Metamusing on Object Persistence

Arthur H. Lee
Joseph L. Zachary

UUCS-92-028

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

July 15, 1992

Abstract

The need to "open up languages" has led to object-oriented programming languages with object-oriented implementations. By encapsulating the fundamental aspects of a language semantics within a set of default classes and giving the programmer the flexibility of deriving new versions of these base classes, a language whose semantics can be tailored to the needs of individual programmers can be provided. The degree of success in designing a language in this way to achieve flexibility and efficiency simultaneously is an open question. The Common Lisp Object System is designed with these techniques and we address this question by reporting our experience with the CLOS metaobject protocol in incorporating support for persistence into CLOS. For many aspects of our implementation we found that the metaobject protocol was perfectly suitable. In other cases we had to variously extend the protocol, pay an unacceptable performance penalty, or modify the language implementation directly. Based on our experience we propose some improvements to the protocol. We also present some performance measurements that reveal the need for improved language implementation techniques.

Contents

1 Introduction

The need to "open up languages" has led to object-oriented programming languages implemented with the help of reflective and object-oriented techniques. By encapsulating the fundamental aspects of a language semantics within a set of default classes and giving the programmer the flexibility of deriving new versions of these base classes, a language whose semantics can be tailored to the needs of individual programmers can be provided. The process of modifying language semantics in this way is called *metaprogramming.*

Whether designing a language in this way achieves flexibility and efficiency simultaneously is an open question. The Common Lisp Object System (CLOS) [BDG88] is designed with these techniques, thus providing for metaprogramming via its *metaobject protocol* [KRB91]. In this paper we address this question by reporting our experience with using the metaobject protocol to incorporate support for persistent objects into CLOS.

Our experiment investigated if we could obtain a version of CLOS with persistence to which we could easily port a commercial CAD system already written in CLOS. We originally wanted to modify CLOS strictly via the metaobject protocol, so that no changes to the compiler or run-time system would be required. Although we ultimately compromised slightly on this point and devoted considerable engineering effort to the implementation, the final product, although fully expressive, was judged too inefficient for commercial use, thus requiring changes in the way we use the metaobject protocol.

In this paper we highlight the strengths and weaknesses exhibited by the CLOS metaobject protocol during our experiment. Extending CLOS with object persistence is no small undertaking, and the metaobject protocol is quite general, so we are convinced that our experience is relevant to metaprogramming in general. For many aspects of the implementation we found that the metaobject protocol was perfectly suitable. In other cases we had to choose among paying a large performance penalty, extending the protocol, and bypassing the protocol entirely and modify the language implementation directly. Based on our experience we propose some improvements to the protocol. We also present some performance measurements that reveal the need for improved language implementation techniques.

The remainder of this paper is organized as follows. In section 2 we briefly describe the CLOS metaobject protocol followed by an application that reveals the problem of object persistence in section 3. In section 4 we describe our approach to adding persistence via metaprogramming followed by four particular extensions we made to CLOS that exhibited strengths and weaknesses of the CLOS metaobject protocol in section 5. In section 6 we present some performance measurements, and some improvements to the protocol are proposed in section 7. After we survey other uses of metaprogramming in section 8, we conclude in section 9.

2 The CLOS metaobject protocol

The design of CLOS is "open" in the sense that some aspects of implementation internals are accessible by application programmers. This object-oriented implementation is made possible by combining reflective techniques and object-oriented techniques in language design [KRB91]. The CLOS metaobject protocol designed in this way allows users to alter the semantics of the language by using the standard object-oriented techniques of subclassing and specialization.

In the design of CLOS, the basic elements of the programming language—classes, slots, methods, generic functions, and method combinations—are 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—thus the term metaobject protocol. For each kind of metaobject a default class is created, which delineates the default behavior of the language in the form of methods.

In the metaobject protocol, for example, the meaning of object instantiation is implemented by a small number of generic functions. These semantics can be changed by defining a subclass in which these generic functions are specialized. In doing this, the user is making an incremental adjustment to the meaning of the language. Most aspects of the language's behavior and implementation remain unchanged, with just the semantics of instances being altered.

