Working with Persistent Objects: To Swizzle or Not to Swizzle*

J. Eliot B. Moss†

COINS Technical Report 90-38
May 1990

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

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

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

Submitted to IEEE Transactions on Computers

Abstract

Abstract. We have developed an abstract model of working with persistent objects, derived from it an analytical performance model, and validated the performance model via experimental measurements using the Mneme persistent object store. The abstract model defines the notions of a work *session* upon a *collection* of persistent objects, allowing collections to be created, loaded, saved, and worked upon. Throughout a session the identity of the objects is preserved, as opposed to creating new collections of objects.

We considered three approaches to managing objects: keeping them in the object store and manipulating them via object store operations; converting all logical pointers to physical pointers ("swizzling") within the object store's buffers; and creating in-memory copies with logical pointers replaced by physical pointers. Our experiments considered object collections that fit in main memory and compared the three approaches with each other, and with a non-persistent baseline where applicable. The in-store approach was found to perform more poorly than the other approaches unless the amount of work was quite small. The copy approach offered the best performance, given that both copies of the object collection fit in main memory. The key result, though, is a simple, accurate, predictive performance model for work sessions. The results are of relevance to designing and implementing object stores, persistent and data base programming languages, design databases, and related applications.

1 Introduction

There is considerable current interest in using objects and object-oriented approaches for building software and applications with improved functionality and extensibility. Since many applications operate on stored permanent data, there is consequent demand for object oriented approaches to manipulating such data. Object oriented database management systems provide one very comprehensive approach, including support for storage, indexing, querying, etc. Database systems take the view that the data mostly reside on permanent secondary storage, with main memory being used as a very temporary scratch buffer. On the other hand, we believe that many modern applications, among them ones that will most benefit from or are most suited to the object-oriented approach, such as design tools, typically exhibit a markedly different kind of behavior. In particular, they perform considerable focused work, such as editing or design rule checking, on readily identified collections of objects, by loading the objects into main memory, working on them, and later saving modifications. CAD and CASE tools and document preparation systems provide interesting examples of this application behavior.

Database systems and their performance measures are oriented towards the problem of searching for relevant data, possibly with some modest updates. We feel that for design (and related) applications there is a significant component that involves little search and that must provide for high performance manipulation of objects in main memory by complex applications programs. We do not mean to imply that searching (querying) is not important, just that there is another very important mode of use.

Persistent object stores, which focus less on search and more on storing objects and retrieving them by "name" (unique identifier), provide an alternative to comprehensive object-oriented databases. Object stores are likely to be more efficient than databases, because they do not attempt to provide all the features of a database system. They might also be used as the foundation for a database system—object stores and databases are thus not incompatible alternatives. However, we are less concerned with whether a database system or persistent object store is used to store and retrieve objects than we are with the performance of applications that conform to the *load/work/save* (LWS) paradigm. Since object stores suit this paradigm and will tend to offer better performance, we explored the LWS approach using the Mneme persistent object store [1, 2, 3].[1] We used Mneme primarily because it was available and was designed for this purpose, but it also offers good performance for this task.

In order to explore the LWS mode of working with objects, we devised a model of *sessions* of work with objects, which we present in detail in the next section. By identifying and separating various activities, the session model allowed us to consider, in a simple and somewhat independent way, the performance of different approaches to carrying out these activities. For example, in the phase of a session where one works on resident objects, there is a fundamental

[1]Mneme is pronounced NEE-mee, from the Greek word for *memory*.

choice of whether to work on their external format or to work on a possibly more efficient internal format. The cost of the different approaches to work can be evaluated independently of the cost of conversion between formats, though working on the internal form does require some form of conversion while the external form does not. A *post facto* analysis of experimental performance results confirmed the important factors affecting performance of each work phase, and thus validated a remarkably simple but quite accurate analytical model, and calibrated the model (provided values for its various constants).

After presenting the session model and the analytical performance model, we discuss different approaches to managing objects, giving a taxonomy of techniques. Next we present more details and background on the Mneme persistent object store. This is followed by a discussion of the experimental methodology and presentation of the experimental results. We defer analysis and discussion of those results to a separate section, and finally offer our conclusions and directions for future work.

The key concepts and results of this paper are: the session model of the LWS style of work, the analytical performance model, its validation, the taxonomy of object management techniques, and the comparisons between techniques possible because of the validated performance model.

2 A Model of Working with Objects

Our model of the LWS (load/work/save) work style is organized around two concepts: that of a *collection* of objects, and that of a *session* of work upon a collection.

2.1 Collections

A collection of objects is a set of objects, including the internal data (immediate contents) of the objects, and the references those objects have to other objects in the collection.[2] A collection can be visualized as a directed graph in which nodes are the objects and the edges are the inter-object references. The graph may have sharing, cycles, and self loops. To model a collection more accurately, we can label the nodes with the (non-pointer) state of the objects, and the edges with relative edge numbers (to distinguish between the various outgoing edges of each object), but we are not concerned with a formal model so much as the general structural properties of a collection of objects.

We assume that for a given session the collection of objects manipulated is *identifiable*, by which we mean no significant searching is required to identify or isolate the collection, and *localized*, by which we mean that the objects are clustered or co-located so that to retrieve them one need not expend effort separating them from larger sets of objects. These definitions are

[2]References to objects not in the collection might not occur (if the collection is transitively closed), or might be considered as ordinary data for purposes of modeling a particular collection.

meant to clarify and expand upon the difference between typical database access and the style of access we considered.

A collection need not be an entire database or file, and inter-object references that are not used in a given session need not be considered, except to the extent that they add bulk to the objects. A collection can always be thought of as a set of root objects plus additional objects reachable from those roots via selected edges. In general, the selected edges might be determined dynamically, but the overall collection is certainly known after the fact. That is, the set of objects in a collection may not be known *a priori* and may be "discovered" as computation proceeds. In some of our techniques we assume the collection is known beforehand and consider the performance advantage of this additional knowledge.

2.2 Objects

Clearly the concept of a *collection* of objects is not well defined in the absence of a model for *objects*, which we now provide. An individual object consists of some number of *slots* and some number of *bytes*. A slot can contain an object reference (the exact form varies with the object management technique used), a null reference, or non-pointer data. Mneme uses the sign bit for a tag to distinguish references from non-pointer data. This particular technique makes a pointer vs. non-pointer check cost the same as a null pointer check, so the cost of tagging can generally be ignored.

The bytes part of objects are sometimes treated as bytes, to model character and bit string manipulation, and sometimes as integers, to model numeric operations. For simplicity, the number of slots and bytes in an object is not changed after the object is created. Objects model the individual records, arrays, strings, etc., of the application.

Note that we assume only *structural* properties of objects and are not requiring any sort of type or class system, much less a class or type hierarchy or inheritance relationships. Neither are we requiring any particular form for operations on objects or invocation of operations. This helps keep the model and results more independent of programming language and application (though admittedly some object management techniques are better suited to some languages than others).

2.3 Sessions

A work *session* has three phases: *loading* an existing object collection or *creating* a new one, *working* on the collection, and *saving* new or modified objects. We also consider the possibility of overlapping loading and working if the object management scheme allows incremental loading.

Loading consists of acquiring the objects and making them available to the application for manipulation. The details vary according to the specific object management technique used, as

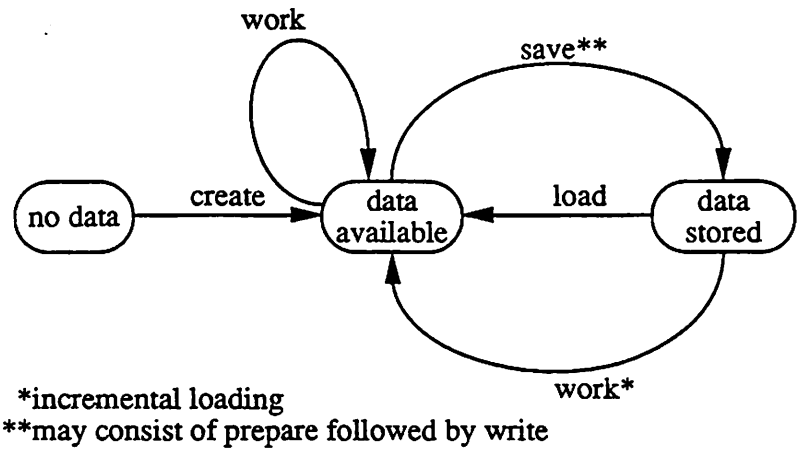


Figure 1: Work session state diagram

described in the next section. Creating does a bulk creation of a collection; we vary the extent of the initialization of the objects to examine initialization cost. Creating is different from loading in that creating is an in-memory operation whereas loading requires secondary storage access. Saving is the converse of loading: the movement of new and modified objects back into the persistent storage system. In some cases it is better modeled as two sub-phases: *preparing*, which consists of moving data from the application into the storage manager's buffers, and *writing*, the movement of the buffered information to secondary storage. Finally, working on objects consists of reading or updating their slots, bytes, or integers (byte data treated as numerics). As with creation, we are mostly concerned with data resident in main memory; but if we are overlapping loading with working, then the data may not be resident. A complete state diagram for work sessions is given in Figure 1.

