Reflections on Metaprogramming

Arthur H. Lee and Joseph L. Zachary

***Abstract*—By encapsulating aspects of language semantics within a set of default classes and allowing the programmer to derive new versions, object-oriented languages whose semantics can be tailored to the needs of individual programmers have been provided. The degree to which such languages are simultaneously flexible and efficient is an open question. We describe our experience with using this technique to incorporate transparent support for persistence into the Common Lisp Object System via its metaobject protocol, an open implementation based on reflection. For many aspects of our implementation the metaobject protocol was perfectly suitable. In other cases we had to choose among extending the protocol, requiring the application programmer to employ special idioms, and tolerating a large performance penalty. Based on our experience we evaluate the metaobject protocol, propose some improvements and extensions, and present performance measurements that reveal the need for improved language implementation techniques.**

***Index Terms*—CLOS, metaobject protocol, object-oriented programming languages, object persistence, open implementation.**

I. Introduction

The spread of object-oriented technology has led to object-oriented programming languages with object-oriented implementations. This has afforded a natural way of "opening up" language implementations through the use of reflective programming techniques. By encapsulating fundamental aspects of a language's semantics within a set of default classes, and then giving the programmer the power to derive new versions of these classes, a language results whose semantics can be tailored to the needs of individual programmers. The process of modifying language semantics in this way is a form of *metaprogramming.*

The degree to which such languages are simultaneously flexible and efficient is an open question. The Common Lisp Object System (CLOS) [3], [27] is designed with an open implementation [10], [11] and provides for metaprogramming via its *metaobject protocol* [13]. In this paper we address the flexibility/efficiency question by reporting our experience with using the metaobject protocol to incorporate support for persistent objects into CLOS.

The goal of our experiment was two-fold:

- to evaluate the CLOS metaobject protocol, the utility of metaprogramming, and the value of open implementation as a language design tool; and
- to see if we could obtain a version of CLOS with persistence to which we could easily port a commercial CAD system already written in CLOS, thus exploiting transparent persistence.

We originally wanted to modify CLOS strictly via the metaobject protocol, so that no changes to the underlying language 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.

Our intent in this paper is to 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 extending the protocol, requiring the application programmer to employ special idioms, and tolerating a large performance penalty. Based on our experience we propose some improvements to the protocol and present performance measurements that reveal the need for improved language implementation techniques.

We begin in Section II by motivating the idea behind open implementations in general and the CLOS metaobject protocol in particular. We continue in Section III by describing the application that motivated our work. In Section IV we discuss how we added persistence via metaprogramming, and we analyze our experience with the CLOS metaobject protocol in Section V. We present some performance measurements in Section VI and suggest improvements to the protocol in Section VII. After surveying other uses of metaprogramming in Section VIII, we conclude in Section IX.

II. The CLOS Metaobject Protocol: An Open Implementation

Traditionally, black box abstraction has been used to control the complexity of a software module by exposing its functionality but hiding its implementation. Recently, however, researchers have explored a different kind of modularity called *open implementation* [13], [10], [11], [12].

Under black box abstraction, the implementation decisions encapsulated within a module are generally made in the light of assumptions about the way in which the client will ultimately use the module. Kiczales [11] terms such decisions, necessarily made in the face of incomplete information, *mapping decisions.* A *mapping conflict* occurs when the assumptions made by the implementor work at cross purposes to an eventual client's actual needs. Clients typically resolve map-

Manuscript received September 1993; revised March 1995.

A.H. Lee is with the Computer Science Department, Korea University, Seoul 136-701 Korea; e-mail: alee@psl.korea.ac.kr.

J.L. Zachary is with the Department of Computer Science, University of Utah, Salt Lake City, Utah 84112 USA; e-mail: zachary@cs.utah.edu.

IEEECS Log Number S95048.

ping conflicts by adding extra code to their applications to compensate for the mapping decisions made in the module. Kiczales calls this "coding between the lines."

The open implementation approach separates implementation decisions into *base* and *meta* parts. The base part of the implementation is closed as in a black box abstraction. The meta part is open to modification by clients via the meta interface designed by the module designer, who carefully decides what will be open and what will be closed. An open implementation thus has both a conventional interface (exposing functionality) and a meta interface (exposing aspects of the implementation).

CLOS has an open implementation, and the CLOS metaobject protocol is the language's meta interface. It is implemented in an object-oriented fashion by exploiting reflective techniques [23], [24]. Via the metaobject protocol, users can alter the semantics of CLOS by using the standard object-oriented techniques of subclassing, specialization, and method combination.