3 CDRS: An object-intensive application

Conceptual Design and Rendering System (CDRS) [Lee89] is a geometric CAD modeler that is used by designers in a dozen major automotive companies worldwide, and our work was initially motivated by the problems of object persistence encountered with CDRS. It is an *object-intensive* application written mostly in Common Lisp [Ste90] as extended by CLOS. A typical model manipulated by an object-intensive application such as CDRS is characterized by:

- containing tens of thousands of objects that may not all fit in virtual memory,
- showing a wide variation in the sizes of objects,
- requiring complex data structures within objects, and
- containing rich relationships (both semantic and structural) among objects.

Supporting object persistence in object-intensive applications is particularly difficult because of these characteristics.

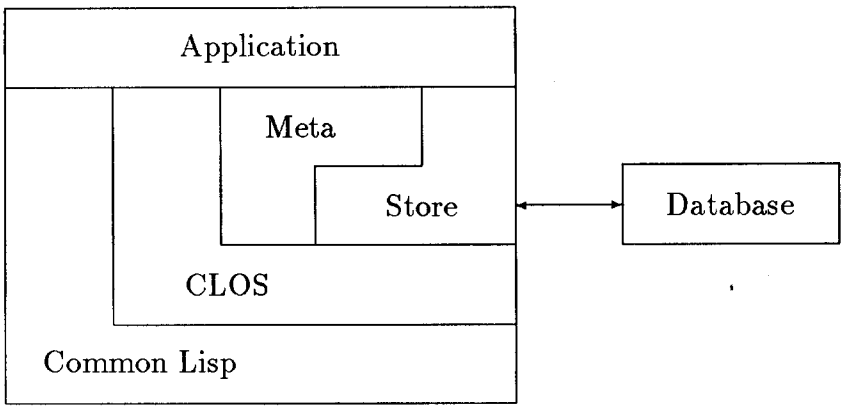

Figure 1: MetaStore architecture

CDRS uses a naive file-based, batch-oriented approach to object persistence that has proven to be ill-suited for the many complex objects found in object-intensive applications [Lee92]. All objects in a design session are saved to a file at the end of a modeling session and are reloaded at the beginning of the next session. This approach requires a huge amount of virtual memory, frequent large garbage collections, and a long time to load and save models. For example, CDRS usually uses 500 megabytes of swap space, requires up to 128 megabytes of main memory, and spends almost 30 minutes loading or saving a typical model. Users are usually noncomputer professionals and tend to save models frequently in fear of losing models due to reliability problems. Reducing user waiting time is critical for the success of an object-intensive system like CDRS. Users claim that anything more than three minutes of waiting for file operations is too much in a production environment.

4 MetaStore

We implemented a persistent object system called MetaStore to address these problems by adding the notion of persistent objects to CLOS. MetaStore has two major components: the language extension portion implemented via the CLOS metaobject protocol, and the database management portion that provides a persistent object store as shown in Figure 1. In this figure, `Store` is the persistent object store and `Meta` is the language extension portion. We are concerned in this paper with the language extension component and the degree to which the metaobject protocol facilitated and frustrated our efforts. For a complete discussion of the resulting system see [Lee92].

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 saves are done incrementally (only modified objects are written to disk) and loads are done on demand (ob-

jects are loaded as they are needed). This approach amortizes the cost of saving and loading models over the entire design session, thus reducing the user waiting time. MetaStore also maintains a virtual object space within virtual memory. As the object space fills, it writes the least recently used persistent objects to disk and makes their virtual images available for garbage collection. To support this, we had to make substantial modifications to the way objects are represented and manipulated by CLOS. The question that concerned us throughout design and implementation was whether the overhead imposed by the metaprogramming would be too costly. It was.

In this section after an overview of MetaStore, we describe the programmer's view of MetaStore; how the metaobject protocol is used; and four different kinds of extensions we made to CLOS.

4.1 Overview

Focusing only on the aspects that require extensions to the object system, we give an overview of MetaStore design. It is given by describing the life cycle of a persistent object.

We distinguish between persistable and persistent objects. 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 2 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). The OID of the object `O1` in Figure 2 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. Address translations in MetaStore are done by the pointer swizzling technique [Lee92].

