one object references another object or a composite slot value. Thus, clustering of an object with the structured data that are referenced by it would be vertical. There are two issues to consider in this kind of vertical clustering: (i) deciding which objects to lazily load, and (ii) deciding how to cluster objects that are to be loaded lazily. The first is easy: any object that is not preloadable, i.e., unnamed objects, nonkeyed objects, or objects of a class that is not defined preloadable by the `define-preload-class` call, is assumed to be loaded lazily. These are the objects that are reachable from one or more preloadable objects. The second issue is described in the rest of this section.

There are several factors that affect loading: (i) relationships among objects, (ii) relationships among slot values within an object, and (iii) interplay of objects and slots with the object's class. These factors must be handled in concert with the typical access patterns. Adopting the preloading scheme for some objects makes our life much easier. It not only makes clustering algorithms much easier, but also diminishes the impact of the kind of clustering algorithms we use as we see below.

4.7.2.1 Relationships among Objects

Relationships among objects are conceptually vertical in nature because the relationships are almost always established via one object referencing another.

Because the objects we consider here are all intended to be lazily loaded, it would make more sense to cluster them vertically by traversing object relationships starting from a named or keyed object as a root object.[5] We traverse the composite

[5]We do not consider a preloadable object that is defined by the `define-preload-class` call a root object because it is supposed to be a subordinate object which happens to be preloaded for efficiency reasons of loading rather than its relationship with its subordinates.

slots of a named or keyed object depth first, and cluster reachable ones. If we, while traversing, encounter any preloadable object or a nonpreloadable object that is already clustered, then we skip it.

It is very possible that there are nonpreloadable objects that are not reachable from any named or keyed object. We handle them also by clustering them one at a time in the order of appearance in the model file. We do not try to find an optimal clustering by figuring out which of the remaining ones should be considered roots and which subordinates, because there might not be enough information to figure this out. Even if there is, it would be quite inefficient because it requires multiple passes over the pholes in the object table.

4.7.2.2 Relationships among Slot Values

Relationships among slot values given an object are horizontal in nature. Atomic slot values are not of any concern since they are already clustered in the husk given an object. Composite slot values can be clustered by specifying them using the following call:

```
(pos-slot-cluster class-name '((slot1 slot3 slot5)
                               (slot2 slot6 slot8))),
```

where `slot1`, `slot2`, `slot3`, `slot5`, `slot6`, and `slot8` are composite slots. Atomic slots, if included, are ignored. Clustering of an object of this kind results in the layout of components in the following order: husk, each specified slot clusters, and the nonspecified composite slot values as a cluster. If the size of an object is small, say smaller than the I/O buffer size, clustering of this sort is unnecessary and is skipped.

4.7.2.3 Class and Its Instances

This kind of clustering is horizontal in nature, and mostly handled by the "preloading" algorithm. An example would be to load the values of a set of pecific

slots of all the objects of a given class, say to display all the objects of a class. One of the main goals of preloading is to handle requests like this.

4.8 Merging with Multiple Models

We have already considered the case where one process is accessing one model file in the object base in section (4.4.1). In this section we treat the case where one process is accessing multiple model files.

Because the object base in MetaStore is a collection of model files, user programs interface the object base via model files. Typically, a user program would open a model file with some preloading done and start accessing objects thus triggering lazy loading from the opened model file.

There are times when a user will want to edit objects from two or more model files in the same editing session. This is quite similar to accessing multiple flat structured object bases because each model file can be regarded as a micro database as each model file is similar to a flat structured database. This section addresses the issue of dealing with multiple opened model files by describing two possible schemes: copying scheme and sharing scheme. We will then describe why we chose the copying scheme for MetaStore.

4.8.1 Copying Scheme

With two or more model files opened, when an object in one model file is referenced by another object in another, the referenced object as a network is *copied* to the other model file that contains the referrer. However, the actual copying does not happen until it is necessary, i.e., when one of objects in the network rooted at the referrer is saved. Thus, a model file is self-contained and objects in a model file can not be shared by other objects in other model files.

With this scheme, a persistent ID in a model file is represented by an integer unique only to that file, although an implicit file ID is used while a model file is opened for editing.

There is the notion of a base model and merged models in this scheme. The first model opened for editing is by default considered the base model. Any other model opened thereafter is regarded as a merged model. Therefore, there could be multiple merged models in a modeling session although there is only one base model at any time. Merged model files are only read accessible, whereas the base model is read and write accessible.

When a model, say **merged**, is merged to a base model, say **base**, we operate under the following policy:

- Any newly created object will automatically belong to **base**. In general, there is no good way of differentiating which of new objects logically belong to **base** and which others to **merged** unless we require users to specify it.

- Any shadow paging of an object which was loaded from either **base** or **merged** will be done to **base**. Because newly created objects will automatically belong to **base**, performing shadow paging of an object which was loaded from **merged** to **merged** may make **merged** non-self-sufficient.

- As a necessary condition of the commit of a save transaction, all the networks from **merged** involved in the transaction will be copied into **base**.

- The merged model, **merged**, will stay unmodified at the end of the design session. The base model, **base**, will in general change if any new design work is done. Thus, if the user wishes to keep the base model and yet build more and save it as a new model file, he or she should copy the base model before beginning the design work.

- Merging more than one model file works similarly.

The disadvantages of this scheme include: (i) the possibility of excessive copying of objects from one model file to another during a save transaction, and (ii) the loss of object or structure sharing thus causing model files to be larger.

A user usually merges multiple model files in an editing session for:

- *Viewing, i.e., read accesses only:* In this case, this scheme works very well. This is quite common in CDRS, for example, because a user quite frequently opens a number of files to render. Rendering a model file does not affect the model file in the object base.

- *Editing, i.e., read and write accesses:* In this case, it would be more efficient to create a new model file (prior to editing any file) which is a merged version of all the files that are to be edited together as a merged editing. This is especially desirable if the user intends to save them later into a merged file. *Store* supports a routine that merges multiple model files into a new one.

4.8.2 Sharing Scheme

With this scheme, an object in one model file can be shared by any object in the entire object base. Thus, there is no copying necessary. With this scheme, a persistent ID is represented by a pair, a file ID unique within the object base and an object or slot ID unique within a model file. Thus, each persistent ID in a model file is by itself unique across the entire object base.

This scheme also maintains the notion of a base model and merged models. With this scheme, however, both the base model and merged models are read and write accessible. Any loaded object is saved to the same model file where it came from if it gets modified after being loaded. Any newly created object is saved to the base

model, an arbitrary decision because there is no good way of differentiating which new objects logically belong to which of the opened model files.

This general scheme supports a clean semantics of merging and sharing. The disadvantages, however, are:

- Each persistent ID in a model file must be a pair instead of a single integer.
- Each model file is not self contained. Thus, the tools that deal with model files such as deleting a model file must make sure that there are no dangling references in a model file.
- This makes the object clustering algorithm within model files unmanageably complex.

4.8.3 MetaStore's Choice

In MetaStore, we chose to adopt the "Copying Scheme" because:

- It is simpler to implement than the "Sharing Scheme."
- It provides simpler semantics for users.
- Garbage collections are not localized in the "Sharing Scheme," thus more complicated and more time consuming.
- Clustering of some objects is impossible to do in "Sharing Scheme" because they exist in different files.

5

Implementation Results

Theorists need not bother:
The European Common Market already has
a glut of butter, milk, wine, and theorems.
Andy Tanenbaum

One of the main goals of this research was the investigation of metaprogramming as a language design tool. We first report our experience of using the metaobject protocol of CLOS in extending it with object persistence in section 5.1. We also propose some improvements to the metaobject protocol of CLOS.

MetaStore was motivated by and intended to be used by CDRS, which is implemented in Lucid CLOS. Therefore, Lucid CLOS was the intended implementation language for MetaStore. It is however implemented in PCL instead because the current implementation of Lucid CLOS is not complete enough to support the persistence granularity at the slot level, whereas PCL is. Because all benchmark results for MetaStore are done in PCL, we present a performance comparison between PCL and Lucid CLOS in section 5.2 so that we can project the expected performance of MetaStore implemented in Lucid CLOS for the benefit of CDRS.

Although object-oriented programming has advantages over conventional programming, there is extra cost associated with objects. In section 5.3 we present the cost of objects compared with some traditional data structures such as structures, arrays, and lists. The comparison is done because: (i) we wanted to use the most efficient data structures in implementing MetaStore because we were very performance conscious, and (ii) we were in effect adding extra cost to objects by

adding persistence to them and we wanted to measure the extra cost. This section can be skipped without losing the continuity in reading.

In section 5.4, we present the cost of the *Meta* portion of MetaStore. This is the cost of using the metaobject protocol of CLOS in extending it with object persistence. We measure the cost of each major feature of *Meta* in isolation by adding one feature at a time incrementally to CLOS until we reach the full MetaStore. Measurements made along the way are presented.

Finally in section 5.5, the cost of the *Store* portion of MetaStore is presented. This is the cost of the secondary storage management aspects of MetaStore.

Wherever appropriate, we present an expected performance of MetaStore implemented in Lucid CLOS: (i) for the benefit of CDRS, and (ii) because Lucid CLOS is considered one of the most efficient implementations of CLOS in use today.

5.1 Persistence and Metaobject Protocols

In this section, we describe our experience with the metaobject protocol of CLOS in extending it to support object persistence. The merits and drawbacks of using the protocol for persistence extension are described. We then discuss the two most significant difficulties we faced: maintaining dirty bits and dealing with shared structures. Finally, we propose possible improvements to the protocol.

5.1.1 Merits

The metaobject protocol of CLOS is sufficient to support language extensions as long as the extension involves modifying or augmenting the structure or behavior of objects.

Supporting object persistence as an extension of a language is a major task. Some extensions are to objects, whereas others are to other data structures. Extensions to objects can easily be done via the metaobject protocol, whereas extensions

to other data structures require some extra mechanism usually with the help from user programs or the language implementation level. All the things that we handle in MetaStore via the metaobject protocol are properties of persistence that can legitimately be considered object related. They include things that are described in section 3.8. They are all done with no help from either user programs, a Common Lisp compiler, or a run-time support system. We summarize some of them here:

- Separating the transient from the persistable.
- Keeping metaobject specific administrative information locally at metaobjects.
- Adding and modifying *pholes* at the time of creation of and access to persistable objects.
- Intercepting read and write accesses to objects to support the virtual object memory.
- Supporting persistence via inheritance.
- Intercepting read accesses to objects to support lazy loading.
- Intercepting write accesses to objects to update dirty bits.
- Adding slot options like `:transient` to control persistence at the slot level.
- Shadowing some system classes belonging to the metaobject protocol.
- Realizing the notion of a *husk*.

5.1.2 Drawbacks

In MetaStore, one level of indirection on slot accesses for each persistable composite slot of a persistable object is necessary to support persistence at the slot level. Each indirection is implemented by a *phole* sitting between the object and a

composite slot value. The value of a composite slot is stored in the phole associated with the slot. Thus, a `slot-value` call to a persistable object for a composite slot value in MetaStore returns the value in a phole rather than the phole associated with the slot. In implementing MetaStore, however, it is often necessary to get a phole rather than the user value in the phole.

This situation requires two different semantics for the `slot-value` method depending on who calls it and for what purpose. There are several ways of resolving this difficulty:

- Use a global variable, say `*return-phole-p*`, that MetaStore maintains. Depending on the value of this variable, the body of the `:around` method defined for `slot-value-using-class`, which is the workhorse routine for `slot-value`, is skipped. Therefore, every time a phole is needed, a caller would set this variable to `t`, call `slot-value` to get the phole associated with the slot, and reset the variable to `nil` on return. This approach is a possible solution since no user program would ever need a phole as the value of a `slot-value` call. Only some routines in MetaStore need pholes frequently. This solution was not chosen because each slot access is slowed down and the efficiency of slot accesses is critical for efficiently using objects.

- Add an extra keyword, say `:return-phole-p`, to the metaobject protocol interface routine `slot-value-using-class`, and conditionally return a phole or the value in a phole depending on the value of this keyword. This was not chosen for the similar reason as the first possibility. Also, adding an extra keyword argument is not allowed in the protocol.