Programmers build applications in CLOS by making use of the five basic elements of the language: classes, slots, methods, generic functions, and method combination. Each such element in a program is represented as a CLOS object. This object is called a *metaobject*, and its class is called a *metaclass*. For example, a user-defined class `student` will be represented as a metaobject that contains the structure and gives the semantics of the `student` class together with the generic functions defined for the class.

Because of this organization, the default semantics of CLOS can be given by the implementations of five metaclasses, one for each of classes, slots, methods, generic functions, and method combinations. The larger part of these implementations compose the *base* part of CLOS.

The five metaclasses from which metaobjects are made behave much like other classes in CLOS. Thus, one can change the semantics of a metaobject by modifying its metaclass. Although a user class definition can be freely changed, a metaclass definition can only be changed *incrementally* via subclassing, specialization, and method combination. Those aspects of the language that can be changed in this fashion compose the *meta* part of CLOS. This meta interface—the metaobject protocol—is part of the CLOS definition.

In this paper, then, we will discuss how we used the meta interface of CLOS to modify mapping decisions pertaining to class and object semantics as a means of incorporating persistence into CLOS.

III. An Object-Intensive Application

The Conceptual Design and Rendering System (CDRS) [14], [15] is a geometric CAD modeler that is being used by designers in dozens of major automotive and product design companies worldwide. The work presented in this paper was initially motivated by the problems of object persistence encountered in CDRS. CDRS is an *object-intensive* application written mostly in Common Lisp [26] as extended by CLOS. A typical model manipulated by CDRS contains tens of thousands of objects that may not all fit into memory, exhibits a wide variation in the sizes of objects, requires complex data structures within objects, and has rich semantic and structural relationships among objects. Supporting object persistence is particularly difficult in the presence of these characteristics.

CDRS uses a naive file-based, batch-oriented approach to object persistence that has proven ill-suited for the mix of objects that it manipulates [16]. All objects in a model are saved to a file at the end of a design session and are reloaded at the beginning of the next session. This approach requires a huge amount of physical 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. To make matters worse, users tend to save models frequently for fear of losing them due to reliability problems.

The solution that we sought was to introduce a tighter interface with a database system by providing incremental saves and loads. We attempted to do this not by modifying CDRS but by modifying CLOS via its metaobject protocol.

IV. MetaStore

We used the metaobject protocol to add persistent objects to CLOS. The resulting system, which we call MetaStore (Fig. 1), has two major components: a language extension (`Meta`) that was implemented via the CLOS metaobject protocol, and a persistent object store (`Store`) that provides database management features. We are concerned here with the language extension component and the degree to which the metaobject protocol facilitated and frustrated its implementation. For a complete discussion of the resulting system see [16].

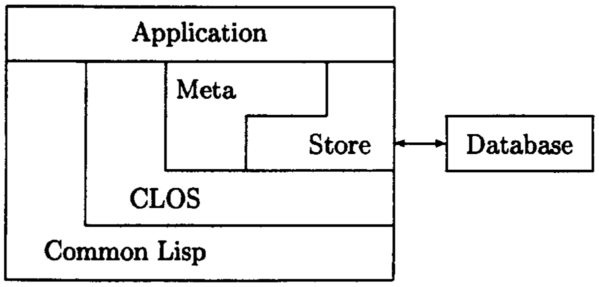

Fig. 1. MetaStore architecture.

MetaStore supports persistence at the granularity of objects and slots by maintaining a virtual object space within virtual memory. As the object space fills, MetaStore writes objects to disk and makes their virtual images available for garbage collection. This frees space for other objects, which are loaded as the application demands them. From an application's point of view, this amortizes the costs of loading and saving models over an entire design session, reducing user waiting time.

To support this form of persistence, we made substantial modifications to the way CLOS represents and manipulates objects. The question that concerned us throughout design and implementation was whether the overhead imposed by the metaprogramming would be too costly. For the most part it was not, although in some cases it was.

A. Overview of MetaStore

After explaining some terminology and exploring several key design decisions, we give an overview of MetaStore by describing the life cycle of a persistable object.

We distinguish *transient objects* from *persistable objects*. A transient object is an object in the conventional sense, whereas a persistable object is one whose value can persist between sessions. A persistable object becomes a *persistent object* when it is eventually saved to the object base. Based on our experience with CDRS, there are many classes whose objects need not be saved because they can be easily reconstructed. In the case of CDRS, treating all the data in a program as persistable, as is done in PS-Algol [6] and by Jacobs [9], would have been both unnecessary and impractical.

We also distinguish between persistable and transient slots within a persistable object. Only the persistable slots of a persistable object are ever saved to the object store; the contents of transient slots are discarded. This decision was also based on our experience with CDRS.