Upon creation of a persistable object, an intermediary data structure called a *phole* is added between a persistable object and each of its persistable composite slot values. In Figure 2, 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

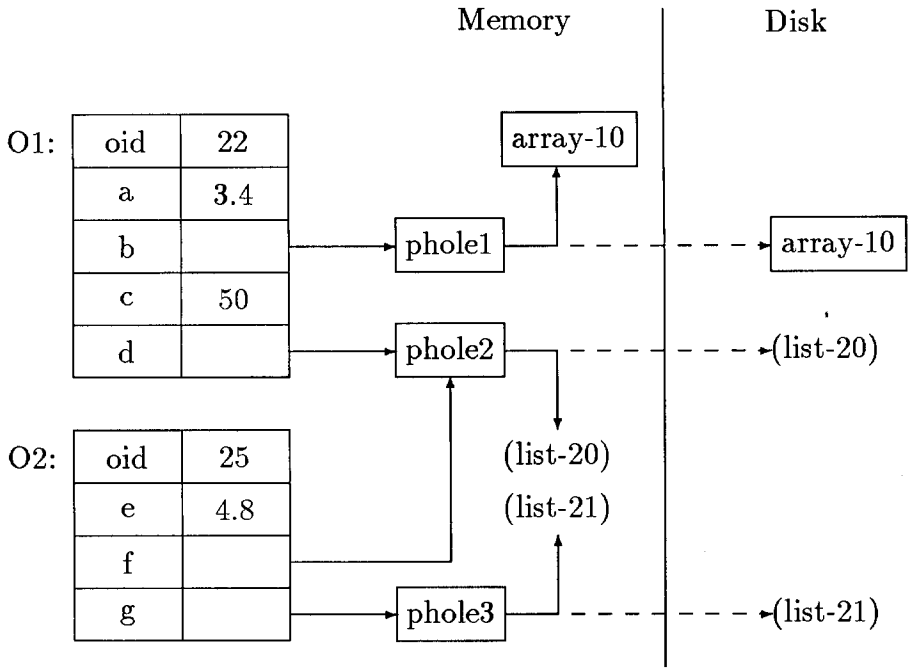

Figure 2: Persistent objects in MetaStore

- Supporting persistence 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 transient object is an instance of a class that does not inherit the persistent class. 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.

At some later time, a repeatedly accessed persistable object is saved if dirty when there is a request for a save, 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. When another process similar to the one that initially created the object loads the saved object, 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. Thus, the life of a loaded object continues as if it were a newly created one.

An object is usually loaded as a husk. 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.[1] In Figure 2, 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 [Lee92]) are flushed to free up some space in order to improve the system 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 is never flushed.

In Figure 2, `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.

4.2 Programmer's view of MetaStore

The application programmer's interface to MetaStore is kept to minimum. 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. For example, in a user defined persistable class `student` below:

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

[1] PCLOS [Pae90] also uses the notion of a husk object, but with a different meaning. A husk in PCLOS is a placeholder for an object and 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.

two things are added to deal with persistence. First, the mandatory keyword `:METACLASS` in the line labeled (2) links a source program, the class definition for **student**, and the meta-code that defines the behavior of class metaobject. This line declares that the class **student** is persistable, thus making all of its instances persistable objects. Second, the optional slot option `:TRANSIENT` in the line labeled (1) declares that the slot **hobby** is not persistable, thus making its value nonpersistable even though the rest of the object is.

4.3 Extensions required for persistence

Extending CLOS with object persistence is a substantial undertaking, and it requires dealing with many aspects of the object system. It requires modifying how objects are represented, adding more information to each object, modifying how some system provided methods, and setting up some conventions that application programs must abide by. Some of them are listed below:

- Adding an OID to each object.
- Separating the transient from the persistable.
- Keeping metaobject specific administrative information locally at metaobjects.
- Adding and removing *pholes* at the time of a creation of or an access to persistable objects.
- 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.
- Handling the persistence of shared structures.
- Shadowing some system classes belonging to the metaobject protocol.
- Realizing the notion of a *husk*.

In MetaStore, we handle these via the metaobject protocol of CLOS without any help from the compiler or run-time support system. We describe how some of these are achieved in the next section.

5 Four kinds of extensions

We used the metaobject protocol to make all the modifications listed in section 4.3 to CLOS. In this section we discuss four of these modifications, which illustrate the kinds of situations in which the metaobject protocol is and is not applicable.

5.1 Structure and behavior of objects

The metaobject protocol is ideal for language extensions that involve modifications to the structure of objects or simple changes to their behavior. Changes to other kinds of data structures (such as arrays) are much more difficult and usually require some help from user programs or the base language implementation level.