3 Techniques of Object Management

We now describe the various approaches to managing objects that we considered in our modeling and performance measurement efforts. We organize the discussion around a taxonomy of the management techniques. The most fundamental consideration, which leads to a top-level dichotomy of approaches, is whether objects are always named and located via the identifiers assigned by the object store software, which we call the *in-store* approach, or whether the application accesses objects via pointers to the object contents, which requires a conversion from object identifiers to pointers, called *swizzling* the pointers.[3] We consider each of these

[3]We would be grateful to the referees if they could indicate the origin or earliest use of this term!

general approaches in more detail below, and develop further taxonomic categories for variants within the approaches.

3.1 In-store techniques

Applications using the in-store approach always name and gain access to an individual object by presenting the object's identifier to the object store software. The object store software then locates the object in memory, or fetches it from secondary storage if it is not resident, and manipulation of the object proceeds. A significant variation within the in-store approach is whether, once the object has been located, the application can manipulate it directly. The original Mneme interface provides a data structure called a *handle* in response to a request to access an object. The handle can then be passed to a variety of subroutines supporting controlled reading and writing of the object's contents. These routines verify the handle and any additional arguments, perform bounds checking to avoid reading or writing outside of the object in question, etc. This interface is known as the *call interface*. While quite safe, the call interface obviously incurs more overhead than allowing the application to manipulate object contents directly.

More recently we added a *pointer interface* to Mneme, which gives the application a pointer to the object's contents within the object store's buffers. The application must follow some sensible rules (e.g., not accessing outside the bounds of the object), so using this interface requires some care. In the case of a persistent programming language, it is the compiler and run-time system that will be making calls on Mneme, so safety is less of an issue. We observe that the Exodus storage manager [4, 5, 6] uses a call interface for updates, using a data structure analogous to Mneme's handles, but allows direct reading of object contents, and hence is intermediate between the Mneme call and pointer interfaces.

As the performance results will show, the call interface is significantly slower than the pointer interface. This is true even though Mneme uses object location techniques possibly faster than those used by database buffer managers to locate resident database records. The specific techniques are described further in the section on Mneme.

3.2 Swizzling techniques

In contrast to the in-store techniques, when using swizzling an application always refers to an object via a direct pointer to the object. Note that this is different from the in-store/pointer technique, which gains access to objects via their object identifiers. That is, in-store methods *always* call the object manager to traverse a reference to the object at the other end. They *may* provide a direct pointer to manipulate to object once it is located, but that pointer is not saved for future use. Swizzling techniques *replace* object identifiers with actual pointers, and do *not* call the object manager to traverse references. There are two separate dichotomies among the

swizzling techniques: whether swizzling is done all at once (in a "batch") or incrementally, and whether swizzling operates in place or produces copies of the original objects.

3.2.1 Eager and lazy swizzling

Swizzling may be *eager* or *lazy*. Let us consider eager swizzling first. When using eager swizzling, all objects of a collection are fetched *in advance* and swizzled. Thereafter, access proceeds little differently from using non-persistent objects. While swizzling, one must maintain a table mapping from object identifiers to in-memory addresses, so that multiple references to the same object will resolve to the same address. Suppose one is swizzling (converting) an object X, and encounters an object id for some object Y (Y may be the same as X). We probe the table and if there is an entry, we replace the id with the address of Y. If there is no entry for Y in the table, then we call the object manager to locate Y and insert Y's id and address in the table. We also insure that the object named by Y is swizzled eventually, too.

In contrast with eager swizzling, which somehow knows where to stop in traversing the object graph and reading objects into memory, lazy swizzling converts only those objects that are actually used. In particular, if object X refers to object Y as above, then when X is converted, Y is fetched and assigned an address, but Y is not converted. Rather, every object is tagged with a bit indicating whether or not it has been converted, and when the object is to be used, this bit is checked. If the object has not yet been converted, it is converted immediately before its first use. This scheme incurs the additional overhead of checking the "converted" bit, but allows one to avoid some prefetching of unnecessary data and to operate reasonably when the collection of objects cannot be determined very well in advance.

Note that the lazy technique just described marks the *objects*; since objects are the nodes of the object graph, we term this *node marking*. There are alternative lazy swizzling techniques, such as marking the edges, using an extra level of indirection, or using memory management hardware faults to detect access to non-resident objects. Such alternative techniques fit into our modeling framework, but we have experimental results only for eager swizzling and node-marking lazy swizzling. Future studies to compare variants of lazy swizzling may be worthwhile, but our study provides basic results comparing swizzling and in-store techniques. In fact, we now argue that all the lazy swizzling techniques will have roughly the same performance, and definitely the same qualitative behavior.

First, *any* lazy scheme needs some sort of check performed all the time, to force the fetching of non-resident objects. We measured working with and without the node-marking check enabled, so we have quantified the cost of such checks when they are done explicitly; edge-marking checks would have similar cost. Total edge-marking costs are probably higher than node-marking, since at the time an edge value is checked, one generally does not know where the value originated, so when a fault is taken, there will either be a cost to locate and update marked edges to become direct pointers, or else one will continue to detect the marked edge and use the object

store (or a table lookup) to locate the object. If, in a node-marking scheme, one inserts an extra level of indirection to avoid fetching an extra level of objects, then the indirection cost is incurred at run-time. It would appear that any incremental scheme will have a cost of a couple of instruction per object use to check residency, unless hardware is available and exploited, or unless more sophisticated techniques are used to do some eager swizzling (to allow the checks to be eliminated). Concerning the latter point, novel compiler optimization techniques, especially ones that collect and apply global properties of programs, would be useful, but go beyond the scope of the present work.

As for hardware support, an idea under current discussion and purported use in commercial products near release is to use memory management access checks to detect references to non-resident data. Here one would assign virtual memory addresses to non-resident objects referred to by resident objects, but set the pages containing these non-resident objects to "no access". If the non-resident objects are accessed, a page fault occurs, the page access is set read/write, the objects are fetched from the store, and execution resumes. This sounds simple, but requires the same kind of swizzling table support as that required by software-only techniques. It also requires knowing the size of objects without fetching them, which is possible only for some objects, even in a system with static type information available.[4] It is simpler to insert a level of indirection, though it could perhaps be avoided in the statically known cases. If one assumes that the cost of fielding the faults is minor compared with fetching the objects from the store and swizzling them, then our current studies can fairly predict, and certainly bound, the performance of such hardware supported schemes.

To summarize the considerations: the present study puts bounds on the cost of lazy swizzling schemes and allows comparison with eager swizzling. While more detailed comparisons between lazy schemes would be of some use, our performance bounds are fairly tight (as will be seen), so we have not undertaken such studies to date.

3.2.2 In-place swizzling and copies

In-place swizzling substitutes pointers for object ids within the copies of objects maintained by the object store in its "private" buffers. That is, any given object is read from disk (or some server) into an object store buffer, and then is swizzled where it lies. This technique generally requires that the object be un-swizzled before completing a work session, since the object store software may write back an entire buffer of objects if *any* object in the buffer has been modified. Clearly, in-place swizzling can be used only if the object store permits general in-place object manipulation. The Mneme pointer interface does permit that; neither the Mneme call interface nor the Exodus storage manager does.

[4]Non-statically sized arrays are a difficult case, though subtypes in inheritance hierarchies could also be larger than apparent from the supertype alone.