Finally, we distinguish atomic and composite slots. An *atomic* slot is one whose value is of an atomic data type, whereas a *composite slot* has as its value composite data (e.g., arrays and lists). In an object-intensive application such as CDRS, there are often a small number of large, persistable slot values in a persistable object. Saving and restoring an object as a single entity, including all of these large slot values, was judged impractical. Instead, we decided to save and restore composite slot values as separate entities.

Fig. 2 contains a non-persistable object, `O0`. Fig. 3 contains two persistable objects, `O1` and `O2`. `O1` has five slots. The `pid` slot is added automatically by MetaStore; the other four slots are defined directly in the class of `O1`. The same considerations apply to `O2`.

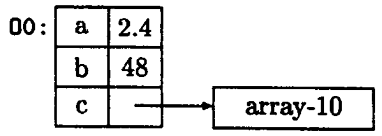

Fig. 2. A transient object in MetaStore.

A persistable object, just like any other object, is created by the CLOS method call `make-instance`. In MetaStore, when a persistable object is created, it is assigned a unique persistent identifier (PID). The PID of the object `O1` in Fig. 3 is `22`. A unique PID is necessary to map physical addresses to logical (persistent) addresses as objects are saved to the object base, and to map logical addresses to physical addresses as objects are loaded from the object base. Address translations in MetaStore are done by pointer swizzling with the help of an object table that maintains the mapping between physical and logical addresses (see [16] for more detail).

Upon creation of a persistable object, an intermediary data structure called a *phole*[1] is added between the persistable object and each of its persistable composite slot values. In Fig. 3, slots `b` and `d` of `O1` and `g` of `O2` are composite slots, and each has its own phole.

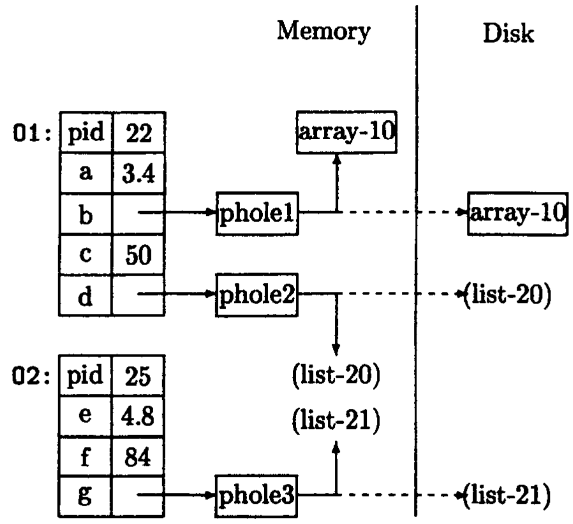

Fig. 3. Persistable objects in MetaStore.

A phole provides one level of indirection between a composite slot and its value and acts as a placeholder for a composite slot value. This makes it possible to do such things as

- maintain dirty bits for incremental saves,
- support persistence at the slot level,
- do lazy loading of composite slot values, and
- support a virtual object memory.

Once a new persistable object is created, it can be repeatedly accessed, read, or written. A write access to a persistable composite slot of a persistable object results in the associated phole being marked dirty. Any other write access to a persistable object, i.e., to an atomic slot, results in the object itself being marked dirty. These dirty bits are used to determine what parts of an object must be saved to the database when an object is saved. All newly created objects and composite values are treated as dirty until they are saved. Conversely, when a saved object is loaded from disk it is treated as clean until modified.

A saved object is usually loaded as a husk. A *husk* is an object whose pholes are uninstantiated. When a persistable composite slot value of a husk is first read accessed, it is loaded, instantiating the corresponding phole. Thus, pholes make lazy loading possible.[2] In Fig. 3, for example, when `O1` is first loaded as a husk, `array-10` and `list-20` are not

1. Phole stands for a *p*ersistent *hole*; it is pronounced "fole."

2. PCLOS [19] 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.

loaded until their corresponding slots are accessed for a read.

When the number of persistable objects in virtual memory reaches a certain limit, some objects (determined by the virtual object memory algorithm [16]) are flushed to free up some space in order to improve the system performance. When an object is flushed, it is converted back into a husk. A flushed object is thus exactly like one just loaded as a husk. A husk is not flushed until its application terminates.

B. Programmer's View of MetaStore

We kept the application programmer's interface to MetaStore small because we wanted to minimize the changes required to the large existing body of CDRS source code. We discuss below the source code modifications required to take advantage of MetaStore.

Each use of the CLOS `defclass` macro must reflect whether or not the user wants the instances of the class being defined to behave as persistable objects. For example, in the user-defined persistable class `student` below, the user must include two things to specify persistence.