In addition to the user-defined slots, we maintain several other slots in each object. A unique object identifier and a "dirty bit" which flags unsaved objects are included. Since the majority of the objects in a typical model are not intended to be saved, we must be careful to distinguish between transient and persistable objects. This burden is borne by the programmer, who must choose between defining classes relative to the standard class meta-class (in the case of transience) or the derived class meta-class (in the case of persistence).

Each read or write access to a persistable object is intercepted so that appropriate actions can be carried out. For example, a read access may result in a composite slot being loaded from disk, and a write access results in a dirty bit being set.

We were able to make all these modifications by changing the appropriate methods via inheritance through the metaobject protocol, and we present some key ones in the following subsections.

5.1.1 Persistent class metaobject class

First, a *specialized class metaobject class* [KRB91], `persistent-metaclass`, is defined as a subclass of the standard metaobject class, `standard-class`:

```
(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 of this metalevel class, a metaobject say `M`, defines the behavior of a user level object, say an instance of the user class `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 sections below.

5.1.2 Object identity and creating persistable objects and husks

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 it is declared persistable. (ii) The necessary information for the virtual object memory is recorded. (iii) The necessary information for handling structure sharing is recorded.

To support object persistence and sharing, each object must have a unique identity (ID). This is a persistent, logical ID as opposed to a volatile ID, which is the virtual address of an object. In MetaStore, a reference in virtual memory from one object to another is done via a volatile ID, and the persistent ID is used only for addressing persistent objects. An OID is added to a persistable object when it is first created and is encapsulated. An application program is given read access to an OID, but not a write access.

We also support the notion of a husk to support persistence at the slot level. Creating a husk is handled somewhat differently from creating objects. When an object is created via a call to `make-instance`, it happens in two steps:

- An instance is allocated by the protocol routine `allocate-instance`.
- 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 replacing the slot values of `O1` with the saved values to fully recover the original state of the saved object. This wastes 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.

5.1.3 Persistence via inheritance

Persistence of program data in MetaStore is done via inheritance. Treating all the data in a program as persistable, as is done in PS-Algol [Coc90], 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 ends up being saved. `persistent-root-class` is defined in MetaStore and is made a superclass of each persistable user class. This is done at `initialize-instance` phase of class definition via the metaobject protocol [KRB91]. 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.

The purpose of `persistent-root-class` is two-fold. 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-obj-or-slot-id)
                               :mid *current-model-id*))
   (dirtyp :initform t :initarg :dirtyp :accessor dirtyp)
   )
  (:metaclass persistent-metaclass))
```

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 whose 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. A user program would either overwrite the default method or define an `:after` method. 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. 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 determines that an object is to be flushed out.
- It handles encoding of objects for saving and decoding of them during loading. Address translations are done as a part of this process.

5.1.4 Accessing objects

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

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`.

A write access is more complicated than a read access. On a write access, 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[2] slot is accessed, then set the dirty bit of the object accessed. In addition, if the new value 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. 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)`.

5.1.5 Persistent *slot-definition* metaobject class

When 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

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

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*, available for use.

There is one more thing to take care of. A rule for inheritance regarding transience of slots must be enforced. A slot is treated as transient only if all classes in the inheritance chain that define a slot with that name has the same declaration as was done in [Pae91]. This is done at the time effective slot definitions are computed by the generic function, `compute-effective-slot-definition`.

5.2 Indirection on slot access

MetaStore supports persistence at the slot level, and Common Lisp allows structure sharing. These two facts required us to maintain one level of indirection for each persistable composite slot. The contents of a persistable composite slot is a pointer to a phole, which (among other things) contains a pointer to the composite value.

When a user program issues a `slot-value` call to a persistable slot, MetaStore must follow pointers and return the value stored in the phole. The implementation of MetaStore, however, must sometimes directly obtain the phole via the same call. Supporting this behavior was not entirely straightforward.

The solution requires providing two different semantics for the method (`slot-value`) depending upon where and for what purpose it is called. The metaobject protocol provides no support for this. Solving this problem involved making minor modifications to the protocol. Specifically, we had to add an extra method for accessing slot values at the protocol implementation level. This kind of problem could be avoided by a minor change to the design of the protocol.

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

5.3 Maintaining dirty bits

When an object is requested to be persistent, only dirty (modified) persistable objects and slot values are ever saved to disk. Because the smallest grain size of persistence is the composite slot, each persistable object and persistable composite slot value has its own dirty

bit. We will concern ourselves here with composite slots. The dirty bit of a composite slot is kept in its phole.