Copy swizzling creates a separate copy of each swizzled object in the application's heap storage area, and does not modify objects in the object manager's buffers. The copy and original object must be maintained in correspondence with one another; the swizzling table supports that correspondence, insuring that each object id maps to at most one copy in the application heap.

The relative advantages of in-place and copy swizzling are reasonably obvious: in-place swizzling demands less memory, but may require more un-swizzling work. Some more subtle advantages accrue to copy swizzling: it may tend to increase locality of reference in virtual memory, it works when the object store does not permit direct buffer access, it allows the object store to replace/remove buffers easily, and it reduces memory demands if the application is not using most of each buffer *and* the buffers can be discarded after creating the application copies. This last advantage might cut both ways, though, in that updates could require re-fetching buffers to allow the modified objects to be merged.

3.2.3 Additional considerations for swizzling

A point we have previously discussed is that all swizzling schemes require the ability to map from an object id to the address of the corresponding resident object, if any. Since an object store must be able to locate objects in its buffers, in-place swizzling can rely on the object store. For better comparison with copy swizzling, we also measured the cost of using the copy swizzling lookup table for doing in-place swizzling lookups. Because the table was tailored to the situation, and hence not fully general, and because it eliminated at least one indirection used for flexibility in the Mneme technique, the table lookup was a bit faster than using the object store. In a practical implementation the lookup cost would likely be closer to that of using the store rather than the table. In any case, we measured in-place swizzling under each lookup scheme.

Another consideration is that if we swizzle, we must be able to *un*swizzle as well: to determine an object's id given a pointer to the object. Further, if presented with a pointer to an object created during the session, we should detect that fact and be able to create a corresponding object in the object store. That is, it is almost certainly a bad idea to create all objects in the store in the first place, since most newly created transient objects in actual programs need not persist and allocation in the store is likely to be substantially more expensive than allocation in the application heap. Note that when an application's work is complete we need to locate and follow all references from persistent objects to not-yet-persistent ones, and make the latter ones persistent. Of course this must be done recursively: the objects to be kept are those reachable from a persistent object via a chain of references of any depth.

The simplest way to deal with unswizzling is to leave room in every object for the object's id. This slot is initially null for transient objects, but can be filled in during the unswizzling process. Persistent objects have the slot filled in all the time. If persistent and non-persistent objects can be otherwise distinguished, and if the object store provides other means to obtain

an object's id given a pointer to the object, then we need not allocate space for ids in persistent objects. Another approach would be to have a separate table mapping from addresses to object ids. This approach is probably a little more painful than leaving space in the objects, since the table must be taken into account by compacting garbage collectors. It is unlikely, though, that any technique will outperform keeping the ids within the objects, so that is what we implemented for our measurements.

Finally, for copy swizzling it is useful and important to know whether an object has been modified. To that end, we added a "modified" bit to every object, in addition to the "swizzled" bit, and all implementations update the modified bit if the object is modified.[5]

3.3 Taxonomy of management techniques

The object management techniques we have discussed in this section are summarized in Table 1 below. We have omitted considerations such as whether or not hardware support is used and whether the object store is used to look up resident objects for in-place swizzling, since they go further into implementation details than is useful for a taxonomy.

In-store	Call interface		
	Pointer interface		
Swizzling	Eager	In-place	
		Copy	
	Lazy	In-place	Node marking
			Edge marking
		Copy	Node marking
			Edge marking

Table 1: Taxonomy of object management techniques

4 A predictive performance model

Given the session model and some knowledge of the various object management techniques, it is reasonable to attempt to devise a model of performance. The performance measure we examined was total (elapsed) time to perform a session of work. This time should be expressed as a function of some arguments describing the *collection* and some arguments describing the nature and amount of work performed in the *session*. Further, it would be desirable if the model's function can be characterized in a general way, with a number of parameters that depend on

[5]The actual encoding is slightly more subtle: 00 = new transient object; 01 = persistent, unswizzled object; 10 = swizzled, modified object; 11 = swizzled, unmodified object.

the object management technique (the *implementation*). Thus, we hope for a relatively simple function of three kinds of parameters: ones describing the collection, ones describing the session, and ones describing performance related aspects of the implementation. Further, for the model to be really useful, all these parameters must be measurable.

We did indeed devise such a model once we had collected the experimental data, and we now present it. First, the three phases of the session model can be separated in most cases and are clearly additive, one of the advantages of the LWS viewpoint. The lazy swizzling technique does blend loading and working, but it turns out that this does not affect the parameterization of the model. We characterize the separation of phases by this equation:

$$T = L + W + S$$

That is, total time is just the sum of the time for each of the phases: *L*oad, *W*ork, and *S*ave. L and S depend only on the collection and implementation parameters, but W depends also on the work parameters.

What are these various parameters? We present the collection and work parameters in Table 2 below:

Collection parameters	
n	number of objects in the collection
p	fraction of object contents devoted to slots ("pointers")
i	fraction of slots "initialized" (containing references)
b	average number of bytes per object, including headers, etc.

Session parameters	
u	fraction of objects updated (modified) in the session
v	average number of times an object is visited and operated upon
w	average amount of work per object-visit (in μs)

Table 2: Collection and session descriptive parameters

The implementation parameters are most easily described by writing down the equations defining L, W, and S:

$$\begin{aligned} L &= l_1 + l_2nb + l_3n + l_4npb + l_5npbi \\ W &= w_1 + w_2nv + nvw \\ S &= s_1 + s_2nb + s_3n + s_4npb + s_5npbi + s_6nbu + s_7nu + s_8npbu + s_9npbiu \end{aligned}$$

Since L and S involve the same forms of terms, we can combine their constants into $m_i = l_i + s_i$ (m is for *move*). The resulting model is:

$$
\begin{aligned}
W &= w_1 + w_2 nv + nvw \\
M &= m_1 + m_2 nb + m_3 n + m_4 npb + m_5 npbi + m_6 nbu + m_7 nu + m_8 npbu + m_9 npbiu \\
T &= W + M \\
&= W + L + S
\end{aligned}
$$

Let us briefly justify the form of these equations. Both working and moving data can be expected to have some small setup time, w_1 and m_1, which can be neglected if n is large enough, though measurements will be needed to determine just how negligible these constants are. The $w_2 nv$ term represents the cost of following references to begin to access objects, and the nvw term captures the actual work. Note that v and w can be arbitrarily large, but it will be useful to obtain some sort of lower bound on w to see how significant the w_2 term is.

The $m_2 nb$ represents the cost of moving the bytes of the objects around (to/from secondary storage, in memory, copying, etc.), and $m_3 n$ captures per-object overheads. The $m_4 npb$ and $m_5 npbi$ terms capture the cost of swizzling: we must examine every slot (the average number of slots is pb divided by the slots size expressed in bytes), and convert every initialized slot. The m_6 through m_9 terms express the *additional* costs incurred when objects are updated as opposed to simply read. (In some schemes m_6 through m_9 might be zero.)

We expect linear effects because we are providing adequate memory resources, and because we assume no data is resident in memory (even in an operating system disk cache) at the start of a session. Thus, the simplicity of the model is due partly to the simplicity of the situation; if memory resources were tight, we would expect non-linear, or perhaps piecewise linear behavior.

To model creation of collections, we note that the behavior is captured by the same model with different m parameters and with u set to 1. Our experiments did compare object creation costs under different object management techniques, so we were able to develop a quantitative cost model for collection creation sessions (create/work/save) as well as load/work/save sessions.

5 The Mneme persistent object store

Since Mneme was the object store used in the experiments, more details concerning Mneme, its abstractions, and their implementation, will help in understanding the results. After introducing Mneme, we offer some comments on the generality of results collected using Mneme.

A Mneme object is a collection of *slots* and *bytes*, along with a 1-byte *attributes* field. Mneme does not specify what the attributes are to be used for. The bytes part is simply a vector of 8-bit bytes. The slots part is a vector of 32-bit slots. Each slot contains one of two things, depending on the sign bit of the slot: an immediate 31-bit integer value, or an *object identifier*. There is a distinguished *empty* (null) object identifier, conventionally represented by 0.

Mneme groups objects together into units called *files*. A file of objects can be separately named and located within the overall store. Files are a convenient unit for storage, providing

modularity of the object space, and are intended to be reasonable units of backup, recovery, and transfer between different Mneme stores.