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

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 the 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.

C. 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 some system-provided methods, and setting up some conventions that application programs must abide by. Some of them are listed below:

- Adding a PID 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 access to persistable objects
- Intercepting read and write accesses to objects to support a virtual object memory
- 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*

We will describe how some of these are implemented in the next section.

V. Four Kinds of Modifications

In this section we describe our experience with making four different kinds of modifications.

- Supporting persistence requires adding to the structure and modifying the behavior of user-defined persistable objects (Section V.A) by a means that is as transparent as possible to the application programmer. We made these kinds of changes effectively via the metaobject protocol.
- MetaStore must customize the meaning of slot accessing to account for the fact it adds a level of indirection between a persistable composite slot and its value (Section V.B). To implement this change we had to extend the metaobject protocol.
- Maintaining the consistency of dirty bits in composite persistable slot values is complicated by the fact that an arbitrary number of references may exist to a composite value from within a CLOS program (Section V.C). The most practical solution to this problem involved constraining application programmers to modify composite slot values in a stylized fashion.
- Dealing with the fact that mutable values such as arrays can be shared by multiple objects was the most difficult issue that we faced (Section V.D). The practical solution involved requiring that such shared values always be encapsulated within objects.

A. Structure and Behavior of Objects

The metaobject protocol is ideal for language extensions that involve modifications to the structure or behavior of objects. Changes to other kinds of data structures, such as arrays, are much more difficult and usually require some help from application programs or from the base language implementation. In this section we will sample the large range of language extensions that were conveniently implemented via the metaobject protocol.

A.1. Adding Slots to Objects

In addition to the user-defined slots, MetaStore maintains a persistent identifier (PID) and a dirty bit in each persistable object. The persistent identifier serves as a logical address in the underlying object store, and the dirty bit flags unsaved objects. As we saw in Section IV.B, a user-defined class is made persistent by specifying `persistable-metaclass` as its metaclass.

The `persistable-metaclass` is a subclass of the standard class metaclass, and thus inherits its structure and behavior. By modifying this inherited structure and behavior, MetaStore is able to modify the behavior of *all* persistable classes. Adding persistent identifier and dirty bit slots to each persistable object is just such a modification.

A.2. Creating Persistable Objects, Pholes, Husks

All objects in CLOS are created via calls to `make-instance`. The meaning of `make-instance` for persistable objects is modified by MetaStore to deal properly with persistable objects, pholes, and husks.

A number of tasks are performed by the modified `make-instance`. The PID of each newly-created persistable object is initialized. Pholes are inserted between each persistable composite slot and the value it contains. The mapping information required by the virtual object memory is updated.

If an object is being loaded from the object store, the behavior of `make-instance` is somewhat different. A husk is created instead of a complete object. Normally, a newly-created object's slots are set to user-defined initial values. Only the transient slots of a husk are initialized in this way. The atomic persistable slots are initialized with saved values, and the composite persistable slots are initialized with empty pholes. Each such phole will eventually be instantiated with a value when its slot is accessed: with a saved value if the slot is read, or with a new value if the slot is written.

A.3. Accessing Objects

Each read or write access to a persistable object is intercepted by MetaStore so that appropriate persistence-related actions can be handled.

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

A write access is more complicated than a read access. A write access to a persistable slot must take care of the following:

- If both the current and new values are atomic, set the object's dirty bit.
- If the current value is atomic (or uninitialized) and the new value is composite, add a phole to the slot with its dirty bit set. Set the dirty bit of the object as well.
- If the current value is composite and the new value is atomic, remove the phole and set the object's dirty bit.
- If both the current and new values are composite, set the phole's dirty bit.

All of this is accomplished by modifying the behavior of `(setf slot-value-using-class)`, which is the workhorse of the user-accessible routine `(setf slot-value)`.

B. Indirection on Slot Access

MetaStore supports persistence at the slot level, which requires it to maintain one level of indirection for each persistable composite slot. A persistable composite slot contains 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 obtain the value of a persistable slot, MetaStore must follow pointers to return the value stored in the phole. The implementation of MetaStore, however, must often directly obtain the phole via the same call. Supporting this dual meaning of `slot-value` was not straightforward and revealed a limitation of the metaobject protocol.

A general solution would have required providing two different semantics for the `slot-value` method depending upon from where and for what purpose it was called. The metaobject protocol provides no support for this, and as a result solving the problem involved making minor modifications to the protocol. Specifically, we had to go outside of the protocol to add an extra method for accessing slot values. `(setf slot-value)`, which is used to modify a slot value, presents the dual of the `slot-value` problem and was treated analogously.