- Isolate the body of the method `slot-value-using-class` and give a new name like `phole-slot-value-using-class`. This then returns a phole, whereas regular `slot-value-using-class` returns the value in a phole.

In MetaStore, the third possibility was chosen even though it violates the protocol because it was the most efficient one. Because MetaStore uses PCL [7], whose source code is available, this was easy to do. If we were more concerned about not violating the protocol than about the efficiency, we would have chosen the first choice.

`(setf slot-value)`, used to modify a slot value, is the dual of `slot-value`, posing the exact same problem. A similar implementation decision was made for `(setf slot-value)`.

There were two most significant difficulties we faced: maintaining dirty bits and dealing with shared structures, which we describe in the next section.

5.1.3 Abstraction Mismatch?

The metaobject protocol of CLOS is designed to support language extensions as long as the extensions are confined to the structure or behavior of objects. However, as soon as we try to augment the language with a feature that is not considered a property of objects, the metaobject protocol level is no longer sufficient.

Supporting object persistence requires some changes that are considered object-oriented as well as others that are considered base language implementation related. All the features of MetaStore that are object related are well supported by the metaobject protocol as we saw in section 5.1.1. The drawbacks discussed in section 5.1.2 are object-oriented, and it seems an oversight in the current design of the protocol, that can easily be fixed.

However, there are issues that must be handled in conjunction with persistence that are closely related to the low level base language implementation. They include:[1]

- Maintaining the dirty bits used to support incremental saves.

[1]See sections 3.10 and 3.11 for details.

- Preserving language semantics on structure sharing with persistence.

Tracking dirty bits and dealing with structure sharing seem to belong to the Common Lisp implementation level, not to the metalevel of CLOS. Dealing with these issues at the metalevel is difficult because of an abstraction mismatch. The Common Lisp Object System can be viewed as having five levels in implementation. They are:

- CLOS objects,
- Common Lisp,
- garbage collection,
- data types, and
- memory

with CLOS objects being the highest and memory the lowest in the abstraction spectrum. Dealing with dirty bits and structure sharing can best be done at levels such as "Common Lisp" and/or "garbage collection" in the list above. In MetaStore we tried to solve these issues at the "CLOS objects" level, so it is not surprising that it was not natural. We had to leave the metaobject protocol at times to deal with these issues. We had to devise extra mechanisms that required some help from user programs and the Common Lisp compiler.

5.1.4 Proposed Improvements to the Metaobject Protocol

There are improvements that can be made to the metaobject protocol for the short term. There are also improvements that could be made to the implementation techniques of Common Lisp for the long term.

5.1.4.1 Short-Term

Based on the experience of adding object persistence to CLOS in MetaStore, only a few minor improvements are proposed here to the existing protocol. They

are related to slot accessing as described in section 5.1.2. We propose that the protocol support a mechanism for one level of indirection on slot accesses. One possibility would be to provide two more routines as follows:

- `slot-value-using-class-direct`:

  This routine is identical to `slot-value-using-class` in all respects. We sometimes want to use the default behavior of `slot-value-using-class` and the changed behavior at other times. The changed behavior is usually obtained by specializing the default method with a `:before`, `:after`, or `:around` method. With the current protocol, once we modify the behavior of a method this way, we cannot use the default behavior any more although we must use it on occasion.

- `(setf slot-value-using-class-direct)`:

  This is the dual of `slot-value-using-class-direct` for write accesses to slots.

In general, this problem can be directed toward the semantics of object-oriented programming. When a method is augmented, say, by an `:around` method, there is no easy way to get the default behavior any more. We may want to extend the semantics of method combinations as follows. Even after a specialization method, e.g., an `:around` method, is defined in addition to a primary method, we are given the option of executing only the primary method without executing the specialization method. This is not an easy extension to support in general since it requires an elaborate control over all the methods: `primary` methods, `:before` methods, `:after` methods, and `:around` methods. Although it may be considered overkill, the `copy-as` operation of Jigsaw [9] would solve this problem.

5.1.4.2 Long-Term

As discussed in section 5.1.3, a seamless extension to CLOS of object persistence requires support from the base language implementation level. Judging from our experience with MetaStore, the metaobject protocol of CLOS seems well designed to support extensions to CLOS as long as the extension is inherently object-oriented.

To stay with the spirit of the metaobject protocol of CLOS to "open" up the language, it would be nice to push the metaobject protocol idea further down to the level of base language implementation. If we can support the metaobject protocol down at the Common Lisp data type level or at the garbage collection level,[2] the problems that we experienced in MetaStore (dirty bits and shared structures) could be easily solved. With this change, we would be able to enjoy the advantages of both "the base level persistence scheme" and "the metalevel persistence scheme" while encountering fewer of the disadvantages described in section 2.4.

5.2 PCL vs. Lucid CLOS

PCL is designed to support development,
not delivery of CLOS programs.
Gregor Kiczales and Luis Rodriguez

Because MetaStore was motivated by and intended to be used by CDRS, we had planned to implement it in Lucid CLOS [30]. However, it was instead implemented in PCL [7,23] because the Lucid CLOS implementation was not complete enough to the specification of [23] to implement MetaStore. Specifically, the slot level protocol, which is necessary for implementing the slot level granularity of persistence, was not available with Lucid CLOS at the time of the MetaStore implementation.

There are some major differences between Lucid CLOS and PCL in the object performance. Because MetaStore is implemented in PCL and the measurements gathered are based on this implementation, we present the difference between PCL

[2]Maybe, we would then call it *metadata protocol* or *metatype protocol.*

and Lucid CLOS so that we can project the expected performance of MetaStore ported to more complete implementation of Lucid CLOS. The current CDRS is implemented in Lucid Common Lisp with Lucid CLOS. Therefore, until MetaStore is fully ported to Lucid CLOS and CDRS is fully ported to MetaStore, we will not have "real" measurements for CDRS. Each measurement in this dissertation is based on *compiled* code.

Because the most important and critical aspects of an object system for a user program in terms of performance are creating objects, read and write accesses to objects, method dispatches, and method specializations, we compare PCL and Lucid CLOS using these dimensions. Objects used in this section are all transient objects. That is, MetaStore is not in the picture at all with the measurements in this section.

5.2.1 Creation

Two identical programs were written, one in PCL and the other in Lucid CLOS. In these programs, a class of 30 slots was used, and 1,000 objects were created. The results of creating these objects were as follows:

	Time	*Bytes Consed*
PCL	3.40 sec	256,008
LucidCLOS	0.06	152,008
Ratio	56.67	1.68

PCL runs over 50 times slower than Lucid CLOS does.[3] PCL uses almost 70% more space than does Lucid CLOS. Similar differences are seen with classes of different sizes. Classes with subclasses also shows similar differences.

5.2.2 Read Access

The context in which a read access to a slot is made makes a significant difference in both PCL and Lucid CLOS. The magnitude of difference in both implementations

[3] A comment by one of the local Lispers was "Holy sh*t!" on his first reading of these numbers.

is similar. Five different contexts were explored with read accesses on an object. They included: (i) reading a slot value of the "self" object within a method, (ii) reading a slot value of the "self" object within a method, but with the "self" object wrapped in a `let` binding, (iii) reading a slot value of an object other than the "self" object within a method, (iv-1) reading the value of the first slot of an object outside a method, e.g., within a function, and finally (iv-20) same as the case (iv-1) except reading the 20th slot.

We only show time comparisons since the space used in all cases is negligible. Using the two programs described in section 5.2.1, 100,000 accesses were made to an object with 30 slots in each benchmark run.

	(i)	*(ii)*	*(iii)*	*(iv-1)*	*(iv-20)*
PCL	0.13 sec	13.67	13.44	21.61	13.40
LucidCLOS	0.13	4.67	4.70	2.24	4.65
Ratio	1.00	2.92	2.85	9.64	2.88

Accessing different slots of a given object did not make a noticeable difference except in the case where a slot was accessed outside a method, e.g., in a function, as shown in cases (iv-1) and (iv-20).

In (i), there was no difference between PCL and Lucid CLOS. In (ii), (iii), (iv-1), and (iv-20), PCL was from approximately 3 to 10 times slower than Lucid CLOS was.

We now compare read accesses in different contexts in both PCL and Lucid CLOS.

	PCL	*Ratio*	*LucidCLOS*	*Ratio*
(i)	0.13 sec	1.00	0.13 sec	1.00
(ii)	13.67	105.15	4.67	35.92
(iii)	13.44	103.38	4.70	36.15
(iv-1)	21.61	166.23	2.24	17.23
(iv-20)	13.40	103.07	4.65	35.77

The difference between (i) and the rest is caused by the special optimization done to case (i). When an access is not made to one of the slots of the "self" object

directly, e.g., without intermediary `let` binding, within a method, the access speed gets drastically slower because of the loss of the optimization. The loss of the optimization makes PCL over 100 times slower, whereas the impact in Lucid CLOS is about 35 times slower.

5.2.3 Write Access

The context in which a write access to a slot is made makes a significant difference in both PCL and Lucid CLOS, as was the case with read accesses. The same five cases, (i), (ii), (iii), (iv-1), and (iv-20), were repeated for write accesses. 100,000 accesses were made to an object with 30 slots in each benchmark run and the time recorded.

	(i)	*(ii)*	*(iii)*	*(iv-1)*	*(iv-20)*
PCL	0.11 sec	14.22 sec	14.49 sec	22.76 sec	14.28 sec
LucidCLOS	0.12	3.90	3.92	1.50	3.90
Ratio	0.92	3.64	3.69	15.17	3.66

As was the case with read accesses, accessing different slots of a given object did not make a noticeable difference except in the case where a slot was accessed outside a method, e.g., in a function, as shown in cases (iv-1) and (iv-20). Lucid CLOS seems to optimize specially on accesses to the first slot of a class.

In the optimized case, there was no difference between PCL and Lucid CLOS. When the optimization was lost, Lucid CLOS was approximately four times faster than PCL was.

Write accesses are highly optimized only for one case where a slot of the "self" is updated within a method. We see a bigger difference in PCL than in Lucid CLOS.

	PCL	*Ratio*	*LucidCLOS*	*Ratio*
(i)	0.11 sec	1.00	0.12 sec	1.00
(ii)	14.22	129.27	3.90	32.50
(iii)	14.49	131.73	3.92	32.67
(iv-1)	22.76	206.91	1.50	12.50
(iv-20)	14.28	129.82	3.90	32.50

5.2.4 Method Dispatch

Because of the dynamic binding (section 2.1.2) of generic functions to methods, the cost of dynamic binding implemented as method dispatch is a critical aspect to consider in an object system. To measure the difference in efficiency of method dispatches between PCL and Lucid CLOS implementations, the following experiment was done. Fifty different methods, one for each of 50 classes of six slots, were defined with one generic function, `foo`. When `foo` is called with an object of one of the 50 classes, the dispatch of generic function `foo` happens to one of 50 methods depending on the class used. Each method performs a simple task, namely incrementing its first slot value by 1. Then, a driver function is defined, and called three times: (i) with `object-0`, an instance of the first class defined, (ii) with `object-25`, an instance of the 26th class, and (iii) with `object-49`, an instance of the 50th class. The driver function calls the generic function `foo` 1,000,000 times with one of these objects as its argument and the resulting time is shown below.

	object-0	*object-25*	*object-49*
PCL	6.00 sec	7.31 sec	7.30 sec
LucidCLOS	5.98	7.37	7.41
Ratio	1.00	1.00	1.01

Both PCL and Lucid CLOS show virtually equal performance. In both PCL and Lucid CLOS, the binding of generic functions to methods seems better optimized for the objects of the first class defined.

5.2.5 Method Specialization

When methods such as `make-instance`, `slot-value-using-class`, and `(setf slot-value-using-class)` in the metaobject protocol are specialized by a `:before`, `:after`, or `:around` method, PCL slows down substantially due to the loss of optimizations done to these methods; Lucid CLOS does not show a noticeable difference except in creating objects. To measure the differences, an `:around`

method is defined with null body for each of these methods. The `:around` methods do nothing but calling `call-next-method`. The effects of `:around` methods are presented here.