Files also allow us to take advantage of modularity of name space. *Persistent* object identifiers, as stored in objects within the Mneme store, always name objects within the *same* file as the object containing the identifier. This allows identifiers to be relatively short (28 bits). References to objects in other files are made by referring to *forwarder* objects within the same file. A forwarder contains enough information to name and locate the intended target object.

Because clients (e.g., Smalltalk) may have many files open at the same time during a session of interaction with the store, a persistent identifier, which is unique only within one file, must be converted into a *client* identifier for use by the client. An object's client identifier is guaranteed to name the object uniquely for the duration of a Mneme *session*. This also permits the persistent identifiers to be reassigned between sessions, allowing reclustering, garbage collection, reuse of the limited space of identifiers, and even explicit deletion. Since identifiers can be reassigned by the object store between sessions they cannot simply be synthesized and presented to the store with any reliability. Thus it is essential that each file have a distinguished *root*: a slot that can be set at will to indicate a starting place for naming objects in the file.

A Mneme session is a period of interaction with the store, establishing a context of use, including open Mneme files and identifiers of objects within those files. We can view a session as being a window onto the store. While the limited size of Mneme client identifiers does limit the number of objects that may be uniquely addressed during any one Mneme session, it does not limit the overall number of objects in the store.

The unit of retrieval in Mneme is the *physical segment*. A physical segment physically groups a number of objects together. When one of the objects in the physical segment is to be faulted in, the whole segment is placed in a buffer in memory. A file constitutes a number of physical segments.

When a client requires access to a persistent object it presents a *client* identifier to Mneme. If the object is already resident Mneme simply returns a pointer to the object in virtual memory. If the object is not already resident, Mneme allocates a buffer in memory in which it places the object's physical segment, and then returns a pointer to the requested object within that buffer.

Mneme is implemented as a collection of C routines that form a library suitable for linking with the application. The goals of the Mneme project include robust support for multiple users, including cooperative work, in a distributed system, with policy extensibility. The prototype used for the experiments is single user, non-distributed, and not resilient to crashes, but does provides some interesting policy extension features.

Our experiments could be criticized on the grounds that we did not include concurrency control and resilience. While concurrency control and resilience will undoubtedly add *some* overhead, we believe that the effects will be minimal in most cases, since concurrency control would generally be performed on physical segments rather than individual objects, and Mneme

physical segments tend to be larger than typical database pages (e.g., 32K bytes versus 4K or 8K). The overhead of an extra one or two disk writes for crash resilient log forcing will also be minimal on top of reading and possibly writing collections of an interesting sizes (100K bytes and up). Still, one can view the current experiments as a lower bound on performance. Further, the object management technique used should not affect the concurrency control and recovery overhead, which should be the same for all techniques. Since our purpose here is to compare object management techniques, overheads common to all of them are less important to model accurately anyway.

F = File # (8 bits)	L = Logical Segment # (10 bits)	O = Object # (10 bits)

Figure 2: Mneme Object Id Format

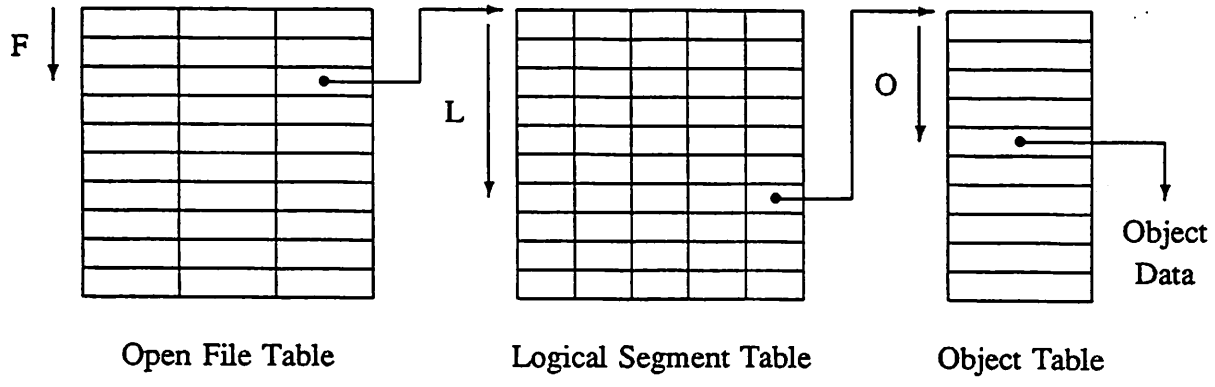


Figure 3: Two Level Direct Map Object Lookup Scheme

More generally, we believe that Mneme provides access to object collections (as defined in this paper) with very nearly minimal overhead. Most object have only 8 bytes of overhead: 4 for the header word, which includes the object's size, and 4 for an indirection pointer, which is necessary for efficient access to objects within segments. Mneme's object location mechanism is very fast for resident objects, operating as follows. An object id consists of three fields: 8 bits identifying an open file, 10 bits identifying a group of object within the file (these objects must lie in the same physical segment and are termed a *logical segment*), and 10 bits identifying the object with that group. To find a resident object, one indexes the open file table with the file number and obtains a pointer to the logical segment table for that file. The logical segment number is used to index that table and obtain a pointer to the table of objects for that logical segment. (This pointer is null if the objects are not resident; this is the place where an object fault is detected by Mneme.) Finally, the object number is used to index the logical segment table and obtain a self-relative pointer to the object within the physical segment. This is all

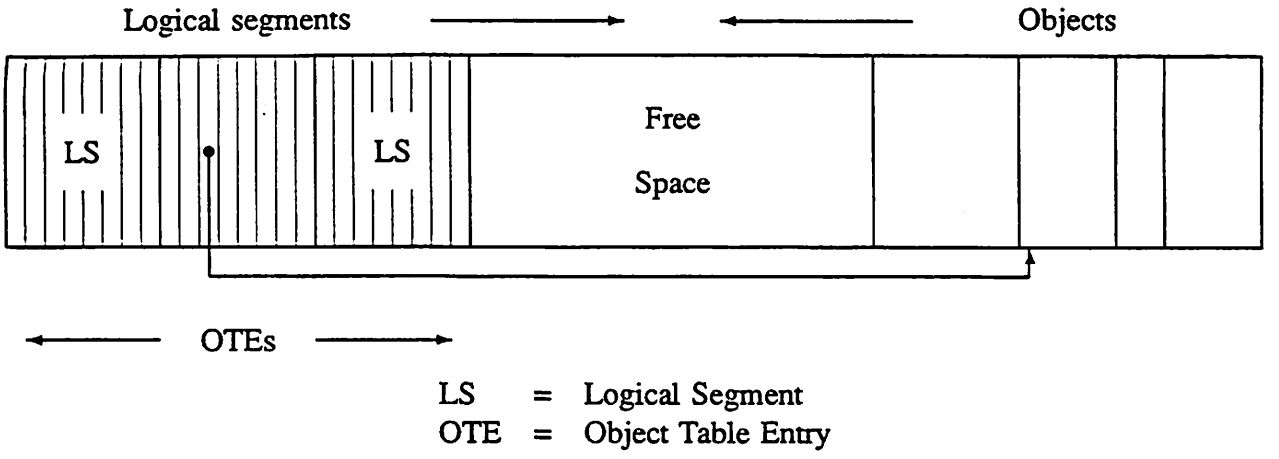


Figure 4: Physical Segment Data Structure

illustrated in Figures 2, 3, and 4. We plan to make a number of changes to this scheme in the next prototype, so it should not be taken as defining the Mneme approach. Still, we do not expect the changes to affect significantly the overall time performance of locating residents objects.

In the experiments, all objects were always grouped in a single file. This avoids overhead from cross file references, etc. It also means that a custom table for looking up the address of an object given the object's id can be a bit faster than the current Mneme implementation, since one can skip indexing the file table. One can also use a direct pointer rather than a self-relative one at the last step (a self-relative pointer is used in Mneme so that no adjustment needs to be made to a physical segment based on its location in virtual memory).