This is an example of a mapping decision that was embedded into the closed portion of the CLOS implementation. While the metaobject protocol makes it easy to modify the behavior of certain existing methods, it does not provide for defining completely new ones. The price paid by MetaStore for overcoming the mapping decision in this way is a loss of portability: MetaStore can be ported only to implementations of CLOS that have been similarly extended.

C. Maintaining Dirty Bits

One of the goals of MetaStore was to provide support for transparent persistence for object-intensive applications. Rather than relying on application programs to keep track of what is saved to the database, MetaStore keeps track of what needs to be saved to maintain consistency of data.

This decision was made for two reasons. First, application programmers could easily make programming mistakes and either not save enough and sacrifice consistency or save too much and sacrifice economy. Second, MetaStore was specifically designed to support an existing application that was already built and being used. We could not rely on application programmers to extensively modify their volumes of existing code to cope with a new persistence scheme.

We decided, therefore, that MetaStore would maintain dirty bits for objects and composite slot values. The dirty bits are crucial, of course, because only dirty objects and values are ever saved to the database. Maintaining the dirty bits for objects was quite simple, as described in Section V.A. Maintaining dirty bits for composite values posed a much trickier problem, because composite values are not objects whose behavior can be easily tailored by the metaobject protocol.

The dirty bit of a composite slot, which is kept in its phole, must be set whenever a write access is made. Performing a write upon a slot value via the public interface of the containing object, i.e., via `(setf slot-value)`, poses no problem. The problem occurs when a program obtains a slot value via a *read* access (e.g., via `slot-value`) and then mutates that value. The following simple 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 through the phole associated with `slot1`, the dirty bit in the phole 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))
```

The extra call in the third line would solve the problem since `(setf slot-value)` can be (and, as discussed earlier, has been) modified via the metaobject protocol to maintain dirty bits. However, requiring this extra call changes the semantics of CLOS. All composite values except objects give rise to this problem.

Below, we sketch two solutions to this problem. The first is an expensive yet fully expressive solution that maintains dirty bits for composite slot values without any help from the application program. The second is a simpler and more practical solution requiring help from the application program developed along the lines suggested by the examples above. We ultimately implemented the second solution in MetaStore.

C.1. A Fully Expressive Solution

This solution supports incremental saves without requiring application programs to update dirty bits. Its design is complicated by the fact that the metaobject protocol affords no way to alter the structure and behavior of arrays and other composite values to add dirty bits, as they are not objects. We will sketch the solution in terms of arrays, but it generalizes naturally to cope with all composite values.

The solution requires:

- Creating and maintaining a hash table containing (pointers to) dirty arrays. This hash table will stay reasonably small if incremental saves are made frequently.
- Overloading the write accessor for arrays, `(setf aref)`, so that it adds the array being mutated to the hash table. This overloading cannot be done via the metaobject protocol, and is possible only if the underlying Common Lisp implementation is accessible.

When a persistable object is ready to be saved, we save each persistable array that it contains as follows. We first traverse the array (and all arrays reachable from it), using the hash table to identify dirty arrays. We remove each dirty array that we encounter from the hash table. If any dirty arrays are found during the traversal phase, we save the the entire slot value as one entry into the object base. If no dirty arrays are found, the slot value has not been modified since the last save.

There are a number of practical problems with this solution.

- Each write access to an array is burdened by an additional write to the hash table.
- The entire slot value for each persistable array slot of an object must be traversed to determine if any component is dirty.
- When a system-wide checkpoint is created, all persistable arrays in existence must be traversed and compared against the contents of the hash table.
- Not all implementations of Common Lisp admit the required overloading of `(setf aref)`.

C.2. A Practical Solution

These considerations led us to develop a more practical solution that places more of a burden upon application programmers. If an array-valued slot is modified via the public interface, then there is no extra responsibility on the part of the application program. However, if a user program modifies a slot value through the back door as illustrated earlier, then it must inform MetaStore which slot is changing via one of several means provided by MetaStore.

Writing a new application using MetaStore under these conditions is tolerable, but porting existing code is rather tedious. If a mistake is made by not informing MetaStore of a mutation, the consistency of the persistent object store may be compromised. A related danger is the possibility of hidden modifications on arrays by library routines. A user program passing a persistable array to a library routine must set dirty bits appropriately.

C.3. Analysis