The dirty bit of an object or composite slot must be set whenever a write access is made. Doing this via the metaobject protocol proved difficult. Performing a write upon a slot value via the public interface of the containing object, i.e., via `(setf slot-value)`, poses no problem because a slot access via public interface is intercepted to maintain the dirty bits. The problem occurs when programmers obtain a slot value via a *read* access and then mutate that value outside the object system. The following code fragment 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 and locally bound to `arr1`. The array is then modified. However, since this modification is not made via the phole of `slot1`, the phole's dirty bit cannot be set. To make sure that the dirty bit is set, 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)` can be easily modified via the metaobject protocol to maintain dirty bits. However, requiring this extra call changes the semantics of CLOS.

An expensive solution that maintains dirty bits without any help from either the application program or the compiler is described in [Lee92] although we chose to implement a simpler solution along the lines suggested above that requires help from user programs for efficiency reasons.

5.4 Persistence of shared structures

Structured data in Common Lisp can be shared freely by variables and other structured data. This freedom adds much difficulty in supporting persistence of shared structures. We could not find any acceptably efficient solution within the metaobject protocol since it does not deal with structured data that are not objects. The central problem is that structures such as arrays and lists, unlike objects, cannot be given unique identifiers via the protocol.

To illustrate the problem, suppose a composite slot value, the array `a1` of the object `O1` in Figure 3, is ready to be saved. Also suppose that `a1` has another array, say `a2`, as one of its elements. Finally, suppose that a slot of another object `O2` also has `a2` as its value through a third array `a3`. Thus, `a2` is shared indirectly by `O1` and `O2`.

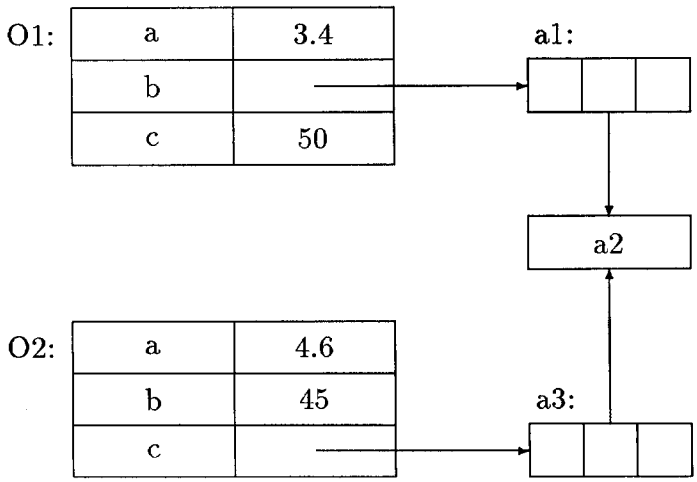

Figure 3: An array shared by two objects

This sort of sharing is perfectly legal in Common Lisp. Assuming that 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 again 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 [Lee92], a solution that, while inefficient, handles persistent shared structures entirely within the metaobject protocol is described. We chose, however, for efficiency reasons a different approach in MetaStore in which we require that composite structures be shared only at the slot level. This approach works because a phole can contain a unique identifier for a composite slot value.

6 Performance measurements

We present a performance comparison between objects and nonobject structured data. We also present performance measurements of our persistence implementation of MetaStore along with a few significant inefficient aspects of two implementations of the CLOS metaobject protocol, PCL [BS83] and Lucid CLOS [Luc90].

6.1 Objects vs. nonobject structured data

The advantages of using object-oriented programming in software engineering are well known, but these advantages come with extra costs. In this section, we analyze these costs by

examining the following aspects of CLOS: object creation, read and write accesses to objects, and method calls.

Transient objects are compared against data structures that are most commonly used in Common Lisp: arrays, structures, and lists. We used Lucid CLOS for these measurements since it is considered most efficient among many implementations of the metaobject protocol.

- *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 each having a simple integer value were created 100,000 times with the following results:

	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 2 to 4 times slower than creating other data structures. Class hierarchy does not seem to add a noticeable difference in either time or space. When different sizes of objects, structures, arrays, and lists were used, the relative differences between them were similar to the measurements shown above.