We should also note that the persistent id to client id conversion is quite cheap in the current prototype: simply tack on the 8 file bits of the object identifier. In future prototypes it will not be so simple, since we do not wish to restrict each file to a maximum of a million objects and neither do we wish to restrict one to such a small number of files. Undoubtedly the cost of id conversion will increase in some cases, but we expect to keep the typical cost close to that of the current scheme. We further note that while the call interface to Mneme is restricted to using only client ids (and thus incurs the conversion costs), the pointer interface allows explicit use of persistent ids, qualified by a separate file number when necessary to disambiguate. Thus, the id conversions of Mneme put the call interface at a slight additional disadvantage.

6 Experimental methodology

We first discuss the nature of the object collections that we used for the experiments, and then describe the experiments performed. Finally, we include details of the hardware and software configuration on which we ran the experiments.

6.1 Collections

For the experiments, we used binary trees for the object collections. In these trees we could vary the number of additional slots (beyond the two required for internal nodes) and the number of bytes in the objects; these parameters were controlled separately for the internal nodes and for the leaves. The height of the trees could also be varied. Thus we could vary the number of objects, the total number of slots and bytes, and the overall size of the collection quite easily. Our goal was to discover such statistics as the base cost per object in visiting each node of the tree, the incremental cost of accessing and updating slots, bytes, or integers during a visit, and so forth. By varying appropriate parameters of the binary tree we could indeed determine such behaviors; how the parameters were varied is mentioned with the descriptions of the experiments below.

An important consideration is that the object collections were always sized to fit in primary memory, though they were large enough to be meaningful (generally on the order of 4 megabytes). One reason we did not consider data sets larger than primary memory is that the behavior would be complex and very application dependent, making it difficult to obtain useful results, and certainly difficult to produce a simple predictive model of performance. Another justification is that for certain load/work/save applications, specifically those that work intensively on the whole object collection, performance will be unacceptable unless the collection is memory resident anyway. VLSI design rule checkers may fall into this category. Finally, we suspect that paging will affect all schemes more or less equally, except perhaps for copy swizzling. In any case, we simply did not study object collections that would induce paging in virtual memory.

We also assumed that relatively small objects were the most interesting, extrapolating from average object sizes in heap based programming languages (e.g., Smalltalk [7]), where average object sizes are on the order of 30 to 50 bytes, including overhead and given 32-bit pointers. In most tests we considered sizes ranging from very small (e.g., no extra slots or bytes) up to 1K bytes, using a Fibonacci sequence to choose the sizes. Considering the strong linearity of most of the data we collected, we believe it reasonable to extrapolate the results to objects somewhat larger than those actually measured.

6.2 Experiments

We performed two main categories of experiments, one considering the in-store model and the other considering the various swizzling approaches. In each category we determined the cost of creating objects, the incremental cost of initializing the contents of objects as they are created, the cost of traversing a tree while doing some work at each node, the cost of loading, and the cost of saving. For the swizzling schemes we also considered lazy as well as eager loading, and we measured the cost of preparing as distinct from writing. Thus we have measures of costs of all the phases of our work session model.

We used the following simple calculations to model actual work and initialization of objects.

Operation	Item kind	Calculation Performed
Reading	Slots	bump global counter if slot is null
	Bytes	add to global sum
	Integers	add to global sum
Updating	Slots	swap even-odd pairs of pointers
	Bytes	negate each byte
	Integers	negate each integer
Initializing	Slots	values taken from global array in reverse order
	Bytes	sequential values from global counter
	Integers	sequential values from global counter

These calculations are specifically designed to prevent block moves alone from being adequate for any of the calculations, and thus correspond more closely to application generated record or array accesses. Since our test programs allow the size of objects to be varied from the input script, all work and initialization involves loops rather than straight line code. A final point is that one may read or update (but not initialize) more slots, bytes, or integers than an object possesses; this simply causes one to start over from the beginning of the object until the required amount of work has been performed.

Note that all our measurements (e.g., incremental cost to read a slot) include loop overheads, but these overheads are the *same* for all schemes. Also, every visit to an object involves conditionals such as "if the number of bytes to be updates is non-zero then ...", as well as additions to various measurement counters, which again add uniformly to all schemes. The point is that our implementation of "work" gives a basis for *comparison*, but it is not possible to take the numbers and predict application costs to the microsecond. Since loop and conditional overheads are included, the absolute number give upper bounds on the time for some individual application actions. On the other hand, tree traversal is such a simple algorithm and we are performing no "real" work, so the aggregate time is probably *less* than that needed by a real application, which would be doing more arithmetic, more manipulation of local variables and register contents, and more subroutine calls.

6.2.1 In-store model experiments

Creation. There were five series of tests run for each software configuration. The first series simply varied the number of slots in the objects (both (internal) nodes and leaves) and the second varied only the number of bytes. Here the number of objects varied (by changing the height of the binary tree) so that the total volume of data remained between 2 and 4 megabytes. The other three series of tests varied the number of slots, bytes, and integers initialized, respectively. Here the number of objects and their size remained fixed, with only the amount of initialization varying. The initialization tests used very small nodes and large leaves, and did initialization on

the leaves (this affects interpretation of the results presented later). All these creation tests were performed for non-persistent C code, the Mneme pointer interface, the Mneme call interface, and the Exodus storage manager.[6]

Work. Two kinds of tests were performed. Tree traversal with no additional work was done for each creation case, providing some baseline data on the general overhead of traversal, and a cross check on the tests that added work on each visit to a node. Traversal with work consisted of six series of tests, one each for reading and updating of slots, bytes, and integers. Thus we determined the incremental cost of, for example, reading one more slot or updating one more integer. These tests were done for the same software configurations as the creation tests.

Saving and loading. For saving and loading we were concerned with two primary variations: how the elapsed time varied with total data size, and to what extent the number of objects affected the time, given the same data size. So we performed two series of tests. In the first series we kept the size of the objects the same, but repeatedly doubled the number of objects, and thus the overall data size, over a range of approximately 1/4 megabyte to 8 megabytes. The second series of tests kept the total size of the data at one of the larger sizes, but varied the number of objects by simultaneously doubling object size and reducing tree depth by one. These tests were done for the same cases as the creation and work tests, except that non-persistent C was omitted.

6.2.2 Swizzling experiments

The swizzling experiments were mostly similar to the in-store model experiments; we note only the differences. All experiments were done using the Mneme object store. The in-place scheme used the pointer interface (of necessity); the copy scheme was tested using both the pointer interface and the call interface.

Creation. Object creation is different in that it always creates objects in the application's heap. Therefore, we did not measure it separately for each swizzling scheme. Creation (and initialization) are largely similar to the non-persistent C case, with the exception that slots must always be initialized to some value or the swizzling and unswizzling operations may fail. Also, in the swizzling approach we add a slot to every object to hold the tag bits and the object id (initially unassigned), as previously described.

Work. There are some more combinations to consider. When traversing without working, one can either assume all objects are resident and thus omit the residency check, or one can perform the check. We tested both cases, to determine the cost of checking. Since checking is independent of reading and updating, we measured reading and updating only in the no-checking case.

[6]We do not present detailed performance results for the Exodus storage manager, since it was not designed to be used in this way. We do make some qualitative comments later.

Saving. Since the actual writing of data by the object store is no different, for saving we measured only the cost of preparing (to which one would add the saving cost of the in-store approach to get a total cost). There are three interesting cases for preparing: when the objects are new, and hence need to have an object store version created; when the objects are old but not modified; and when the objects are old but have been modified. We examined these cases similarly to the saving of the in-store approach. Another interesting parameter to vary is the number of initialized (non-null) references within the objects, since these-null references cause swizzling and unswizzling work to be done. We examined this situation by varying the number of initialized slots in leaves; the value used for initialization in each leaf was a reference to the leaf itself (the particular value does not matter, and that value was handy).

Loading. For (eager) loading we varied the total data set size, the number of objects (holding the data set size fixed), and the number of initialized leaf slots (holding the number of objects and object size fixed). We also considered lazy loading (without additional work on the nodes) to see if it had any any significant advantage. As previously mentioned, the Mneme pointer interface provides means to map object ids to addresses. For the in-place scheme we also considered the effect of using the separate table in the same fashion as the copy scheme.