Both the fully expressive and practical solutions are textbook examples of "coding between the lines" to overcome a mapping dilemma. The fundamental problem is that arrays and other composite values are native to Common Lisp and are thus beyond the purview of the CLOS metaobject protocol.

The fully expressive solution requires MetaStore to redefine an existing Common Lisp primitive (something that is not possible in all Common Lisp implementations), maintain a global hash table, and do repeated traversals of every persistable data structure accessible from any objet. The practical solution removes this burden from MetaStore and places it upon the application programmer, who must be careful to maintain dirty bits under penalty of losing data.

D. Persistence of Shared Structures

Structured data in Common Lisp can be shared freely by other structured data. This freedom causes much difficulty in supporting persistence for 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 Fig. 4, 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`.

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 [16], a fully expressive solution that, while inefficient, allows persistable structures to be shared is described. It also describes the practical solution implemented in MetaStore in which non-object structured data are not allowed to be shared unless they are encapsulated within objects. With the practical solution that we implemented, undetected sharing poses a serious problem. Therefore, we provided a tool to help to debug systems.

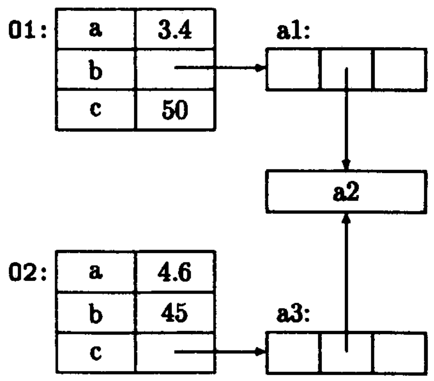

Fig. 4. An array shared by two objects.

VI. Performance Measurements

In this section we present some performance measurements that we made on our implementation of MetaStore. These measurements reveal several significant inefficient aspects of two implementations of the CLOS metaobject protocol: PCL [4] and Lucid CLOS [18]. The machine and the configuration we used are not included since we are interested in relative comparisons.

A. 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.

We begin by describing one significant inefficiency we found with both PCL and Lucid CLOS. In implementing MetaStore, a number of `:around` methods are attached to 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. An `:around` method is used to specify custom code that is to be run immediately before an existing method is executed.

To measure the cost of `:around` methods, we attached empty `:around` methods to each of the three methods identified above. We then compared the time required to execute each method to the time required to execute the method along with the dummy `:around` method. The ratios between the times reveals the overhead of using `:around` methods.

The measurements in this section were obtained using PCL, but we present what we learned about Lucid CLOS where appropriate.

- *Creation*: Using `make-instance`, we created 1,000 objects.

	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 object creation using dummy `:around` methods was about 50 times slower than without in Lucid CLOS [16]. 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*: We read accessed an object's slot 100,000 times using `slot-value-using-class`.

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

  Dummy `:around` methods made read accesses over 250 times slower than the normal transient read accesses.

- *Write Access*: We write accessed an object 100,000 times using `(setf slot-value-using-class)`.

	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 [13] 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.

B. MetaStore

In this section we present measurements based on our implementation of the MetaStore kernel. The MetaStore kernel provides the basic mechanism that supports 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, and being able to load on demand. Therefore, the kernel maintains object identities, pholes, the object table, dirty bits, and model identities for interfacing the object base.

Our measurements include the cost of the metaobject classes, the `:around` methods, and slot level persistence. Shared structures and virtual object memory are not included.

- *Creation*: We created 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*: We read accessed an object 100,000 times within a method using a persistable composite slot.

	Time
Transient	0.13 sec
MetaStore Kernel	35.62
Ratio	274.00

  Read accesses in the MetaStore kernel were about 270 times slower than the normal transient read accesses. Note that almost all of this slowdown can be accounted for in the overhead of using `:around` methods.

- *Write Access*: We write accessed an object 100,000 times.

	Time
Transient	0.11 sec
MetaStore Kernel	254.70
Ratio	2,315.45

  Write accesses in the MetaStore kernel were over 2,000 times slower than the normal transient write accesses. Factoring out the slowdown due to the overhead of `:around` methods, the slowdown factor for MetaStore is approximately 7.

Read accesses in the MetaStore kernel did not add any additional cost because there is no extra work added to the read mechanism at the kernel level. Most of the object creation overhead is due to the addition of pholes to persistable composite slots and the case analysis on slot values that are being used as initial values. Write accesses added substantial extra cost. Most of it is caused by the case analysis on the slot values in order to add or remove a phole as 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 limited by the speed of user interaction.

VII. Observations

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

A. 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 one tries 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 V.C and V.D was difficult because there are no metaobjects corresponding to the nonobject structures. In other words, 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.