5.2.5.1 Creation

All three methods: `make-instance`, `slot-value-using-class`, and `(setf slot-value-using-class)`, in CLOS are used in creating objects. One thousand objects were created with and without `:around` methods for these three methods with the following results:

	w/out :around Methods	*w/ :around Methods*	*Ratio*
PCL	3.40 sec	4.32 sec	1.27
LucidCLOS	0.06 sec	2.86 sec	47.67
Ratio	56.67	1.27	

Without these `:around` methods, PCL runs almost 60 times slower than Lucid CLOS does. With them, there is no noticeable difference between PCL and Lucid CLOS.

The `:around` methods did not make much difference in creating objects in PCL. It is because there is not much optimization done in creating objects in PCL in the first place. For Lucid CLOS, the cost of these `:around` methods was almost 50 times of slowdown.

5.2.5.2 Read Access

One hundred thousand read accesses were made to a slot of an object with and without an `:around` method for `slot-value-using-class` with the following results:

	w/out :around Methods	*w/ :around Methods*	*Ratio*
PCL	0.13 sec	34.86 sec	286.15
LucidCLOS	0.13 sec	0.14 sec	1.07
Ratio	1.00	248.99	

Without the `:around` method, PCL and Lucid CLOS ran at about the same speed. With them, however, PCL ran about 250 times slower than Lucid CLOS did.

For PCL, the `:around` method made read accesses to a slot about 270 times slower. For Lucid CLOS, however, the overhead of the `:around` method was almost nonexistent.

5.2.5.3 Write Access

One hundred thousand write accesses were made to a slot of an object with and without an `:around` method for `(setf slot-value-using-class)` with the following results:

	w/out :around Methods	*w/ :around Methods*	*Ratio*
PCL	0.11 sec	35.90 sec	326.36
LucidCLOS	0.12 sec	0.12 sec	1.00
Ratio	1.09	299.17	

Without the `:around` method, PCL and Lucid CLOS ran at a similar speed. With them, however, PCL ran about 300 times slower than Lucid CLOS did.

For PCL, the `:around` method made write accesses to a slot over 300 times slower. For Lucid CLOS, however, the `:around` method made no difference.

5.2.6 Remarks

Object creation, read accesses, write accesses, and method specialization are all very critical aspects for MetaStore to be efficient. In implementing MetaStore, we used many `:around` methods: object creation, read accesses, and write accesses were all augmented by `:around` methods. In PCL, `:around` methods slowed down two of these three critical aspects substantially: almost 300 times for read accesses and almost 350 times for write accesses. In Lucid CLOS, on the other hand, was not affected at all by `:around` methods except on object creation: almost 50 times slowdown.

Because we are being severely penalized by PCL in the current implementation of MetaStore due to the heavy use of `:around` methods, we will take this into consideration in predicting the performance of MetaStore that will be running on Lucid CLOS.

5.3 Cost of Objects

The advantages of using object-oriented programming in software engineering are well known, but these advantages come with some extra costs. In this section, these costs are analyzed. This analysis is not critical to the rest of the dissertation. It is done as a reference information and can be skipped without losing the continuity.

We examine the following aspects of an object-oriented programming language, CLOS: object creation, read and write accesses to objects, and method calls.

Data structures that are most commonly used with programming in Common Lisp are arrays, structures, and lists. Thus, objects are compared against these data structures for the efficiency of creation of and accesses to them. The comparison was done because: (i) We wanted to use the most efficient data structures in implementing MetaStore since we were very performance conscious, and (ii) we were in effect adding extra cost to objects by adding persistence to them and we wanted to measure the extra cost. Because Lucid CLOS is the target implementation for MetaStore in CDRS, Lucid CLOS is used for these measurements. The differences we will see in the following subsections would be larger if PCL were used. The differences in the PCL case could be estimated based on the comparisons that was just done in the previous section.

5.3.1 Creation

A class of 30 slots, a structure created by `defstruct` with 30 slots, an array of 30 elements, and a list of 30 elements were created 100,000 times. A simple value,

an integer, was used as the value of each slot of an object, each slot of a structure, each element of an array, and each element of a list. The results were as follows:

	Time	*Ratio*	*Bytes Consed*	*Ratio*
Object	7.20 sec	1.00	19,200,288	1.00
Structure	2.00	3.60	13,600,008	1.41
Array	2.24	3.21	13,600,008	1.41
List	4.44	1.62	24,800,008	0.77

Creating objects is from approximately two to four times slower than creating other data structures. The fact that structures and arrays take about the same amount of time and use the same amount of memory for creation suggests that structures in Lucid Common Lisp are implemented as arrays.

Class hierarchy does not seem to add a noticeable difference. For example, for three classes `A`, `B`, and `C`, where `A` has 30 slots with no superclass, `B` has 15 slots with no superclass, and `C` has 15 slots and has `B` as its superclass, creating instances of `A` and creating instances of `C` are not noticeably different in both time and space.

When different sizes of objects, structures, arrays, and lists were used, the relative differences between them were similar to the measurements shown in this section.

5.3.2 Read Access

Because the context in which a read access to a slot of an object is made makes a significant difference, we consider two cases for read accesses to an object: (i) reading a slot value of the "self" object within a method, and (ii) reading a slot value of an object outside of a method, e.g., within a function. These two cases are compared against the cases with structures, arrays, and lists. In each case, the 20th element out of 30 was accessed 1,000,000 times. Memory consumptions on read accesses to these entities were negligible, thus not included.

	Time	*Ratio*
Object (i)	1.41 sec	1.00
Object (ii)	40.30	28.58
Structure	0.20	0.14
Array	4.04	2.87
List	0.22	0.16

Read accesses to a slot of an object within a method, case (i), was 7.04 times slower than read accesses to a slot of a structure, 2.87 times faster than read accesses to an element of an array, and 6.41 times slower than read accesses to an element of a list.

Read accesses to a slot of an object outside a method, case (ii), however was 201.50 times slower than read accesses to a slot of a structure, 9.98 times slower than read accesses to an element of an array, and 183.18 times slower than read accesses to an element of a list.

Lucid Common Lisp compiler seems to have done a better job in optimizing list read accesses than array read accesses.

5.3.3 Write Access

Because the context in which a write access to a slot of an object is made makes a significant difference, here again we consider two cases on write accesses to an object: (i) modifying a slot value of the "self" object within a method, and (ii) modifying a slot value of an object outside of a method, e.g., within a function. These two cases were compared against the cases with structures, arrays, and lists. In each case, the 20th element out of 30 was modified 1,000,000 times. Memory consumptions on write accesses to these entities were negligible, thus not included.

	Time	*Ratio*
Object (i)	1.18 sec	1.00
Object (ii)	39.74	33.68
Structure	0.43	0.36
Array	4.72	4.00
List	22.28	18.88

Write accesses to a slot of an object within a method, case (i), was 2.74 times slower than write accesses to a slot of a structure, 4.00 times faster than write accesses to an element of an array, and 18.88 times faster than write accesses to an element of a list.

Write accesses to a slot of an object outside a method, case (ii), however, was 92.41 times slower than write accesses to a slot of a structure, 8.42 times slower than write accesses to an element of an array, and 1.78 times slower than write accesses to an element of a list.

5.3.4 Functions vs. Methods

To measure the difference between a function call and a method call, the following experiment was done. A function, `foo`, was defined to increment an integer variable by 1. Because a method call requires at least one argument, `foo` is also defined with one argument. For the method case, 50 different methods, one for each of 50 classes of six slots, were defined with one generic function, `bar`. Each method again increments an integer variable, not a slot but a global variable as was done in the function case, by 1. Then, two driver functions, one for the function and the other for the method, were defined. Each driver function calls the function `foo` or the method `bar` 1,000,000 times and the resulting time is shown below. In the case of method calls, three different cases are sampled: (i) with `object-0`, an instance of the first class defined, (ii) with `object-25`, an instance of the 26th class, and (iii) with `object-49`, an instance of the 50th class.

	Time	*Ratio*
Function calls	1.98 sec	1.00
Method calls: object-0	6.00	3.03
object-25	7.31	3.69
object-49	7.30	3.69

The majority of the time spent on method calls are assumed to be spent on dispatching of the generic function call to appropriate methods. Although a method

call to an object of the first class defined seems to be a little faster than the other cases, method calls in general are almost four times slower than function calls.

5.3.5 Remarks

In general using objects is more expensive than other data structures. This was the case with all categories that we considered: creating objects, accessing objects, and calling generic functions.

The cost was especially significant when objects are not read or write accessed as the "self" object within a method. This suggests that some cases are optimized.

Based on these measurements, a programmer must decide on what data structures to use depending on how often and in what context they are accessed. As the rule of thumb, we use objects only when the fundamental features of object-oriented programming, e.g., inheritance and dynamic binding, are either necessary or very useful. Otherwise, structures, arrays, or lists are used.

In implementing MetaStore, objects were used according to this rule of thumb. Data structures and algorithms were carefully chosen in implementing MetaStore because efficiency was the foremost concern.

5.4 Cost of *Meta*

how long shalt it be then?
II Samuel 2: 26

There is certain overhead added to normal object creations and accesses by the extra mechanism added to support object persistence in CLOS via the metaobject protocol in MetaStore. In this section, we focus on the cost of dealing with objects in virtual memory such as creating objects, reading values of slots in an object, and changing the values of slots in an object. The cost associated with dealing with a secondary storage is analyzed in section 5.5.

The plan is to start with a very minimal subset of MetaStore and add one feature at a time with measurements made along the way. Thus, at each new stage, the cumulative cost of MetaStore is obtained. It would be ideal to isolate a feature, and measure the cost of it alone. However, in most cases, it is not feasible to isolate a feature because in most cases features are interdependent. In general, we will measure the cumulative cost by adding one feature at a time. Wherever feasible, an isolated cost for a feature is presented.

Our analysis of MetaStore is divided into six stages, starting with the simplest version of MetaStore and ending with the most comprehensive version. Each stage is designed to measure the cost of the following six features:

1. The cost of a metaobject class. To measure this, we strip down the implementation of MetaStore only to include the metaclass, **`persistent-metaclass`** that inherits the system defined **`standard-class`**, and a few extra things such as the inserted persistent root class that is to be inherited by a persistable user class.

2. The cost of **`:around`** methods used in MetaStore as specializations to methods in the metaobject protocol. The **`:around`** methods are defined as null routines except for calling **`call-next-method`**.

3. The cost of supporting persistence at the slot level. There is an extra mechanism added to the slot metaobject class to implement the grain size of persistence at the slot level, and the overhead of supporting this extra mechanism is measured.

4. The cost of MetaStore kernel. This is the basic mechanism of MetaStore that maintains pholes, the object table, keeping track of dirty bits, etc. At this level, shared structures and virtual object memory are not included.

5. The cost of handling shared structure. This stage measures the cost of supporting shared structures as described in section 3.11.3. Because a shared structure is perceived to be an expensive feature to support in a persistent object system and yet is not a very important feature to be supported in practice,[4] an isolated cost is measured so that a decision can be made as to whether the support of shared structures should or shouldn't be omitted in a persistent object system such as MetaStore.

6. The cost of the virtual object memory. This stage measures the cost of supporting the virtual object memory as described in section 3.12. The virtual object memory idea deals with the virtual memory of a program being filled up, thus affecting the system performance. Because it appears to be expensive to support in a persistent object system, an isolated cost of this feature is measured.

In the following subsections, each of these is added one at a time in order using the PCL implementation of MetaStore, and measurements are made. Where appropriate, calculated, expected performance of MetaStore for Lucid CLOS is also presented. In each subsection, the same dimensions of an object system that have been used are tried again for measurements: creating objects, read accesses to objects, and write accesses to objects.

The following experiment was done for each of these six features: one in each subsection below. A persistable class of 30 slots was defined. For creation, 1,000[5] objects were created while the time and space consumptions were measured. For

[4]Shared structures have not been supported in object persistence in CDRS thus far since there have been only a few isolated cases in use. They have been dealt with by application programs.