6.3 Hardware and software used

All tests were performed on a DECStation[7] 3100 (Mips R2000A cpu[8]) running Ultrix 2.1 rev 14. The system had 24 megabytes of main memory, of which 10% was used for operating system disk buffers. The disk used for the tests was a relatively empty RZ56 drive (665 megabytes, SCSI); the software used ordinary buffered file I/O (as opposed to the raw disk device). All programs were coded in C and compiled with the Mips C compiler version 1.31 at optimization level 2. The tests were performed in single user mode and the process's address space was locked into main memory. We observed no paging or swapping during the tests. Tests were run once with recording turned off and then repeated three times with recording on; this prevented events such as growth of the virtual address space from affecting the experimental data. Whenever a load was to be performed, either eager or lazy, we first read through a large file, to insure that none of the object collection's data was in the operating system disk cache.

7 Performance results

We consider the experimental results according to phase of the work session model. Overall analysis and comparison is deferred to the next section.

[7]DECStation and Ultrix are registered trademarks of Digital Equipment Corporation.

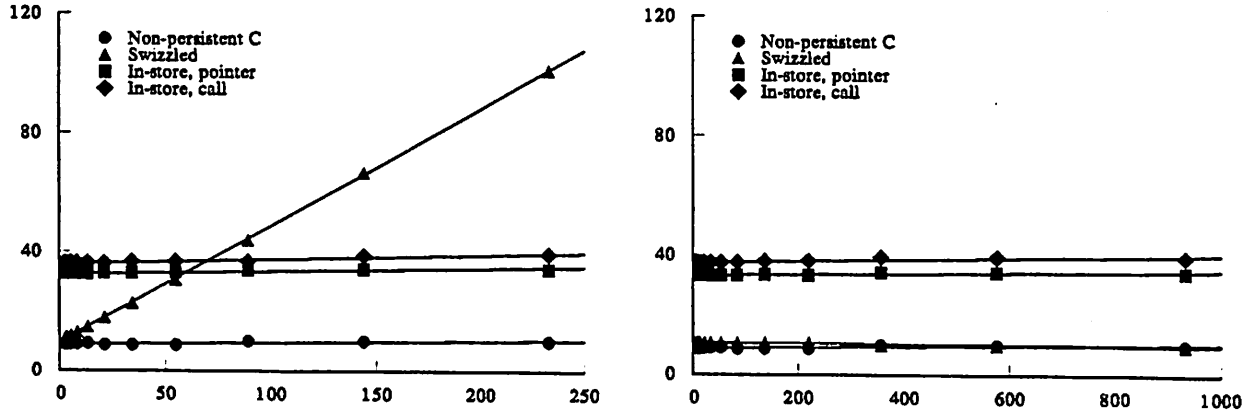
[8]Mips and R2000 are trademarks of Mips Computer Systems.

7.1 Creation

There are a total of four software configurations to consider for the create phase: non-persistent C, in-heap creation for the swizzled cases, and the two Mneme interfaces for the in-store cases. For each configuration we varied the number of slots and the number of bytes (separately) and measured the per-object creation cost. Elapsed and cpu times were identical, and (allowing for poor clock granularity) varied not at all or quite linearly in the size of the objects. The line fits are summarized in Table 3 and displayed in Figure 5.

Software	Slots		Bytes	
	a	b	a	b
Non-persistent C	9.0	—	9.0	—
Swizzled	9.5	0.39	10.3	—
In-store, pointer	32.9	—	33.3	—
In-store, call	36.6	—	37.8	—

Table 3: Creation times: $\mu s/\text{object} = a + bx$



(a) number of slots (b) number of bytes

Figure 5: Creation times: μs/object vs. object size

The in-store lines probably do have a slightly positive slope, but it would require much larger data sets to measure it, and it is clearly small enough to neglect for all but very large object sizes. The base cost of the swizzled form was slightly greater than the non-persistent case because the header must be initialized; the positive slope of the slots line is because the slots must be initialized. The in-store lines are flat because the space used for allocating new Mneme objects is guaranteed to be zeroed in advance, so the object allocation code does not need to clear the slots. The slightly greater base cost for a swizzled bytes object versus a swizzled slots object reflects the cost of initializing the two slots in every object.

It is clear that the non-persistent and swizzled forms cost less than the in-store forms except for moderately large numbers of slots, and that the non-persistent and swizzled forms are reasonably close (especially if the non-persistent code initialized all slots). We should note that we did not use the C malloc routine to allocate heap objects, but our own allocator that allocates linearly from large (32 Kbyte)[9] blocks. This allocation routine was used was used for both the C and swizzled cases, is substantially faster than malloc, and incurs less space overhead.

Tests of creation with initialization also resulted in very straight lines, though with more noticeable slopes, as summarized in Table 4 and Figure 6. This clearly shows the effect of the call interface (versus the pointer interface) clearly: it costs 50 to 100% more to move data into the objects across the call interface. For purposes of measuring the true cost to initialize a slot (byte, integer) this data must be interpreted with care: not only does it include loop overhead and other calculation in addition to the actual store instructions, but the time quoted is divided by the total number of objects rather than just the number of leaves, so the cost of one iteration of the non-persistent slot initialization loop is really 0.62μs, for example.[10] The swizzled initialization time is flat because the swizzled form always initializes all slots; it is large because the objects used in this test had 233 slots. It should be noted that if all schemes were required to initialize all slots, then the swizzled scheme would be the fastest persistent technique, only slightly slower than non-persistent C.

Software	Slots		Bytes		Integers	
	a	b	a	b	a	b
Non-persistent C	8.9	0.19	9.1	0.18	9.0	0.19
Swizzled	56.2	0.00	10.5	0.18	9.9	0.20
In-store, pointer	34.7	0.20	33.7	0.19	33.5	0.20
In-store, call	37.9	0.51	38.8	0.28	38.4	0.40

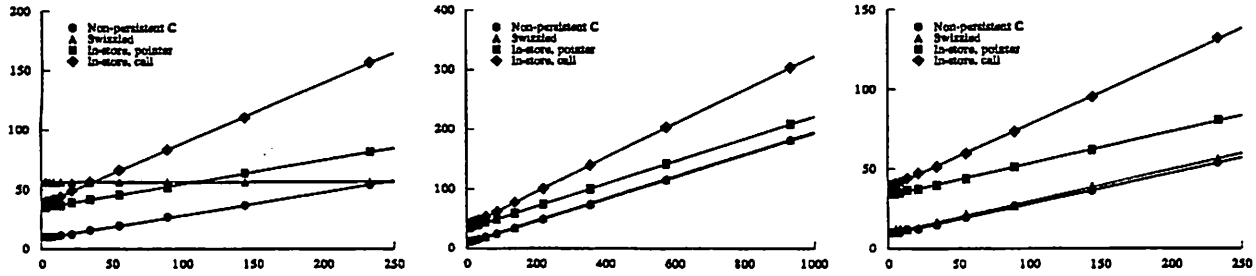
Table 4: Creation with initialization: μs/object $= a + bx$

7.2 Working

First we consider traversing the tree without any work, as a baseline for comparison. We have five software configurations: non-persistent C, swizzled with and without residency checking, and the two Mneme interfaces for the in-store approach. In each case we varied the number of slots and the number of bytes, expecting little variation in the per object cost. This expectation was borne out, so we omit slopes from Table 5, which presents the results.

[9]We will use K and M to mean 1024 and 1024 × 1024, respectively, and kilo and mega to mean 1000 and 1,000,000.

[10]This adjustment is required when interpreting *any* of the experiments where the number of initialized slots was varied.



(a) slots initialized (b) bytes initialized (c) integers initialized

Figure 6: Creation with initialization: μs/object vs. amount initialized

Software	Slots	Bytes
Non-persistent C	11.7	11.7
Swizzled, no check	12.6	12.1
Swizzled, check	13.3	13.1
In-store, pointer	20.1	20.2
In-store, call	22.1	22.2

Table 5: Traversal without work: μs/object

The swizzled form without checks is well within 10% of the non-persistent form, with the variation probably resulting from the fact that the swizzled form marks the object as having been modified, which the non-persistent code does not do. Residency checking has an overhead of less than 10% in the context of this traversal, and its relative impact would diminish in the presence of real work. The in-store forms show a substantial penalty: 65% to 85% higher than swizzled without checks.

We expected linear fits when we added work, with the intercepts as in Table 5 and the slopes giving us incremental costs of work. There was an interesting effect we did not anticipate, however, though it could have been predicted: the incremental cost of working is higher for the first pass through an object. When we do enough work to require multiple iterations over each object, the incremental cost drops. We suspect this is caused by the data cache associated with the cpu. In any case, we thus get *two* lines in each case, with the breakpoint determined by the size of the objects. We present the slopes and intercepts of the two lines for each case in Table 6, along with the break point between them, with Figure 7 illustrating the results.