- *Read Access:* Because the context in which a read access to 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 within a function. These two cases were compared against the cases with structures, arrays, and lists. In each case, the 20th element was accessed 1,000,000 times. Memory consumptions on read accesses were negligible.

	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 an object within a method, case (i), was about 3 to 7 times slower than accesses to other data structures. Read accesses to an object outside a method, case (ii), however was about 10 to 200 times slower than accesses to other structures.

- *Write Access:* Similar experiment was tried on write accesses to objects and other structured data and the results were:

	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 an object within a method was about 3 to 19 times slower than accesses to other structures. Write accesses to an object outside a method, however, was about 2 to 90 times slower than accesses to other structures.

- *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 were 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 4 times slower than function calls.

In general using objects is more expensive than using 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 only some cases are optimized.

6.2 PCL and Lucid CLOS

We had originally intended to use the Lucid CLOS version of the metaobject protocol, but it did not have a complete implementation of slot-level metaobjects. As a result we were forced to use the PCL version, even though it is not an industrial-strength implementation. Since the performance measurements of MetaStore is based on PCL and our application CDRS will be running on Lucid CLOS, we present the comparison between PCL and Lucid so that we can predict the performance of MetaStore running on Lucid later.

We describe one significant inefficiency we found with PCL and Lucid CLOS before we present the measurements of MetaStore. In implementing MetaStore, a number of `:around` methods are defined to the protocol routines, the following three being the most notable: `make-instance` for creating objects, `slot-value-using-class` for read accessing an object, and `(setf slot-value-using-class)` for write accessing an object. To measure the cost of `:around` methods, an `:around` method was defined for each of these methods, whose body does nothing but calling `call-next-method`. The measurements in this section are based on PCL and we also present what we learned about Lucid CLOS where appropriate.

- *Creation:* Creating 1,000 objects showed:

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

  Other measurements showed that creating objects with `:around` methods was about 50 times slower than without in Lucid CLOS [Lee92]. In PCL we do not see as big a difference as with Lucid CLOS since creating objects in PCL without `:around` methods is already much slower than it is in Lucid CLOS.

- *Read Access:* Read accessing a slot of the "self" object 100,000 times showed:

	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.

- *Write Access:* Write accessing a slot of the "self" object 100,000 times showed:

	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.

The extent to which dummy `:around` methods compromise the performance of both the PCL and Lucid implementations of CLOS belies the claim of [KRB91] that the metaobject protocol is both elegant and efficient. Specializing default behavior by the use of `:around` methods is the most commonly used tool in the metaobject protocol.

Notice that in PCL we observed a 300 times slowdown when reading and writing slots, whereas in Lucid we observed a 50 times slowdown when creating objects. This is primarily due to the loss of optimization when `:around` methods are added. Neither of these figures can be tolerated in a commercial application. In fairness, we must emphasize that Lucid is in general far more efficient than PCL.

6.3 MetaStore

In this section we present measurements based on our implementation 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. It also includes the cost of metaobject classes, `:around` methods, and slot level persistence. Shared structures and virtual object memory are not included.

- *Creation:* The measurements were made while creating 1,000 objects.

	Time	*Bytes Consed*
Transient	3.40 sec	256,008
MetaStore Kernel	56.35	6,152,008
Ratio	16.57	24.04

The MetaStore kernel made creating objects about 16 times slower than creating objects without MetaStore. Creating objects in the MetaStore kernel used about 24 times more space than creating objects without MetaStore.

- *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
Transient	0.13 sec
MetaStore Kernel	35.62
Ratio	274.00

Read accesses in the MetaStore kernel was about 270 times slower than the normal transient read accesses.

- *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
Transient	0.11 sec
MetaStore Kernel	254.70
Ratio	2,315.45

Write accesses in the MetaStore kernel was over 2,000 times slower than the normal transient write accesses.

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 with the `:around` overhead removed, our measurements predict that 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. Although obviously not ideal, we believe that these overheads would be tolerable in CDRS, which is governed by the speed of user interaction.

7 Observations

In this section we offer our observations on why some of these modifications were difficult to make within the scope of the metaobject protocol. We also propose some short-term and long-term improvements to the metaobject protocol.

7.1 Abstraction mismatch

The metaobject protocol is designed to support language extensions that have to do with the structure or behavior of objects. As soon as we try to augment the language with a feature that is not a property of objects, the protocol is no longer sufficient.