[5]Creating 1,000 objects at some levels does not take much time. However, it takes much more at different levels. Although a large number like 100,000 or 1,000,000 could have been used, 1,000 is chosen to reduce the waiting time during benchmark runs. Using the same number at each level is critical for accurate comparisons.

read and write accesses, 100,000 accesses were made to an object and the time was measured. For read and write accesses, the measurements on the space consumption were not included since they were all negligibly small. Although there were sizable differences depending on the context in which a read or a write access was made, only one case, reading or updating the value of a slot of the "self" object within a method, was measured here because:

- This is the context in which a read or a write access is most often made.
- We want to simplify the comparisons between different levels of MetaStore.
- From the differences we see in this one case, we can expect to see similar differences in other cases except in one case—the case where an access is made to the first slot in a function (case (iv-1) in section 5.2.2). Although case (i) in section 5.2.2 shows a huge difference compared with other cases, this analysis is valid because the huge difference is wiped out by the `:around` methods in MetaStore, bringing the difference to a negligible magnitude.

5.4.1 Cost of Metaobject Class

The persistent metaobject class `persistent-metaclass` was created in implementing MetaStore as a subclass of `standard-class`, a system provided metaobject class. The persistent root class `persistent-root-class`, whose metaobject class is `persistent-metaclass`, is also created to handle the persistence of user classes that inherit `persistent-root-class`. Thus, `persistent-root-class` is the workhorse for all the user defined persistable classes. In this section, we measure the cost of using `persistent-metaclass` and that of `persistent-root-class` being inherited by user classes. To measure this cost, a stripped down version of MetaStore is created which includes only `persistent-metaclass`, `persistent-root-class`,

and a few extra routines that establish the integration of these classes to the underlying CLOS implementation.

5.4.1.1 Creation

The measurements were made while creating 1,000 objects to be compared against those obtained from using a transient class of the same size.

	Time	*Bytes Consed*
Transient	3.40 sec	256,008
Persistent Metaclass	3.21	256,008
Ratio	1.05	1.00

There is no noticeable difference in either time or space. The minor difference in time measurements is attributed to the margin of errors rather than the difference caused by the extra mechanism being measured. There is no reason to believe that there would be any difference with Lucid CLOS.

5.4.1.2 Read Access

The measurements were made while read accessing an object 100,000 times to be compared against those obtained from using a transient class of the same size.

	Time
Transient	0.13 sec
Persistent Metaclass	0.13
Ratio	1.00

There is no effect of a metaobject class on the read access of a persistable object.

5.4.1.3 Write Access

The measurements were made while write accessing an object 100,000 times to be compared against those obtained from using a transient class of the same size.

	Time
Transient	0.11 sec
Persistent Metaclass	0.11
Ratio	1.00

There is no effect of a metaobject class on the write access of a persistable object.

5.4.1.4 Remarks on Metaobject Class

There is no noticeable overhead at this level. This is expected because all we are doing at this level is introducing a few extra subclasses.

5.4.2 Cost of Method Specialization via *:around* Methods

In implementing MetaStore, a number of `:around` methods are defined, the following three being the most notable: `make-instance`, `slot-value-using-class`, and `(setf slot-value-using-class)` to create objects, to read access an object, and to write access an object, respectively. To measure the cost of `:around` methods, an `:around` method was defined for each of these three methods, whose body does nothing but calling `call-next-method`. These `:around` methods were added on top of what was done in the previous subsection (5.4.1).

5.4.2.1 Creation

The measurements were made while creating 1,000 objects:

	Time	*Bytes Consed*
Before MetaStore	3.40 sec	256,008
After :around methods	4.32	368,008
Ratio	1.27	1.43

Creating objects after `:around` methods were added to those three methods mentioned above takes about 1.27 times more than creating objects without MetaStore in PCL.

Since Lucid CLOS was measured to be about 1.27 times faster than PCL when `:around` methods were used as we saw in section 5.2.5, MetaStore implemented in Lucid CLOS would have shown at most 3.40 (4.32 / 1.27 = 3.40) instead of 4.32, thus would be about the same as the native speed (3.40) of PCL.

5.4.2.2 Read Access

The measurements were made while read accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
Before MetaStore	0.13 sec
After :around Methods	34.86
Ratio	268.15

Dummy `:around` methods made read accesses almost 300 times slower than the normal transient read accesses. `:around` methods made the optimization done on read slot accesses disappear.

5.4.2.3 Write Access

The measurements were made while write accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
Before MetaStore	0.11 sec
After :around Methods	35.90
Ratio	326.36

Dummy `:around` methods made write accesses over 300 times slower than the normal transient write accesses. `:around` methods made the optimization done on write slot accesses disappear.

5.4.2.4 Remarks on Method Specialization

Object creations were not affected much (only about 30%) by the `:around` methods, whereas read and write accesses were slowed down by about 300 times. This is due primarily to the loss of optimization when `:around` methods are added because the body of these `:around` methods are kept empty except for calling `(call-next-method)`.

5.4.3 Cost of Slot Level Persistence

To support persistence at the slot level as well as to let users be able to selectively make slots persistable, the slot metaobjects are augmented in MetaStore. The cost of MetaStore is measured after this extra mechanism is added on top of MetaStore measured in the previous subsection (5.4.2).

5.4.3.1 Creation

The measurements were made while creating 1,000 objects:

	Time	*Bytes Consed*
Before MetaStore	3.40 sec	256,008
After :around Methods	4.32	368,008
After Slot Level Persistence	4.67	368,008
Ratio wrt up to :around Methods	1.08	1.00
Ratio wrt Transient	1.37	1.43

The slot level persistence made creating objects about 1.37 (4.67 / 3.40 = 1.37) times slower than creating objects without MetaStore in PCL. With respect to MetaStore up to `:around` methods, it added about 8% overhead.

5.4.3.2 Read Access

The measurements were made while read accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
Before MetaStore	0.13 sec
After :around Methods	34.86
After Slot Level Persistence	38.41
Ratio wrt up to :around Methods	1.10
Ratio wrt Transient	295.46

MetaStore up to this level made read accesses about 300 times slower than the normal transient read accesses. With respect to MetaStore up to `:around` methods, it added about 10% overhead.

5.4.3.3 Write Access

The measurements were made while write accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
Before MetaStore	0.11 sec
After :around Methods	35.90
After Slot Level Persistence	39.00
Ratio wrt up to :around Methods	1.09
Ratio wrt Transient	354.55

MetaStore up to this level made write accesses over 350 times slower than the normal transient write accesses. With respect to MetaStore up to `:around` methods, it added about 9% overhead.

5.4.3.4 Remarks on Slot Level Persistence

Slot level persistence added about 10% overhead in all three aspects with respect to the previous level, up to `:around` methods. The main source of the overhead is the one level of indirection added to all persistable composite slots.

5.4.4 Cost of MetaStore Kernel

This is the cost of the basic mechanism of MetaStore that allows the minimum functionality of MetaStore: being able to define persistable classes, being able to selectively declare slots to be persistable, being able to perform incremental saves, being able to load on demand, etc. Therefore, we maintain object identities, pholes, the object table, dirty bits, model identities for interfacing the object base, etc. at this level. Features measured up to the previous subsection (5.4.3) are of course all included. They include the cost of metaobject classes, `:around` methods, and slot level persistence. At this level, shared structures and virtual object memory are not included because they will be treated as separate features on top of the MetaStore kernel in the coming sections.

5.4.4.1 Creation

The measurements were made while creating 1,000 objects.

	Time	*Bytes Consed*
Before MetaStore	3.40 sec	256,008
After Slot Level Persistence	4.67	368,008
MetaStore Kernel	56.35	6,152,008
Ratio wrt Slot Level Persistence	12.07	16.72
Ratio wrt Transient	16.57	24.04

The MetaStore kernel made creating objects about 16 times slower than creating objects without MetaStore. It ran about 12 times slower than MetaStore with up to slot level persistence.

Creating objects in the MetaStore kernel used about 24 times more space than creating objects without MetaStore, and about 17 times more than MetaStore with up to slot level persistence.

5.4.4.2 Read Access

The measurements were made while read accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
Before MetaStore	0.13 sec
After Slot Level Persistence	38.41
MetaStore Kernel	35.62
Ratio wrt up to Slot Level Persistence	0.93
Ratio wrt Transient	274.00

Read accesses in the MetaStore kernel was about 270 times slower than the normal transient read accesses. With respect to MetaStore with up to slot level persistence, it actually ran faster by about 7%. This unexpected result is attributed to the margin of errors.

5.4.4.3 Write Access

The measurements were made while write accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
Before MetaStore	0.11 sec
After Slot Level Persistence	39.00
MetaStore Kernel	254.70
Ratio wrt up to Slot Level Persistence	6.53
Ratio wrt Transient	2,315.45

Write accesses in the MetaStore kernel was over 2,000 times slower than the normal transient write accesses. With respect to MetaStore with up to slot level persistence, it ran about 6 times slower. The slowdown by about six times is attributed to the case analysis on the value of a slot before and after the update because a phole is either added to or removed from a composite slot depending on what kind of a value is changed to what other kind of a value.

5.4.4.4 Remarks on MetaStore Kernel

Read accesses in the MetaStore kernel did not add any additional cost as expected because there is no extra work added to the read mechanism at the kernel level. The main cost added on object creation is due to the addition of pholes to persistable composite slots and the case analysis of slot values that are being used as the initial values. Write accesses added substantial extra cost. Most of it is caused by (i) the `:around` methods and (ii) the case analysis on the slot values in order to add or remove a phole if necessary.

If we were to eliminate the overhead caused by `:around` methods, object creation and write accesses would be about 13 and 7 times slower respectively than the transient case; read accesses would be about the same as the transient case.

If MetaStore were to run on Lucid CLOS, object creation and write accesses would be about 4 and 7 times slower respectively than the transient case; read accesses would be about the same as the transient case.

5.4.5 Cost of Shared Structures

On top of the MetaStore kernel, supporting the persistence of shared structures as designed in section 3.11.3 was added, and the measurements were made in creating objects, reading slot values, and writing slot values to see the isolated cost of handling shared structures.

5.4.5.1 Creation

The measurements were made while creating 1,000 objects:

	Time	*Bytes Consed*
MetaStore Kernel	56.35 sec	6,152,008
With Shared Structures	56.22	6,224,008
Ratio	1.00	1.01

In creating objects, there was no noticeable difference added by handling structure sharing. The fact that it took less time with structure sharing handled above is considered the margin of error in benchmark runs. The main cost of dealing with shared structures in MetaStore comes from two sources: (i) making pholes lazily and (ii) looking up an "eq-test" hash table. The cost of creating pholes lazily was negligible compared with the cost of the MetaStore kernel due to the optimization done as follows. A resource manager was used to minimize the effect of having to create pholes on the fly. A large number (100,000) of pholes was made in advance and put into a pool. Each time one is needed, one is checked out of the pool. When the pool is empty, it is refilled with a number smaller (10,000) than the original size. The cost of creating pholes became negligible with the pool. The cost of looking up a hash table was again negligible compared with the cost of the MetaStore kernel. For an "*eq*-test" hash table, looking up each entry once of a table with 100,000 entries takes only 0.02 seconds although initializing a hash table of the same size takes about 66.77 seconds as shown in the table below.

	Time	*Bytes Consed*
Initializing a Hash Table	66.77 sec	62,764,728
Looking up a Hash Table	0.02	352

The time measured for the cost of creating objects above did not include adding entries to the hash table. It only included the time to look one up as many times as it was necessary to create 1,000 objects.

5.4.5.2 Read Access

The measurements were made while read accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
MetaStore Kernel	35.62 sec
With Shared Structures	38.83
Ratio	1.09

The version of MetaStore that handles shared structures added about 9% overhead in reading slot values over the MetaStore kernel.

5.4.5.3 Write Access

The measurements were made while write accessing an object 100,000 times. A slot of the "self" object within a method was accessed that many times.

	Time
MetaStore Kernel	254.70 sec
With Shared Structures	261.90
Ratio	1.03

The version of MetaStore that handles shared structures added about 3% overhead in writing slot values over the MetaStore kernel.