Again, swizzled objects have essentially the same performance as non-persistent objects (a cross check), the Mneme pointer interface pays a flat per-object penalty of about 8μs for locating objects, and the call interface has a higher penalty since it cannot operate on data in place but must fetch it into a scratch buffer (and also write it back for updates). The intercepts of the lines fit to the first data points ($x \leq n$) are reasonably consistent with the cost of traversing without

Software	Reading				Updating			
	a	b	c	d	a	b	c	d
Slots, $n = 34$								
Non-persistent C	12.3	0.41	18.1	0.25	11.8	0.44	19.8	0.19
Swizzled	12.7	0.42	18.4	0.25	12.5	0.44	20.4	0.19
In-store, pointer	20.7	0.42	26.6	0.25	20.3	0.44	28.5	0.19
In-store, call	24.3	0.83	44.4	0.25	25.7	1.06	55.5	0.20
Bytes, $n = 84$								
Non-persistent C	12.2	0.20	15.8	0.15	12.1	0.51	12.1	0.51
Swizzled	12.6	0.20	16.2	0.15	12.7	0.52	12.8	0.51
In-store, pointer	20.8	0.20	24.2	0.15	20.7	0.51	20.8	0.51
In-store, call	25.0	0.25	32.7	0.16	27.7	0.62	36.6	0.52
Integers, $n = 21$								
Non-persistent C	12.2	0.32	16.8	0.16	12.0	0.52	18.8	0.19
Swizzled	12.6	0.32	16.1	0.16	12.6	0.52	19.5	0.19
In-store, pointer	20.8	0.33	24.4	0.16	20.8	0.50	27.5	0.19
In-store, call	25.2	0.51	32.9	0.16	27.7	0.72	39.1	0.20

Table 6: Traversal with work: μs/object $= a + bx$ (for $x \leq n$) or $c + dx$ (for $x > n$)

work; they are a little higher because of the fixed cost of calling the subroutine that does the work. In all cases the non-persistent and swizzled forms are quite similar, and dominate the Mneme pointer form, which in turn dominates the Mneme call interface. The slight differences between the non-persistent and swizzled forms are probably explained by checking and manipulation of the "modified" bit, as before.

7.3 Saving

The writing aspect of saving, which is the only aspect for the in-store approach, gives quite simple results: the time is strongly affected by the total volume of data written (including Mneme header information of 8 bytes per object), and not affected much at all by the number of objects. We conclude that writing is strongly I/O bound. The average write rate was 321 kilobytes/sec. For the (rather small) objects used, this amounts to about 20,000 objects per second. This is most of the available disk bandwidth.[11] Writing performance is illustrated in Figure 8. The peak in the write rate curve is probably due to the operating system disk buffer cache (size 2.4 Mbytes), with 300 kilobytes/sec being the sustained rate.

The prepare aspect of saving is more interesting in that it exhibits more variation. Only the

[11]We have seen unsubstantiated claims of 350 kilobytes/sec maximum read performance for these disks, and sustained write rates could not be very much different.

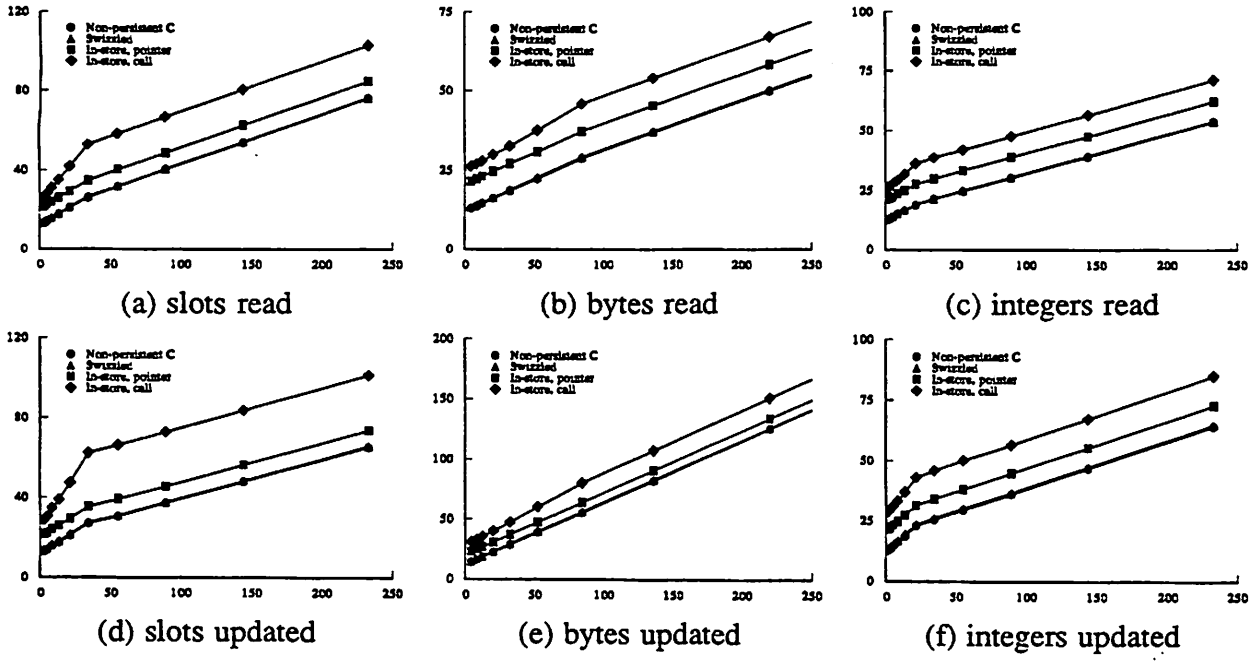


Figure 7: Traversal with work: μs/object vs. number of items

swizzling approach prepares, so we are concerned with the in-place scheme and the two versions of the copy scheme (pointer and call interface). We are also concerned with three distinct kinds of collections to be saved: new objects, unmodified old objects, and modified old objects. As with creation, we varied the number of slots and the number of bytes. We did multiple regression fits expecting the total time to be determined by the number of objects, the total number of bytes (including slot data and overhead), and the total number of slots (slots per object times number of objects). We got excellent fits in all cases. The results are shown in Table 7. The constants are not significantly different from zero so we omit them.

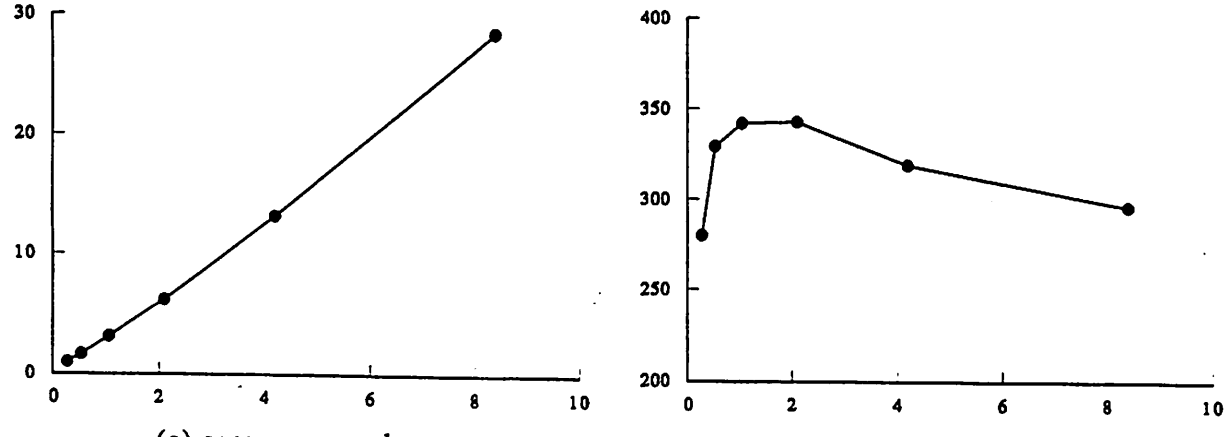


Figure 8: Writing times

Software	New			Unmodified			Modified		
	a	b	c	a	b	c	a	b	c
In-place	0.20	27.5	0.43	—	4.4	0.77	—	6.6	0.77
Pointer	0.21	30.4	0.41	—	4.6	0.76	0.20	12.7	0.40
Call	0.22	39.0	0.58	—	4.3	0.77	0.20	21.4	0.58