B. Short-Term Improvements

Based on our experience with adding object persistence to CLOS in MetaStore, we propose here a few minor improvements to the existing protocol. They are related to slot accessing as described in Section V.B.

In MetaStore, we specialize the behavior of `slot-value-using-class` (which performs read accesses on slots) to account for the extra level of indirection that we add (via pholes) to persistable composite slots. While we always want an application program to use this specialized behavior, the MetaStore implementation itself sometimes needs to exploit the default behavior. Unfortunately, once the behavior of a method is modified via the metaobject protocol, the default behavior is no longer accessible.

One possible solution to this problem would be the addition of two more routines for slot access as follows:

- `slot-value-using-class-direct` would have the *default* `slot-value-using-class` behavior, even after the latter had been specialized.
- `(setf slot-value-using-class-direct)` is the dual of `slot-value-using-class-direct` for write accesses to slots.

A more general solution would be to extend the semantics of method combination by optionally allowing the execution of the original version of a method even after it is specialized. This is not an easy extension to support in general since it requires elaborate control over `primary` methods, `:before` methods, `:after` methods, and `:around` methods. The `copy-as` operation of Jigsaw [5] would solve this problem.

C. Long-Term Improvements

As discussed in Sections V.C and V.D, 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 equipped to support extensions to CLOS only as long as the extension is inherently object-oriented.

By pursuing the open implementation idea further it might be possible to push the metaobject protocol idea down to the base language implementation so that the metaobject protocol could operate at the Common Lisp data type level. If that were done, the problems that we experienced in MetaStore with implementing dirty bits and coping with shared structures could be readily solved. (Perhaps, we would then call it the *metadata protocol* or the *metatype protocol*.) Performance would, of course, be a critical concern with such a protocol. The choice of what to make open and what to keep closed would be the paramount design issue.

VIII. 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 [21], [22], 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 [25] program to an SPMD Scheme [21] program at compile time.

The authors in [1] 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 [19] is CLOS extended with persistence via the metaobject protocol of CLOS. PCLOS also supports persistence at the object level. It uses a database management system for secondary storage management, and suffers from the phenomenon known as impedance mismatch [2], [7].

The authors in [9] implemented persistence for Common Lisp values. Their language supports orthogonal persistence as does PS-Algol [6]. Orthogonal persistence is theoretically very attractive but so far has been impractical as described in [9] and [6].

Unlike PCLOS [19] and the work described in [1], 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 V. Two other important differences are that MetaStore, unlike [19] and [1], supports incremental saves and persistence of shared structures. MetaStore, also unlike [19], does not suffer from impedance mismatch as it uses a custom object database.

IX. Summary

The authors in [13] state that they have simultaneously achieved elegance and efficiency by basing language design on metaobject protocols, an open implementation. Our experience of extending CLOS with persistence via the metaobject protocol shows that current implementations are fairly efficient for the most part. However, we encountered some features that do not live up to this claim. For example, we observed up to 300 times slowdown in PCL, and up to 50 times slowdown in Lucid, with `:around` methods. Unfortunately, specializing default behavior by the use of `:around` methods is the feature of the metaobject protocol most commonly used in implementing MetaStore.

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 (and thus of open implementation) 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 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 is supported. An even better solution would be to extend the semantics of method combinations in CLOS in such a way that specialized methods such as `:around` methods could 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 in the current implementations. 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. We propose that all composite data types be implemented as objects so that they can be included in the metaobject protocol. That is, we propose that the base language itself be implemented in an open fashion. This would, of course, entail a significant effort.

Finally, the poor performance results and the difficulties we experienced with dirty bits and shared structures should be interpreted with caution. That is, we must differentiate the cost incurred by the metaobject protocol itself from the cost stemming from the persistence requirements. Much of the performance hit that we encountered was due to the severe requirement of supporting almost transparent persistence. Our experiment seemed too ambitious for these early implementations of a pioneering, experimental idea: open implementation. Despite our negative reports, therefore, we see a great deal of merit to this idea and are excited about its potential impact on software engineering and software science in general.

ACKNOWLEDGMENTS