As we have seen, supporting object persistence required some changes that were object related as well as others that were base language related. Dealing with the kinds of modifications described in sections 5.3 and 5.4 was difficult because there are no metaobjects corresponding to the nonobject structures; i.e., there is an abstraction mismatch.

CLOS can be viewed as having five levels of implementation ranging from high-level to low-level:

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

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 by devising extra mechanisms that required some help from user programs and/or the Common Lisp compiler.

7.2 Short-term improvements

Based on the experience of adding object persistence to CLOS in MetaStore, a few minor improvements are proposed here to the existing protocol. They are related to slot accessing as described in section 5.2. 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 in all respects to `slot-value-using-class`, which performs read accesses to slots. We sometimes want to use the default behavior of `slot-value-using-class` and at other times the changed behavior, and this new routine would always give the default behavior. This changed behavior is typically obtained by specializing the default method. With the current protocol, once we modify the behavior of a method this way, we cannot use the default behavior anymore.
- `(setf slot-value-using-class-direct)`:
  This is the dual of `slot-value-using-class-direct` for write accesses to slots.

When a method in CLOS is changed via specialization, 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 method is specialized, we are given the option of executing the original version alone. This is not an easy extension to support in general since it requires elaborate control over all the methods: `primary` methods, `:before` methods, `:after` methods, and `:around` methods. The `copy-as` operation of Jigsaw [Bra92] would solve this problem.

7.3 Long-term improvements

As discussed in sections 5.3 and 5.4, 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 useful to push the metaobject protocol idea further down to the level of the base language implementation. If we could support the metaobject protocol at the Common Lisp data type level or at the garbage collection level, the problems that we experienced in MetaStore (dirty bits and shared structures) could be easily solved. With this change, the protocol would allow more flexibility for extensions of the sort done in MetaStore. Perhaps, we would then call it the *metadata protocol* or *metatype protocol.*

8 Related work

Metaprogramming has been used in a variety of different applications by a number of researchers. Interestingly, none of these researchers reported the kinds of problems with metaprogramming that we have observed. We believe that this is because our application was much more ambitious than any of the others.

Rodriguez, with Anibus [Rod91,Rod92], investigated whether it was possible to use the metaobject protocol approach to develop an open parallelizing compiler in which new "marks" for parallelization could be defined in a simple and incremental way. Anibus has its own metaobject protocol. Unlike the metaobject protocol of CLOS, which is intended to be used in executing CLOS programs at run-time, that of Anibus was intended to be used to map a Scheme [SS75] program to an SPMD Scheme [Rod91] program at compile-time.

The authors in [ABB89] present three examples of how the CLOS metaobject protocol could be used. The first example shows how atomic objects could be implemented for concurrency control. Their second example outlines how persistence could be implemented through metalevel manipulations. This supports persistence at the object level. Their final example illustrates how graphic objects could be implemented via the protocol.

PCLOS [Pae90] is CLOS extended with persistence via the metaobject protocol of CLOS. PCLOS also supports persistence at the object level. It uses data base management systems for secondary storage management, which suffers from the phenomenon known as impedance mismatch [BM88,CM84].

Unlike PCLOS [Pae90] and the work described in [ABB89], MetaStore supports persistence at the slot level, which we believe is critical for the performance of a CAD application. Therefore, neither of these efforts experienced the kinds of problems that we described in section 5. Two other important differences are that MetaStore, unlike [Pae90] and [ABB89], supports incremental saves and persistence of shared structures.

9 Conclusion

The authors in [KRB91] state that they have simultaneously achieved elegance and efficiency by basing language design on metaobject protocols. Our experience of extending CLOS with persistence via the metaobject protocol shows that current implementations do not live up to this claim. The extent to which even dummy `:around` methods compromise the performance of both the PCL and Lucid implementations of CLOS makes them unacceptable for production programming. We observed up to 300 time slowdowns in PCL, and up to 50 time slowdowns in Lucid. Specializing default behavior by the use of `:around` methods is the most commonly used tool in the metaobject protocol.

Nevertheless, most of the extensions required to support object persistence were easily carried out in the metaobject protocol. We are convinced that the idea of metaprogramming is the right approach for applications such as ours. A few extensions to the protocol, coupled with better implementation techniques, would yield a uniquely useful tool. Adding persistence to CLOS is no small undertaking, and the metaobject protocol is quite general, so we are convinced that our experience is relevant to metaprogramming in general.