5.4.5.4 Remarks on Shared Structures

Supporting the persistence of shared structures as designed in section 3.11.3 did not add much overhead when compared against the MetaStore kernel as demonstrated with the measurements in creating objects and read and write accesses to objects. This was consistent with what was expected because the main source of

overhead was from maintaining an extra "*eq*-test" hash table. When the incremental difference was compared against the measurements made from the transient case, however, the difference was still significant in read and write accesses as we can see below:

	Time	*Ratio*
Read Access: Transient	0.13 sec	1.00
Shared structures	3.21	24.69
Write Access: Transient	0.11	1.00
Shared structures	7.20	65.45

5.4.6 Cost of Virtual Object Memory

On top of the MetaStore kernel with shared structures as was done in the previous subsection (5.4.5), the virtual object memory as designed in section 3.12 was added and the measurements were made in creating objects, reading slot values, and writing slot values to see the isolated cost of the virtual object memory. Here, 10,000 objects were created and 1,000,000 read and write accesses were made. A larger numbers than the ones used so far were used because the virtual object memory idea itself was designed to deal with large number of objects.

5.4.6.1 Creation

The measurements were made while creating 10,000 objects. Three variations were tried: (a) with the high water mark at 50,000 so that there was no flushes happening while objects were being created, (b) with the high water mark at 5,000 so that there was 5,000 flushes done, one flush at a time, and (c) with the high water mark at 5,000, but 20% (1,000) of objects in use were flushed at a time so that there was 5 flushes. The results were as follows:

	Time	*Bytes Consed*
Kernel w/ Shared Structures	224.35 sec	21,200,288
(a) No Flushes	234.99	23,360,008
(b) One Flush at a Time	240.45	23,360,008
(c) Flush 20% at a Time	247.54	23,360,248
Ratio: (a) vs. Kernel	1.05	1.10
(b) vs. Kernel	1.07	1.10
(c) vs. Kernel	1.10	1.10

Supporting the virtual object memory added some overhead in all three cases experimented. (a) When the high water mark was set high enough so that no flushing was done, about 5% overhead was added. (b) When the high water mark was set low and if one object was flushed at a time, about 7% overhead was added. (c) When the high water mark was set low and if 20% of objects in virtual memory was flushed every time the high water mark was reached, about 10% overhead was added. In space, about 10% overhead was added in all three cases compared with the MetaStore kernel.

5.4.6.2 Read Access

The measurements were made while read accessing an object 1,000,000 times with 10,000 objects in the high water watch and with the high water mark set at 50,000. A slot of the "self" object within a method was accessed that many times, and the results were as follows. Note that it did not make sense to try different values for the high water mark on read accesses, thus only one case was explored.

	Time
Kernel w/ Shared Structures	320.93 sec
With Virtual Object Memory	356.20
Ratio	1.11

The version of MetaStore that handles the virtual object memory added about 11% overhead in reading slot values over the MetaStore kernel with shared structures handled.

5.4.6.3 Write Access

The measurements were made while write accessing an object 1,000,000 times with 10,000 objects in the high water watch and with the high water mark set at 50,000. A slot of the "self" object within a method was accessed that many times, and the results were as follows. Here again, it did not make sense to try different values for the high water mark on write accesses; thus only one case was explored.

	Time
Kernel w/ Shared Structures	2,652.04 sec
With Virtual Object Memory	2,702.53
Ratio	1.02

The version of MetaStore that handles the virtual object memory added about 2% overhead in writing slot values over the MetaStore kernel with shared structures handled.

5.4.6.4 Remarks on Virtual Object Memory

Supporting the virtual object memory as designed in section 3.12 did not add much overhead when compared against the MetaStore kernel with shared structures handled as demonstrated with the measurements in creating objects and read and write accesses to objects. This was consistent with what was expected because the main source of overhead was from maintaining `high-water-watch`, whose algorithm was analyzed in section 3.12.3. When the incremental difference was compared against the measurements made from the transient case, however, the difference was still significant in read and write accesses as we can see below:

		Time	*Ratio*
Creation:	Transient	16.10 sec	1.00
	Virtual Object Memory	34.00	2.11
Read Access:	Transient	1.30	1.00
	Virtual Object Memory	35.27	27.13
Write Access:	Transient	1.10	1.00
	Virtual Object Memory	50.49	45.90

5.4.7 Remarks

cause me to understand wherein I have erred
Job 6: 24

Focusing on the MetaStore kernel, most of the overhead was caused by (i) the `:around` methods and (ii) the case analysis on slot values to add or remove pholes if necessary on object creations and write accesses. Although substantial, some changes in the design of MetaStore would eliminate most of this overhead. To see what kind of difference in performance we could expect, some changes in the design of MetaStore were explored. The use of metaobject protocol was still kept except that all `:around` methods were eliminated. The changes that were necessary to eliminate most of the overhead included:

- Replacing all the `:around` methods with macros. For example, the macro `p-make-instance` is defined, which calls `make-instance` to create an object. Immediately after calling `make-instance`, it handles other persistence related work such as adding pholes to slots. Similar macros are defined for `slot-value-using-class` and `(setf slot-value-using-class)`. These macros in essence simulate the `:around` methods.

- Blindly adding a phole to each slot so that the case analysis on the type of a slot value is not necessary. Because the case analysis is done on every write access to an object, the overhead was significant. Adding a phole to each slot would eliminate the case analysis. This new algorithm, of course, uses more space. Thus, it is the issue of trade offs between time and space. Because either a read or a write access to a slot is one of the most critical aspect of an object system in terms of performance, time might be considered more important than space. Each phole is a record with five fields.

With these changes, measurements were made to compare with the measurements obtained with the MetaStore kernel in the original design as was presented in section

5.4.4. The measurements we see with these changes are the best we could hope for unless we ask users to explicitly tell MetaStore about dirty bits at all times.

5.4.7.1 Creation

For creating objects, the most noticeable difference was seen when the MetaStore kernel (56.35 seconds) was compared with the normal transient case (3.40 seconds). With these changes, the following measurements were obtained:

	Time
Before MetaStore	3.40 sec
MetaStore Kernel (Original Design)	56.35
MetaStore Kernel (Changed Design)	3.44

With these changes, creating objects in MetaStore kernel took about the same amount of time as creating objects with the native PCL. Because Lucid CLOS is about 56.67 times faster (section 5.2.1) in creating objects than PCL, the overhead of MetaStore in Lucid CLOS with these changes in the design would not be noticeable.

5.4.7.2 Read Access

For read accesses to an object, the most noticeable difference was seen when an `:around` method was defined for `slot-value-using-class`. A read access with the `:around` method was about 268.15 times slower than one without it (section 5.4.2). With these changes, the following measurements were obtained:

	Time
Before MetaStore	0.13 sec
MetaStore Kernel (Original Design)	35.62
MetaStore Kernel (Changed Design)	0.13

With these changes, a read access to an object in MetaStore was the same in speed as the one to a transient object. This is quite consistent with our expectation because about the only extra work that a read access does with this new design

is one level of indirection, which is looking up a value in a structure created by `defstruct` in our implementation. Looking up a value in a structure is negligible as was shown in section 5.3.2. The access is to a slot value that has been instantiated. The slots that are not instantiated will be analyzed in section 5.5.

Even with the original design of MetaStore, a read access in MetaStore will be almost as fast as a read access in the native Lucid CLOS. As is shown in section 5.2.5, Lucid CLOS was not affected much on read accesses by the `:around` methods. That is, the optimization done on reading slot values is not lost by added `:around` methods in Lucid CLOS. Thus, most of this difference on read accesses between the MetaStore kernel and transient case would disappear when MetaStore is ported to Lucid CLOS.

5.4.7.3 Write Access

For write accesses to an object, the most noticeable difference was seen again when an `:around` method was defined for `(setf slot-value-using-class)`. A write access with the `:around` method was about 326.36 times slower than one without it (section 5.2.5). With these changes in the design of MetaStore, the following measurements were obtained:

	Time
Before MetaStore	0.11 sec
MetaStore Kernel (Original Design)	254.70
MetaStore Kernel (Changed Design)	0.54

Even with these changes, a write access to an object with MetaStore was still about five times slower than the one with a write access in the native PCL. The extra overhead was caused by maintaining dirty bits. When a slot is modified repeatedly, this overhead may be too much to bear. If the overhead in write accesses is too large to accept, one possible solution would be to force user programs to tell the MetaStore explicitly what is becoming dirty. When a slot value is

repeatedly modified, this change would be a big win because a user program must tell MetaStore only once. With the tools that detect programming errors as described in section 3.10.3, this may be an acceptable alternative.

5.5 Cost of *Store*

So far, the focus has been on the cost of dealing with objects in virtual memory. In this section, the cost associated with dealing with secondary storage is analyzed. It is expected that a disk access is much slower than an in-memory access. Thus, the secret is to minimize disk accesses in tight loops. One of the goals in MetaStore is to keep as much data in virtual memory as possible while maintaining the system performance at an acceptable level.

First, the performance of saving objects is analyzed, followed by that of lazily loading objects. Then, object and slot clustering is analyzed, followed by preloading and some general remarks.

5.5.1 Saving Objects

To measure the cost of saving objects to secondary storage (the object base), 1,000 objects of various sizes were created and saved. In the current implementation of MetaStore, all Lisp objects are saved in textual forms, i.e., in the ASCII format. Based on our experience with CDRS, saving some selected Lisp data types such as floating point numbers into a binary format reduces the saving time by a factor of 3. Saving all persistent data in a binary format would certainly improve even further, but it was not done in the initial implementation. The measurements made on saving objects were as follows:

Total File Size	*Average Object Size*	*Saving 1,000 Objects*
151,101 bytes	151 bytes	24.85 sec
951,785	951	35.45
1,500,656	1,500	44.68
2,940,156	2,940	65.56
6,046,286	6,046	107.39

From the table above, the time it took to save an object was almost directly proportional to the size of an object. Because a small number (on the order of 10 to 20) of objects are modified on the average in each user operation in a CAD system like CDRS, where the average object size is about 350 bytes, it would take about 0.53 seconds for 20 objects if we decided to save all the modified persistable objects after each user operation. That would be quite acceptable.

The major contributors on the saving time are: (i) encoding objects which translates memory addresses to persistent identifiers and (ii) the actual writing of encoded objects to a disk file. Most of the saving time is spent on the latter phase as we can see in the table below. In each case, 1,000 objects were saved:

Total File Size	*Average Object Size*	*Encoding Time*	*Writing Time (%)*
151,101 bytes	151 bytes	12.27 sec	12.58 sec (50.62)
951,785	951	13.40	22.05 (62.20)
1,500,656	1,500	16.94	27.74 (62.08)
2,940,156	2,940	21.83	43.73 (66.70)
6,046,286	6,046	31.52	75.89 (70.65)
			Average % (62.45)

5.5.2 Lazily Loading Objects

In this section, the speed of lazy loading is analyzed. The objects saved in various files in the previous section (5.5.1) were loaded one file at a time in the following fashion. As soon as objects are saved to a model file, the objects just saved are modified so that the value of a slot is `nil`. Then, each object is accessed one at a time for a read of the slot that is just set to `nil`, which causes the slot to be instantiated by loading the value of the slot from the model file. Because only one slot of each object is loaded, the reads done for each object were not sequential. This test was done for each of those five model files saved in the previous section. Thus, in each test, 1,000 composite slot values were loaded, each instantiating a slot in an object. The results are shown below:

Total File Size	*Average Slot Size*	*Loading 1,000 Slots*
151,101 bytes	50 bytes	2.86 sec
951,785	317	7.14
1,500,656	500	10.11
2,940,156	980	18.05
6,046,286	2,015	35.32

In the MetaStore kernel, the time taken for 1,000 in-memory read accesses was about 0.36 seconds (section 5.4.4) and the comparison between this number with the ones in the above table is shown below:

	1,000 Reads	*Ratio*
MetaStore Kernel (in-memory)	0.36 sec	1.00
MetaStore kernel (loading from disk)		
50 bytes	2.86	7.94
317	7.14	19.83
500	10.11	28.08
980	18.05	50.13
2,015	35.32	98.11

As expected, a read access to a slot whose value is in disk is much slower than the one to a slot whose value is in virtual memory. The cost of instantiating a slot with a value from a disk file is the combination of the following:

- Disk seek time
- Reading bytes in a disk file and parsing them into Lisp objects
- Translating persistent identifiers back to virtual addresses

As is the case with saving objects, most of the time was spent on the middle phase ("Reading bytes in a disk file and parsing them into Lisp objects") as can be seen in the table below. In each case, 1,000 composite slot values were loaded:

Total File Size	*Ave Object Size*	*Seek*	*Reading (%)*	*Translate*
151,101 bytes	151 bytes	0.08 sec	1.17 sec (40.91)	1.61 sec
951,785	951	0.10	4.11 (57.56)	2.93
1,500,656	1,500	0.09	6.19 (61.23)	3.83
2,940,156	2,940	0.10	11.72 (64.93)	6.23
6,046,286	6,046	0.08	23.47 (66.45)	11.77
			Average % (58.21)	

5.5.3 Object and Slot Clustering

To reduce the loading time of an object or a slot, clustering objects and slots were exploited. The main goal of clustering objects and slots is to reduce the disk seek time by controlling the locality of data.