The authors would like to thank G. Kiczales for helping us with the metaobject protocol idea and its implementation in PCL; A. Paepcke for also helping us with the metaobject protocol of PCL; R. Kessler, G. Lindstrom, and M. Swanson for stimulating discussions on technical issues; and the members of the CDRS group at Evans & Sutherland Computer Corporation for the opportunity to work on object persistence and for technical discussions.

REFERENCES

[1] G. Attardi, C. Bonini, M.R. Boscotrecase, T. Flagella, and M. Gaspari, "Metalevel programming in CLOS," *Proc. European Conf. on Object-Oriented Programming,* 1989.

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

[3] 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.

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

[5] G. Bracha, "The programming language Jigsaw: Mixins, modularity, and multiple inheritance," PhD dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

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

[7] G. Copeland and D. Maier, "Making Smalltalk a database system," *Proc. ACM SIGMOD Int'l Conf. on Management of Data* (June 1984). *ACM SIGMOD Record,* vol. 14, no. 2, 1984.

[8] IEEE Computer Society, *IEEE Standard for the Scheme Programming Language,* IEEE Std 1178-1990. New York: IEEE, 1991.

[9] J.H. Jacobs and M.R. Swanson, "Syntax and semantics of a persistent Common Lisp," *Proc. ACM Symp. on Lisp and Functional Programming,* 1993.

[10] G. Kiczales, "Towards a new model of abstraction in software engineering," *Proc. IMSA'92 Workshop on Reflection and Meta-level Architectures,* 1992.

[11] G. Kiczales, "Why are black boxes so hard to reuse? (Towards a new model of abstraction in the engineering of software)," invited talk at the *OOPSLA'94 Conf. on Object-Oriented Programming Systems, Languages, and Applications,* Portland, Ore., Oct. 1994.

[12] G. Kiczales, R. DeLine, A.H. Lee, and C. Maeda, "Open implementation analysis and design of substrate software," tutorial to be given at the *OOPSLA'95 Conf. on Object-Oriented Programming Systems, Languages, and Applications,* Austin, Tex., Oct. 1995.

[13] G. Kiczales, J. des Rivières, and D.G. Bobrow, *The Art of the Metaobject Protocol.* Cambridge, Mass.: MIT Press, 1991.

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

[15] A.H. Lee, "Managing complex objects," internal document, Evans & Sutherland Computer Co., 1990.

[16] A.H. Lee, "The persistent object system MetaStore: Persistence via metaprogramming," PhD dissertation, Dept. of Computer Science, Univ. of Utah, 1992.

[17] A.H. Lee and J.L. Zachary, "Using metaprogramming to add persistence to CLOS," *1994 Int'l Conf. on Computer Languages.* Los Alamitos, Calif.: IEEE CS Press, 1994.

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

[19] A. Paepcke, "PCLOS: Stress testing CLOS: Experiencing the metaobject protocol," *Proc. ACM Conf. on Object-Oriented Programming Systems, Languages, and Applications,* 1990.

[20] A. Paepcke, "User-level language crafting: Introducing the CLOS metaobject protocol," A. Paepcke, ed., *Object-Oriented Programming: The CLOS Perspective,* 1992.

[21] L.H. Rodriguez, Jr., "Coarse-grained parallelism using metaobject protocols," MS thesis, Massachusetts Institute of Technology, 1991. (Also available as Tech. Rep. SSL-91-06, Xerox Palo Alto Research Center, 1991.)

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

[23] B.C. Smith, "Reflection and semantics in a procedural language," (PhD dissertation), Technical Report TR-272, Laboratory for Computer Science, Massachusetts Institute of Technology, 1982.

[24] B.C. Smith, "Reflection and semantics in Lisp," *Proc. ACM Symp. on Principles of Programming Languages (POPL),* Salt Lake City, Utah, 1984.

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

[26] G.L. Steele, Jr., *Common Lisp: The Language,* second ed. Digital Press, 1990.

[27] G.L. Steele, Jr., *Common Lisp: The Language,* chapter 28, second ed. Digital Press, 1990.

Arthur H. Lee received the BS degree from the University of Utah, the MS degree from Stanford University, and the PhD degree from the University of Utah, all in computer science. Before he joined the Computer Science Department at Korea University in Seoul, Korea, as an assistant professor in 1993, he worked for about 11 years combined at three different places: at Sandia National Laboratories as an MTS on computer graphics, at Xerox Palo Alto Research Center as a research staff member on document recognition, and at Evans & Sutherland Computer Corp. as a research computer scientist on object management, object persistence, and computer graphics. His current research interests include object-oriented systems, open implementations, object-oriented database systems, and computer graphics. Dr. Lee is currently the editor-in-chief of the *Journal of the Korea Computer Graphics Society*, is a member of the Board of Directors of the Korea Computer Graphics Society, and is scheduled to teach tutorials on open implementations at OOPSLA'95 and the 1996 International Conference on Software Engineering. He is also known as Hee-woong Lee in Korea.

Joseph L. Zachary received the SB degree in 1979, the SM degree in 1983, and the PhD degree in 1987, all in computer science from the Massachusetts Institute of Technology. In 1987 he became an assistant professor in the Department of Computer Science at the University of Utah, where he is now an associate professor. His current research interests center around the application of computing to undergraduate education. He has developed extensive online tutorial materials for teaching introductory courses in computing. Dr. Zachary is a charter member of the Department of Energy's Undergraduate Computational Engineering and Science Committee. In that capacity he has crafted a variety of courseware and supporting software for teaching computational science to undergraduate engineering and science students.