The protocol is sufficient to support language extensions as long as these extensions involve modifying or augmenting the structure and/or behavior of objects. Since most of what was required to extend CLOS with object persistence was related to objects, it was done easily via the protocol.

To support persistence at the slot level requires one level of indirection on slot accesses and the current protocol does not provide this feature. We were, however, able to deal with this by extending the protocol by adding two more interface routines. We propose that two new routines be added to the protocol so that one level of indirection on slot accesses can be done. An even better solution would be to extend the semantics of method combinations in CLOS in such a way that specialized methods such as an `:around` method can optionally be skipped during execution.

There were a few difficulties that we faced that could not be resolved with the protocol alone. They were maintaining dirty bits for composite values and handling persistence of shared nonobject structured data. They are not object related and do not belong to the domain of the metaobject protocol. Instead they belong to the base language implementation level, thus requiring help from the language compiler and the run-time support system. Since we could not get help from these either, we handled them with some help from application programs. Here, we propose that all persistable data types be implemented as objects so

that they can be included in the metaobject protocol. This would be a significant effort and we consider this a long-term goal.

Well implemented objects fare well against other structured data as we showed by comparing them against structures, arrays, and lists. Some features, however, were found to be very expensive to use in the current implementations of CLOS. `:around` method was one such example which slowed down slot accesses by a factor of 300 although accessing slot values is considered one of the most critical aspects of object performance.

References

[ABB89] G. Attardi, C. Bonini, M. R. Boscotrecase, T. Flagella, and M. Gaspari. Metalevel programming in CLOS. In *Proceedings of the European Conference on Object-Oriented Programming,* 1989.

[BM88] F. Bancilhon and D. Maier. Multilanguage object-oriented systems: new answer to old database problems? In *Programming of Future Generation Computers II,* K. Fuchi and L. Kott, editors. Elsevier Science Publishers B.V. (North-Holland), 1988.

[BDG88] D. G. Bobrow, L. DeMichiel, R. P. Gabriel, G. Kiczales, D. Moon, and S. E. Keene. *The Common Lisp Object System Specification:* Chapters 1 and 2. Technical report 88-002R, X3J13 Standards Committee Document, 1988.

[BS83] D. G. Bobrow and M. Stefik. *The Loops Manual.* Intelligent Systems Laboratory, Xerox Palo Alto Research Center, 1983.

[Bra92] G. Bracha. The programming language Jigsaw: mixins, modularity, and multiple inheritance. Ph.D. dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

[Coc90] W. P. Cockshott. *PS-Algol Implementations: Applications in Persistent Object-Oriented Programming.* Ellis Horwood Limited, 1990.

[CM84] G. Copeland and D. Maier. Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June 1984). *ACM SIGMOD Record 14, 2* (1984).

[KRB91] G. Kiczales, J. Rivières, and D. G. Bobrow. *The Art of the Metaobject Protocol.* The MIT Press, 1991.

[Lee89] A. H. Lee. An object-oriented programming approach to geometric modeling. In *Proceedings of Evans & Sutherland Technical Retreat,* Ocho Rio, Jamaica, 1989.

[Lee92] A. H. Lee. The persistent object system MetaStore: persistence via metaprogramming. Ph.D. dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

[Luc90] *Lucid Common Lisp/MIPS Version 4.0, Advanced User's Guide.* Lucid, Inc., 1990.

[Pae90] A. Paepcke. PCLOS: stress testing CLOS: experiencing the metaobject protocol. In *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications,* 1990.

[Pae91] A. Paepcke. User-Level Language Crafting—Introducing the CLOS Metaobject Protocol. Draft copy via personal communication, 1991.

[Rod91] L. H. Rodriguez, 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.)

[Rod92] L. H. Rodriguez, Jr. Towards a better understanding of compile-time metaobject protocols for parallelizing compilers. Submitted to *IMSA'92: International Workshop on Reflection and Meta-level Architecture,* Tokyo, Japan, 1992.

[SS75] G. L. Steele, Jr and G. J. Sussman. *Scheme: An interpreter for the extended lambda calculus.* Memo 349, MIT Artificial Intelligence Laboratory, 1975.

[Ste90] G. L. Steele, Jr. *Common Lisp: The Language,* Second edition. Digital Press, 1990.