To see the effect of clustering objects and slots, three different possibilities were prepared and compared:

1. *Loading by sequential reads:* This is the best any clustering algorithm could do.

2. *Typical lazy loading:* These are the results shown in the previous section.

3. *Worst case lazy loading:* On each loading of a slot value, a substantial disk seek is made to occur. That is, on each slot access, a load is done from a location far enough from the previous read so that a substantial disk seek time is guaranteed to happen. In each try with those five files saved in section 5.5.1, the first object was accessed, followed by the 500th object, the half way down in the file, followed by the second, followed by 501st, then 3rd, followed by 502nd, etc.

The measurements made in each of these three cases were as follows:

Total File Size	*Ave Slot Size*	*Sequential*	*Typical*	*Worst*
151,101 bytes	50 bytes	1.21 sec	2.86 sec	3.16 sec
951,785	317	6.33	7.14	7.50
1,500,656	500	9.97	10.11	10.60
2,940,156	980	17.44	18.05	18.23
6,046,286	2,015	35.12	35.32	36.12

The typical case is about 2.36 times, 1.13 times, 1.01 times, 1.03 times, and 1.01 times faster than the sequential case for the average slot size of 50 bytes, 317 bytes, 500 bytes, 980 bytes, and 2,015 bytes respectively.

The worst case is about 2.61 times, 1.18 times, 1.06 times, 1.05 times, and 1.03 times faster than the sequential case for the average slot size of 50 bytes, 317 bytes, 500 bytes, 980 bytes, and 2,015 bytes respectively.

The effect of clustering is the most noticeable when the average size of data read in is small, which is expected because multiple reads can be made in a single read buffer fill when there are many small items in the buffer. As the average size of slot values gets larger, the effect of clustering is negligible. With large slot values or objects, large enough to fill at least half the buffer, each read would require a substantial disk head movement, thus diminishing the effect of clustering. In most object-intensive applications that use many large slot values and objects, clustering would not help very much.

The effect of disk seek time was not large at all based on the numbers in the table above, and this was supported by the following simple experiment. To see the cost of disk seek time, 3,000 seeks were made at 980 byte intervals for a file of 2,940,156 bytes. These 3,000 seeks took only 0.25 seconds, which is negligible compared with the overall time spent on reading any of those five files above. This suggests that the majority of time spent on loading an object or a slot value is for something other than disk seek time.

We expected that clustering would not be good enough to compensate for the extra overhead of loading objects from a disk file, but did not expect that the effect would be this small. An alternative was thus benchmarked in the next section.

5.5.4 Preloading Objects

Among the major factors that contribute to the cost of loading an object or a slot value as listed in section 5.5.2, reading and making Lisp objects out of a byte stream and decoding persistent identifiers were the main sources of overhead, much larger than the time spent on disk seeks as we saw in the previous section.

Therefore, substantial improvements can only be made by improving one of these main contributors. However, dealing with reading bytes in a disk file and parsing bytes into Lisp objects are beyond the scope of this research. There isn't much that can be done to improve the algorithm for decoding persistent identifiers. Therefore, the idea of preloading was exploited, and some benchmark results are presented in this section.

The most important and critical aspect of preloading was to decide what to preload. Unfortunately, it is very much application dependent. In CDRS, for example, things that are needed to display objects on the screen will be preloaded. Beyond that, other things are preloaded depending on what a user intends to do, e.g., modeling, rendering for pictures, or both.

On a real world model built by one of the automotive companies that use CDRS as a design tool, the time taken to preload all the objects needed for displaying and modeling was compared with the time it took to load the model in batch mode without MetaStore.

File Size	*Full Load*	*Preload*	*Ratio*
5,529,850 bytes	1,455 sec	158 sec	9.21

"Full load" loads more data than "Preload" does. However, with "Preload," one can choose what to load initially and load only the things that need be loaded on demand later while a user is using the system. Although some extra time is spent on lazy loading, the benefit of not having to have all the objects loaded into virtual memory all the time would outweigh the annoyance brought by the extra time required by lazy loading. Because a small number (on the order of 10 to 20) of objects are needed on the average in each user operation in a CAD system like CDRS, where the average object size is about 350 bytes, it would take about 0.14 seconds for 20 objects if we decided to lazily load all the objects needed for each operation. Once enough objects are loaded, the overhead of lazy loading will not

be very noticeable simply because there won't be much lazy loading to do any more then.

Cutting the initial waiting time in loading by a factor of 10, possibly more depending on what a user wants to do, is a significant improvement. With the capability of preloading objects selectively, users now have control over what is loaded, thus being able to cut the waiting time.

5.5.5 Remarks

The critical aspects of dealing with secondary storage is the performance of saving and loading objects. The numbers in section 5.5.1 suggest that the performance of saving objects is acceptable because there will in general be many small transactions of saving small number of objects at a time. In an application like CDRS, it would generally be 10 to 20 objects and about a half second to save this many objects at a time. Being able to do an incremental saving of objects reduces the waiting that a user must do in many full saves of all the objects as has been done in CDRS.

The numbers in section 5.5.2 suggest that a truly lazy loading of objects and slots of an average size is about 20 times slower than the speed of an in-memory access of a slot in the MetaStore kernel. It wouldn't be acceptable to load all the objects one needs lazily because of this overhead. It has been shown in CDRS that loading all the objects at the beginning of an editing session is not acceptable, either. Therefore, a compromise between these two extremes is benchmarked. It is the combination of preloading and lazy loading. Once we load enough of what a user initially needs given an editing session, whatever else needed is then loaded lazily. The numbers in section 5.5.4 suggest that preloading of all the keyed objects in a real world model in CDRS is about 10 times faster, which takes about 2.5 minutes, than a full load taking over 24 minutes. When a user wants to render a

model for pictures instead of editing models, for example, much less data would be necessary to be loaded, thus requiring much less waiting time than even 2.5 minutes, we project. A more important benefit of this approach is to be able to control what is loaded. In most cases, only the things that are truly needed are loaded, thus enabling a better utilization of the memory space available for a program. With CDRS, we have seen it crashing many times because the system runs out of space in performing a garbage collection. With preloading, lazy loading, and the virtual object memory, this sort of problem will go away.

One disappointing aspect in our benchmark results is the insignificance of the benefits from object and slot clustering. Because the majority of loading time is spent on something other than disk seeks, improvements made by clustering is negligible. The benefit of clustering objects is most noticeable when the objects are small in size. In an application like CDRS, however, objects are large enough that the benefit of clustering is negligible. Improving the Lisp reader would be a good investment for a quick improvement on loading objects. Based on our experience with CDRS, a customized reader for a specific kind of data are about 30 times faster than the general reader supported by Lucid Common Lisp. Although the benefit would be quite noticeable, it is not very interesting and is beyond the scope of this research.

6

Summing Up

Better is the end of a thing than the beginning of it
Ecclesiastes 7: 8

In this dissertation we have described the design, an implementation, and the implementation results of the persistent object system MetaStore. MetaStore was implemented using the metaprogramming facilities of an object system to extend it with persistence. In the initial implementation, the metaobject protocol of CLOS was used. The main goal of MetaStore was to support persistence for object-intensive applications such as CDRS.

Achieving this goal required designing a persistent object store that provides a database management system to deal with secondary storage and extending an object-oriented programming language via its metaprogramming facilities to add object persistence. We summarize these research contributions in section 6.1. In the course of our research we have identified a number of areas that merit further investigation; we outline these areas in section 6.2.

6.1 Contributions

Know honor, yet keep humility.
Lao Tsu, Tao Te Ching

The primary contribution of this dissertation is the design of a portable persistent object system that uses the metaprogramming facilities of an object-oriented programming language, thus not requiring any help from user programs or any support from the language compiler or the run-time system. A persistent object store was

also designed to support the database management features for the persistent object system. To monitor the amount of data in virtual memory, a virtual object memory scheme was also added to the system.

We identified three schemes via which an object-oriented programming system can be extended with persistence. Persistence can be added to a programming system at the base language level, at the application level, or at the metalevel. The metalevel scheme was chosen for MetaStore.

The metaobject protocol [23] of the Common Lisp Object System [8] was used to extend CLOS with object persistence at the metalevel in the initial implementation of MetaStore. By doing so, we investigated the feasibility of using metaprogramming in language design. Our experience showed that the metaobject protocol of CLOS was indeed powerful enough to do most of what was required for persistence extension although a few deficiencies and some concerns were encountered. Some were deficiencies in the design of the metaobject protocol, and others were related to performance in the current implementation of the metaobject protocol used in MetaStore, and they are listed here:

1. Maintaining dirty bits for incremental saves could not be done for things that do not have their own persistent IDs without some help from user programs and/or the base language implementation level. An array in Common Lisp is such an example. Objects and slots in MetaStore were fine because they do have IDs. Trying to maintain dirty bits for nonobject composite values in MetaStore was the case when we were attempting to support orthogonal persistence without the proper support from the language implementation layer.

2. The situation in supporting the persistence of shared structures was quite similar to the one in dealing with dirty bits. Structured data that are neither

objects nor direct slot values in MetaStore have no persistent IDs. Therefore, help from user programs and/or the base language implementation level was also required to support persistence of such structures.

3. Supporting persistence at the slot level granularity required one level of indirection on slot accesses and the metaobject protocol does not have a way of supporting the indirection.

4. Some optimizations done in the current implementations of PCL [7] and Lucid CLOS [30] were lost when some aspects of the metaobject protocol were used. The loss of optimizations is significant enough that we would have to either change the design of MetaStore or improve the implementation of CLOS, the latter being our preference.

The first two are considered properties belonging to the base level language implementation technology, the third is a deficiency of the metaobject protocol, and the last a limitation in the current implementation of the protocol.

Based on the initial implementation of MetaStore, we propose some short-term and long-term improvements to the metaobject protocol of CLOS:

- *Short-term:* In the short-term we propose the addition of one level of indirection, that can be used optionally, on slot accesses of objects. To support the slot level persistence, one level of indirection on slot accesses was necessary for the persistence implementor.

  To be more general, it would be even better to extend the syntax and semantics of method combinations in object-oriented programming languages in such a way that specialized methods can be optionally skipped at the user's control. This is not an easy extension to support in general because it requires an elaborate control over all the methods: `primary` methods, `:before` methods,

`:after` methods, and `:around` methods. Given this option, an `:around` method that handles persistence can optionally be skipped when necessary.

- *Long-term:* In the long-term, we propose that the metaobject protocol be pushed down to the base language implementation level by implementing each data type supported in the language as an object with appropriate access via metaobject protocols so that seamless extension of persistence can be made without having to depend on the user programs' help or on ad hoc means of dealing with difficult issues such as maintaining dirty bits or handling shared structures as was done in the initial implementation of MetaStore. By pushing the protocol down to the base language implementation level, we can enjoy the advantages of the metalevel and base level persistence schemes as well as those of the application level scheme. The only concern with this change would be of performance, but we have shown in section 5.3 that objects are not much more expensive than other traditional data structures such as arrays, lists, and structures as long as we can add adequate optimizations and preserve them. Designing CLOS with the metaobject protocol was a careful decision and proven by MetaStore and by Rodriguez [41] to be a good one; we now have to extend it to another level down with another careful decision.

The most critical aspects of an object-oriented programming system in terms of performance are: (1) creating objects, (2) accessing an object for a read, (3) accessing an object for an update, and (4) method dispatching or more generally method calls. The impact of extending an object-oriented programming system with persistence at the metalevel turns out to be substantial: the current design of MetaStore overburdens object creation and write accesses by approximately 13 times and 7 times respectively compared with transient cases. The reason object creation has larger overhead than write accesses is that it is compounded by

write accesses because object creation requires write accesses. The impact on read accesses and method calls was negligible. Any overhead we add to these critical aspects may be too much if we want objects to be competitive with other traditional data structures. Therefore, we have to be willing to accept some overhead with the benefit of convenient persistence support or find an alternative way of supporting persistence otherwise. If MetaStore were to run on Lucid CLOS, we would expect object creation to be about 4 instead of 13 times slower than the transient case, and other aspects to be about the same as they are in PCL.

Substantial improvements can be made on the performance of MetaStore by changing our design somewhat by: (i) eliminating the overhead of `:around` methods by replacing them by macros, and (ii) eliminating case analysis on slot accesses as described in section 5.4.7. With these changes, object creation and read accesses were as fast as they were with transient objects. Write accesses were still about five times slower than they were in the transient case. The overhead on write accesses could be eliminated by requiring user programs to inform MetaStore on dirty bits and sacrifice on the sharing of persistent structured data.

Supporting persistence grain size at the slot level was critical to the design of MetaStore. It made the virtual object memory scheme and sharing composite slot values by multiple objects possible. Because most objects are large due to large composite slot values in object-intensive applications, being able to flush out some of objects that are not likely to be accessed for a while is critical in maintaining the system performance at an acceptable level.

Another critical aspect of a persistent object system is how the database management features, including secondary storage management, are handled. We summarize notable aspects of them here:

- *Preloading:* A pure lazy loading of all persistent objects is too slow to be practical as shown in section 5.5.2. The batch-oriented load of an entire

model file at the beginning of each session is also not practical enough as we experienced with pre-MetaStore CDRS. A combination of these two seems most appropriate and is adopted in MetaStore. Preloading of certain objects in the batch-mode in sequential reads at the start of each session lets us start quickly and lazy loading of slot values and other objects helps us continue with only the data that are necessary to be loaded. With a graphical user interface-based application like CDRS, preloading of certain objects turns out to be critical because a user has to be able to see what is in a design to continue editing.

- *Queries:* Even with preloading of certain objects, user programs would have to load other objects, and several ad hoc means of querying the object base turns out to be adequate for a design environment like CDRS. It is not clear if a general purpose query language support is necessary for this kind of applications.

- *Indexing:* Indexes were used to load keyed or named objects efficiently on queries. Supporting indexes on these objects alone rather than on all objects were adequate because some objects are owned by others. Because there is no use loading the owned when the owner is not loaded, maintaining indexes for top level objects (owners) alone was adequate.

- *Transactions:* Because we chose to use a minimal degree of concurrency control, the most critical aspect was ensuring atomic updates in the object base. We used the shadow paging algorithm [51] and it was quite adequate for our purposes. The degree of sharing of models between users and the degree of simultaneous editing of the same model in CDRS are so small that locking a design file for concurrency control was again quite adequate. The nested transaction scheme was particularly useful for atomic updates of the object

base with incremental saves because of the long duration locking scheme used with checkout-locks.

- *Crash recovery:* The checkpoint mechanism along with shadow paging seems to provide adequate support for object-intensive applications. Incremental checkpoints reduce the overhead of checkpoints to a minimum and the use of checkpoints for *undo* operations would be quite useful for many application programmers who use MetaStore. *Undo* operations are critical in design environments with long transactions and it is very difficult and complicated to support in application programs without the help of a tool such as MetaStore. With the help of the nested transaction scheme, recoveries are done to the beginning of a nested transaction rather than to the beginning of a checkout transaction, thus reducing the chance of a large loss on a crash.

- *Object base garbage collection:* The shadow paging algorithm used for atomic updates of the object base and for the purpose of supporting crash recoveries and multiple versions generates a lot of garbage in model files. Collecting garbage in batch mode during off hours seems adequate because it can be done on many workstations in a distributed fashion. When a user is ready to go home, he would simply start the garbage collector on his machine. The collector then checks a file out of the object base and performs the collection. Many workstations can perform the collection as many can interface the object base.

- *Object clustering:* Object clustering is done to reduce the disk seek time during loading of objects. Our experience was quite disappointing because the benefit of clustering was insignificant. Because the majority of loading time was spent on something other than disk seeks, improvements made by clustering were negligible. The benefit of clustering objects are most noticeable when the

objects are small in size. In an application like CDRS, however, objects are large enough that the benefit of clustering is negligible. Improving the Lisp reader would be a good investment for a quick improvement on loading objects. Based on our experience with CDRS, a customized reader for a specific kind of data is about 30 times faster than the general reader supported by Lucid Common Lisp. Although the benefit would be quite noticeable, it is not very interesting and is not pursued in this research.

- *Version control:* A version is an attribute of a model file rather than that of an object in MetaStore. It was designed so because each model file is roughly what a user considers an "object." For example, a `door` of a car is an object to the user rather than a `curve` that makes up the `door`. Because a new version is created at the control of a user, we felt that supporting versions at the user's object level seemed right. Because of complex relationships among objects, treating the user's "object" as an object rather than an instance of a class as an object made many of our algorithms simpler. The version control and atomic updates of the object base are two such examples.

- *Multiple model files:* Because of the user's view of the object base as a collection of user level "objects," it was important to support editing or viewing multiple user "objects" in MetaStore. It is much like dealing with multiple flat structured databases. Some difficulties of editing multiple model files include: (i) when a new object is created, there is no good way of knowing which model the new object logically belongs to, and (ii) there is no simple way of dealing with object references that crosses a model file boundary. In MetaStore, we chose a rather simple scheme in which: (i) there is only one model file that is writable in a session, and (ii) a copying scheme is used rather than a sharing scheme to deal with object references that cross a model file boundary as described

in section 4.8. The choice of copying scheme was motivated by the fact that object clustering, deleting a model file, merging multiple model files into one, extracting a model file to send to a different site, and garbage collecting a model file would all be much simpler with the copying scheme than with the sharing scheme.

To monitor the amount of data in virtual memory, we included the virtual object memory in MetaStore. When many objects are loaded into virtual memory either by preloading, lazy loading, or queries, there is no way of reducing the amount of data in virtual memory other than by deleting them. However, depending on the semantics of a delete operation, deleting may not be the right thing to do. Thus, we introduced the notion of *flushing* an object. When an object is flushed, the space occupied by the persistable composite slot values of the object is released. We chose the *least recently used* flushing policy in MetaStore. Registering object accesses on each read or write access via the use of the metaobject protocol is expensive as shown in section 5.4.6. To reduce the cost of monitoring accesses, we may have to require user programs to tell MetaStore on some selective read or write accesses. When an object is repeatedly accessed in a tight loop, telling MetaStore only once would be enough, whereas MetaStore would register the access on every access if it was done by MetaStore with no user program's help as in the current implementation.

In designing a persistent object system we had to resolve many issues: some language related and others database system related. There were different possible solutions in all of these issues, and we had to make design choices on each of these. Design decisions were made based on practicality, generality, completeness, and elegance. Many decisions were, however, influenced by the experience with and requirements of a real world application that initially motivated this research, CDRS [26]. We feel that the decisions made in the design of MetaStore will be of

a significant value to the future research in this area because they were based on a production system being used by many users.

Based on the initial implementation of MetaStore, we also presented various benchmark results in sections 5.2 through 5.5 that could be used in making future design decisions for persistent object systems. The results are analyzed and summarized in the subsection titled "Remarks" in each of these sections. Rather than repeating the results here, we refer the readers to these subsections for details.

6.2 Future Work

Better stop short than fill it to the brim.
Lao Tsu, Tao Te Ching

Future work can take on different directions: in the direction of programming language design, database management, and/or implementation techniques. We suggest some possibilities in each of these areas.

Designing programming languages with the metaobject protocol concept is exciting and seems practical enough as is being demonstrated by the successful use of the Common Lisp Object System in many industrial, research, and academic environments. Although there were a few minor deficiencies with the current design of the metaobject protocol of CLOS and there were some efficiency concerns due to the loss of optimizations when some features of the protocol were used, our experience with the protocol was quite positive. The metaobject protocol idea is revolutionary in language design and it was almost perfect to use in extending CLOS with object persistence, which is a substantial task. We list some future directions in this area:

- We should investigate the possibility of extending the syntax and semantics of method combinations in object-oriented programming languages so that some specialized methods in a combination can optionally be omitted at the user's control. CLOS supports a simple ordering of `:before`, `:after`, `:around`, and

`primary` methods, and we should see if some of these can safely be omitted from the operational semantics without disturbing the generality. It is necessary to be able to skip an `:around` method, for example, in implementing MetaStore. This extension, if feasible, should be included in the protocol.

If this extension is not feasible, then we should support some way of performing one level of indirection on slot accesses of an object for the persistence extension.

- The current implementations of CLOS seem to be efficient enough to be competitive to other traditional data structures such as arrays, lists, and structures in some special contexts. We should pursue better implementation techniques for objects so that they can be competitive to other data structures in almost all contexts. By competitive we mean objects are *almost* as efficient as other data structures because some deficiency in speed can be made up for by the advantages of objects such as modular design and software reuse.

- We should also investigate the possibility of pushing the metaobject protocol idea down to the base language implementation level so that supporting the persistence of all the persistable data types in a language can easily be implemented.

In MetaStore we focused on persistence of passive data, i.e., the instances of classes. A natural extension to it would be to handle the evolution of not only the passive data but also the code that manipulates the data. The notion of class evolution in the context of persistent programming is an open research issue. One possible direction of research on class evolution would be doing it with the help of metaobject protocols.

In an application like CDRS which is implemented mostly in Common Lisp and CLOS, some parts of the system could be implemented in a more efficient language

such as C++ for better performance. Mecklenburg [34] deals with the problem of mixed language programming in an object-oriented programming environment in a general fashion by using an object specification language in a language independent way. Although less general but more pragmatic than the approach of [34], extending MetaStore to deal with foreign objects via the metaobject protocol of CLOS is a distinct possibility. Just as we introduced the slot level persistence with the metaobject protocol, we can allow a slot to be declared a type of a `:foreign` slot, which points to a foreign object, say a C++ object. With this scheme, we would have the notion of master and slave languages. Because Common Lisp with CLOS is considered a good language to use for user interface design and controlling the execution and C++ is considered good for the efficiency of computationally intensive routines, we could use Common Lisp with CLOS and the metaobject protocol as the master and C++ as the slave.[1]

We described ideal solutions as well as practical solutions for dealing with dirty bits and shared structures in this dissertation. Based on our analyses without actually implementing different possibilities, we chose the practical solutions for the initial implementation of MetaStore. We could investigate these issues further by actually implementing different possible solutions so that we can make concrete comparisons.

Once a real world application like CDRS is fully ported to MetaStore, we can tune the system with the virtual object memory by coming up with good heuristics for the high water mark and the size of the buffer used by the virtual object memory algorithm.

So far, we have looked at the possibilities of future work in the context of programming language design. We now do the same in the context of database management. In general, a simple scheme for each of various issues related to

[1] A comment by one of the local Lispers: "I like it. C++ people would hate it ⌣."

the database management was adopted in MetaStore partly because it was good enough for the purpose of MetaStore supporting an application like CDRS and partly because elaborate schemes would be beyond the scope of this dissertation. More elaborate schemes can be investigated on some issues in the future and they include:

- *Distribution:* So far, we have not considered MetaStore in a distributed environment, and it is only natural to think about extending it to be a distributed persistent object system.

- *Version control:* In the current design of MetaStore, a version is an attribute of a model file. For more elaborate control of versions, we can easily extend it so that a version is an attribute of objects. The design of MetaStore is flexible enough that this extension would be quite smooth to make.

- *Concurrency control:* Concurrency control was done at the model file level. We could extend it to the individual object level.

- *Query language:* We could certainly investigate the possibility of introducing a query language to MetaStore rather than using several ad hoc means of querying the object base. We are not quite convinced that it is necessary to do so though.

- *Object clustering:* Although the benefit of object clustering for applications like CDRS was minimal due to the large size of objects, it might be useful for the applications that use objects of small sizes. It is stated in Mneme [37], for example, that the "typical size" of objects for languages such as CLU [28,29], Smalltalk-80 [16], and Trellis [42] appears to be in the range of 30 to 50 bytes.

- *Multiple models:* When multiple models are opened in an editing session, a copying scheme is used in the current design of MetaStore in dealing with one

object making references to other objects in different model files. Investigating the cost of supporting the sharing scheme would be an interesting research because dealing with multiple model files in MetaStore is similar to dealing with multiple flat structured databases, which could be of interest to a wider database research community.

- *Object base garbage collection:* In the current design of MetaStore, garbage collections are done statically. Investigating the cost of supporting dynamic garbage collection would be a possibility for future work.

- *Object base file format:* For the performance and compatibility reasons, saving objects in a standard binary file format would be useful for an application like CDRS.

As described in section 2.2, there were some issues that MetaStore did not include in its design although they are included in "The Object-Oriented Database System Manifesto" [3]. Perhaps, a persistent object system like MetaStore should ultimately be compatible to something like this manifesto.

Before CDRS is ported to MetaStore, we propose some modifications to the design of MetaStore:

- Eliminating the use of `:around` methods in the critical aspects of MetaStore, namely, the ones that affect object creation and accesses. This is only due to the inefficient implementations of the object systems currently available to us. Using macros as we described in section 5.4.7 is a possibility.

- Eliminating the case analysis on write slot accesses by adding a phole at object creation time to each persistable slot that is declared to be composite. Thus, we would add `:composite` as a new slot option as we did with `:transient`. If a slot is declared `:composite`, then it is assumed to be persistable. Space

occupied by pholes will be a factor to consider, but we can reduce the size of pholes depending on what we decide to do with dirty bits and slot value sharing.

- Not handling dirty bits on slot accesses. Let user programs explicitly inform MetaStore of dirty bits. Programming errors can be detected by a tool provided by MetaStore.

- Possibly not supporting persistence for structure sharing even on the composite slot values. Because supporting structure sharing in MetaStore adds extra cost to each write access to objects, this may be a reasonable sacrifice to absorb based on our experience with CDRS. We have not supported persistence for shared structures in pre-MetaStore CDRS, and it has not been a big issue for the application programmers.

If we port MetaStore to Lucid CLOS with these changes to the current design, the overhead of MetaStore on objects will be minimal and quite acceptable. Porting CDRS to MetaStore will be done when the design and implementation of the extension of MetaStore to handle foreign objects (C++) are complete.

Persistence is an inherent feature of a programming system. It is as inherent a feature as garbage collection is in a language system. A language system will ultimately have to support it as a part of the base language implementation or provide a mechanism via which persistence implementor can access the base language internals. A metaobject protocol is one such mechanism possible if we implement the persistable data types in a language as objects, and we should investigate the protocol at a lower level as a possible, ultimate solution for providing flexible hooks to the language internals for persistence implementors. Persistence we support in a language system has to be orthogonal to the data types in the language and also has to support a way of specifying selective persistence: selective in the sense that only

certain instances of a given type is persistable. Persistence via inheritance is a good way of supporting selective persistence. Atomic data types are not only inefficient both in overhead and in lost concurrency opportunities [54], but also too low a level to support persistence because of interdependence among persistable instances. Therefore, some kind of high level concurrency control is needed to manage the dependence among instances of data types. Persistence is inherently a low level feature in a language system, and yet requires a high level control due to this interdependence. Use of dirty bits as a mechanism to handle the interdependence as we did in MetaStore might be good enough with more flexible access to the language internals.

References

[1] Abelson, H., and Sussman, G. J., with Sussman, J. *Structure and Interpretation of Computer Programs.* The MIT Press, 1985.

[2] Archibald, J. L., and Yakemovic, K. C. B., Eds. Workshop Report on Reflection and Metalevel Architectures in Object-Oriented Programming, OOPSLA/ECOOP (1990).

[3] Atkinson, M. P., Bancilhon, F., DeWitt, D., Dittrich, K., Maier, D., and Zdonik, S. The object-oriented database system manifesto. In *Proc. of the 1st International Conference on Deductive and Object-Oriented Databases* (1989).

[4] Bancilhon, F., and Maier, D. Multilanguage object-oriented systems: new answer to old database problems? In *Programming of Future Generation Computers II,* K. Fuchi and L. Kott, Eds, Elsevier Science Publishers B.V. (North-Holland), 1988.

[5] Banerjee, J., Kim, W., and Kim, K. C. Queries in object-oriented databases. MCC Tech. Rep. DB-188-87, 1987.

[6] Beeri, C., Bernstein, P. A., Goodman, N., Lai, M. Y., and Shasha, D. E. A concurrency control theory for nested transactions. In *Proc. of the Second ACM Symposium on Principles of Database Systems* (1983).

[7] Bobrow, D. G., and Stefik, M. *The Loops Manual.* Intelligent Systems Laboratory, Xerox PARC, 1983.

[8] Bobrow, D. G., DeMichiel, L., Gabriel, R. P., Kiczales, G., Moon, D., and Keene, S. E. *The Common Lisp Object System Specification:* Chapters 1 and 2. Tech. Rep. 88-002R, X3J13 Standards Committee Document, 1988.

[9] Bracha, G. The programming language Jigsaw: mixins, modularity, and multiple inheritance. Ph.D. dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

[10] Chou, H.-T., and Kim, W. Versions and change notification in an object-oriented database system. In *Proc. of the 25th ACM/IEEE Design Automation Conference* (June 1988).

[11] Cockshott, W. P., Atkinson, M. P., Chisholm, K. J., Bailey, P. J., and Morrison, R. Persistent object management systems. In *Software – Practice and Experience, Vol. 14* (1984).

[12] Cockshott, W. P. Addressing mechanisms and persistent programming. In *Data Types and Persistence,* M. P. Atkinson, P. Buneman, and R. Morrison, Eds, Springer-Verlag, 1985.

[13] Cockshott, W. P. *PS-Algol Implementations: Applications in Persistent Object-Oriented Programming.* Ellis Horwood Limited, 1990.

[14] Cook, W. R. A denotational semantics of inheritance. Ph.D. dissertation, Brown Univ., 1989.

[15] Copeland, G., and Maier, D. Making Smalltalk a database system. In *Proc. of the ACM SIGMOD International Conference on Management of Data* (June 1984). *ACM SIGMOD Record 14, 2* (1984).

[16] Goldberg, A., and Robson, D. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[17] Halstead, R. H. Multilisp: a language for concurrent symbolic computation. In *ACM Transactions on Programming Languages and Systems* (October 1985).

[18] Jensen, K., and Wirth, N. *Pascal User Manual and Report,* Second edition, Springer-Verlag, 1978.

[19] Kaehler T. Virtual memory for an object-oriented language. In *Byte* (August 1981).

[20] Kaehler T., and Krasner, G. LOOM–large object-oriented memory for Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice,* G. Krasner, Ed., Addison-Wesley, 1982.

[21] Keene, S. E. *Object-Oriented Programming in Common Lisp: A Programmer's Guide to CLOS.* Addison-Wesley, 1989.

[22] Kernighan, B. W., and Ritchie, D. M. *The C Programming Language.* Prentice-Hall, 1978.

[23] Kiczales, G., Rivières, J., and Bobrow, D. G. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[24] Kim, W., Garza, J. F., Ballou, N., and Woelk, D. Architecture of the ORION next-generation database system. MCC Tech. Rep. ACT-OODS-315-89, 1989.

[25] KIM, W., KIM, K.-C., AND DALE, A. Indexing techniques for object-oriented databases. In *Object-Oriented Concepts, Databases, and Applications,* W. Kim and F. Lochovsky, Eds., ACM Press, 1989.

[26] LEE, A. H. An object-oriented programming approach to geometric modeling. In *Proc. of Evans & Sutherland Technical Retreat,* Ocho Rio, Jamaica (1989).

[27] LIPPMAN, S. B. *C++ Primer,* 2nd Ed., Addison-Wesley, 1991.

[28] LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, C., SCHEIFLER, R., AND SNYDER, A. *CLU Reference Manual.* Springer-Verlag, 1981.

[29] LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. In *Communications of ACM 20, 8* (August, 1977).

[30] *Lucid Common Lisp/MIPS Version 4.0, Advanced User's Guide,* Lucid, Inc., 1990.

[31] MAES, P., AND NARDI, D., Eds. *Meta-Level Architectures and Reflection,* North-Holland, 1988.

[32] MAIER, D. Indexing in an object-oriented DBMS. Tech. Rep. CS/E-86-006, Dept. of Computer Science and Engineering, Oregon Graduate Institute of Science and Technology, 1986.

[33] MAIER, D., AND STEIN, J. Development and implementation of an object-oriented DBMS. In *Research Directions in Object-Oriented Programming,* The MIT Press, 1987.

[34] MECKLENBURG, R. W. Towards a language independent object system. Ph.D. dissertation, Univ. of Utah, 1991.

[35] MILLER, J. S. MultiScheme: a parallel processing system based on MIT Scheme. Ph.D. dissertation, Massachusetts Institute of Technology, 1987.

[36] MORRISON, R. *S-Algol Language Reference Manual.* CS-79-1, University of St. Andrews, 1979.

[37] MOSS, J. E. B. Design of the Mneme persistent object store. In *ACM Transactions on Information Systems, Vol. 8, No. 2* (April 1990).

[38] PAEPCKE, A. PCLOS: a flexible implementation of CLOS persistence. In *Proc. of the European Conference on Object-Oriented Programming,* S. Gjessing and K. Nygaard, Eds. Lecture Notes in Computer Science, Springer-Verlag (1988).

[39] Paepcke, A. PCLOS: a critical review. In *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (1989).

[40] Paepcke, A. PCLOS: stress testing CLOS: experiencing the Metaobject Protocol. In *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (1990).

[41] Rodriguez, L. H., Jr. Coarse-grained parallelism using metaobject protocols. M.S. Thesis, Massachusetts Institute of Technology, 1991. (Also available as Tech. Rep. SSL-91-06, Xerox Palo Alto Research Center, 1991).

[42] Schaffert, C., Cooper, T., Bullis, B., Kilian, M., and Wilpolt, C. An introduction to Trellis/Owl. In *Proc. of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications* (September 1986). *ACM SIGPLAN Notice, 21, 11* (1986).

[43] Schwarz, P. Transactions on typed objects. Ph.D. dissertation, Carnegie-Mellon University, 1984.

[44] Shannon, K., and Snodgrass, R. Semantic clustering. In *Proc. of the Fourth International Workshop on Persistent Object Systems,* A. Dearle, G. M. Shaw, and S. B. Zdonik, Eds., Morgan Kaufmann Publishers, Inc. (1990).

[45] Smith, B. C. *Interim 3-LISP Reference Manual,* Xerox PARC Report CIS-nn, 1984.

[46] Steele, G. L., Jr. *Common Lisp: The Language,* Second edition, Digital Press, 1990.

[47] Stonebraker, M., Wong, E., Kreps, P., and Held, G. The design and implementation of INGRES. In *ACM Trans. on Database Systems, 1:3* (1974).

[48] Stroustrup, B. *The C++ Programming Language,* Second edition. Addison-Wesley, 1987.

[49] Teitelman, W. *Interlisp Reference Manual,* Tech. Rep., Xerox Palo Alto Research Center, 1975.

[50] Thatte, S. M. Report on the object-oriented database workshop. In *Addendum to the Proceedings of Object-Oriented Programming Systems, Languages, and Applications* (1987).

[51] Traiger, I. L. Virtual memory management for database systems. In *ACM Operating Systems Review, 16* (October 1982).

[52] Ullman, J. D. *Principles of Database and Knowledge-Base Systems, Vol. I and II.* Computer Science Press, 1988.

[53] Ungar, D. M. *The Design and Evaluation of a High Performance Smalltalk System.* The MIT Press, 1987.

[54] Weihl, W. E. Linguistic support for atomic data types. In *ACM Transactions on Programming Languages and Systems, Vol. 12, No. 2* (April 1990).

[55] Zdonik, S. B. Version management in an object-oriented database. In *Proc. of the International Workshop on Advanced Programming Environments* (1986).

[56] Zdonik, S. B., and Maier, D. Fundamentals of object-oriented databases. In *Readings in Object-Oriented Database Systems*, S. B. Zdonik and D. Maier, Eds., Morgan Kaufmann, 1990.

So was ended all the work.
I Kings 7: 51