Table 7: Prepare times: seconds = $(a \cdot \text{bytes} + b \cdot \text{objects} + c \cdot \text{slots}) \times 10^{-6}$

New objects must always be created and copied into Mneme, and the byte copying has similar cost in all cases. The in-place and pointer schemes are quite similar in performance since they are doing almost exactly the same things. The call scheme is more costly per object and per slot, because the call interface. Unmodified objects require only a traversal (to check that they are not modified and to reach all objects in the collection), which has some per object and per slot cost, but no per byte cost. Hence all schemes cost almost exactly the same for these objects. Modified objects highlight a major difference between in-place and copy swizzling: the copy schemes must copy the entire object back into the object store. The copy schemes also have higher per object overhead than the in-place scheme. The lower per slot cost of the copy schemes presumably results from better cache performance since the copying step "warms" the data in the cache. Another way to look at the numbers is to realize that since there are 4 bytes per slot, there is an additional cost of 0.80 μs per slot for the copy schemes.

The in-place scheme is always fastest, ignoring differences in the last decimal place that are probably within experimental variation. Similarly, copy swizzling using the call interface is always the slowest (if we neglect the minor difference between 4.6 and 4.3).

The results just quoted showed the slot cost when the slots were empty (except for the slots containing tree pointers, of course). What about when they contain pointers that must actually be converted? To find out, we used a fixed number and size of objects, but varied the number of initialized slots; the results appear in Table 8. These were large objects (233 slots), so the intercept should be interpreted with that in mind. The incremental cost of converting a slot can be found by doubling the slope from the table (we varied only the leaves, not the internal nodes). These results are illustrated in Figure 9.

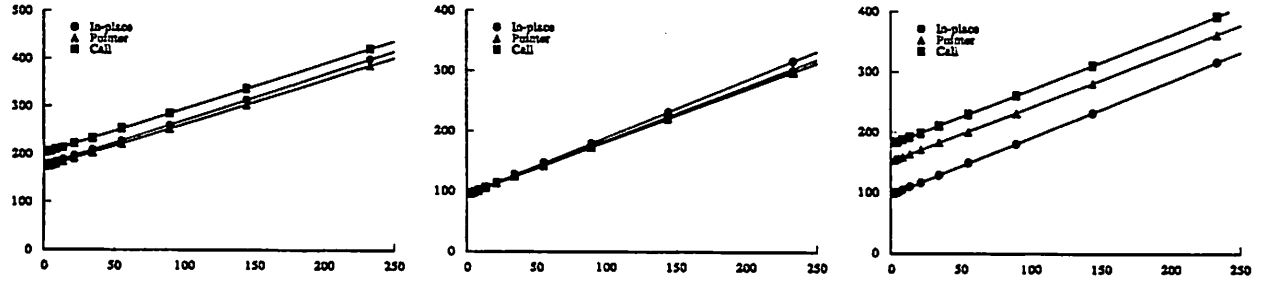
While the line slopes show a little variation, the essential result is that unswizzling *per se* has the same cost in each scheme, with the cost variations resulting from other factors such as per object processing overheads and copying of modified objects back to the store.

7.4 Loading

Finally we turn to the loading phase. For the in-store approach, when we fixed the size of the objects and varied their number so as to vary total data set size, the load rate climbed rapidly

Software	New		Unmodified		Modified	
	a	b	a	b	a	b
In-place	175	0.95	94	0.95	97	0.94
Pointer	171	0.91	94	0.90	152	0.91
Call	201	0.93	93	0.88	180	0.91

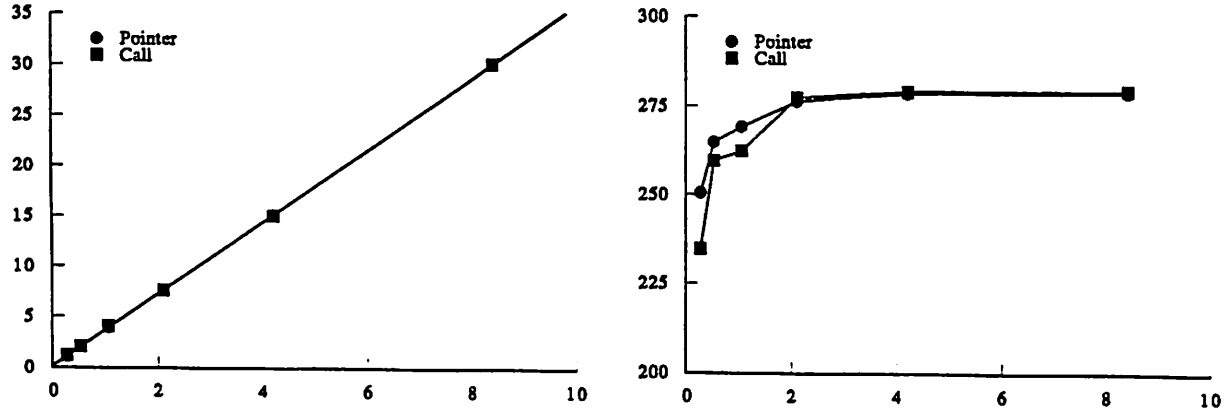
Table 8: Prepare times varying non-empty slots: $\mu s/\text{object} = a + bx$



(a) new objects (b) unmodified old objects (c) modified objects

Figure 9: Prepare times for objects with 233 slots; μs/object vs. slots initialized

to a maximum value. For both the pointer and the call interface that rate was 279 kilobytes/sec, or about 17,500 objects/sec, for the size of object used (small). This is shown in Figure 10.



(a) secs vs. megabytes (b) kilobytes/sec vs. megabytes

Figure 10: Load times

When we kept the size of the data set fixed and varied the number of objects, we saw more clearly that the number of objects affected the total time, presumably by adding to the cpu time of the traversal required to do the loading. A multiple regression analysis showed that bytes and number of objects both have a linear effect, and that the pointer and call interfaces have nearly identical performance. The regression equations are presented in Table 9. The constant terms were not significantly different from zero, so are omitted. These results show that Mneme

sustains an estimated raw read rate of 567 kilobytes/second and that this rate is effectively cut in half by object processing when objects are quite small (16 bytes or so). We note, though, that the disks and software used probably cannot deliver data at the estimated raw read rate. In fact, we are probably already using about 80% of the available disk bandwidth and getting good overlap of disk access and cpu time. With larger objects we believe we come close to the I/O limits of the configuration.

Software	a	b
Pointer	1.767	28.6
Call	1.762	28.5

Table 9: Load time: $\mu s = a \cdot \text{bytes} + b \cdot \text{objects}$

For the swizzling cases, we must also take into account the total number of slots in the data set. There are four cases: in-place, in-place using the explicit id-to-address lookup table, copy using the pointer interface, and copy using the call interface. We also have the additional variation of lazy loading, which combines loading and working. The multiple regression fits are quite good and are presented in Table 10.

	Eager loading				Lazy loading			
Software	a	b	c	d	a	b	c	d
In-place	—	2.0	13	—	—	2.3	25	0.36
Table	0.5	1.7	15	0.43	—	2.2	26	0.71
Pointer	—	2.2	12	—	—	2.5	29	0.36
Call	—	2.3	19	—	—	2.6	35	0.67

Table 10: Load-and-swizzle time: $\mu s = a \cdot 10^6 + b \cdot \text{bytes} + c \cdot \text{objects} + d \cdot \text{slots}$

Copy swizzling using the call interface is clearly slower than using the pointer interface, and the in-place scheme is faster than the copy scheme. Neither of these results is particularly surprising. Using the explicit table with the in-place scheme has some complicated interactions, with some cost factors increasing and others decreasing, but the net difference between the two schemes is slight.

As with preparing, we also considered the effect of varying the number of non-null slots. We have the same situations (scheme, eager vs. lazy) as in the loading tests just presented, but the only significant variable is the number of non-null slots since the size and number of objects were held fixed. The line fits are shown in Table 11 and illustrated in Figure 11.

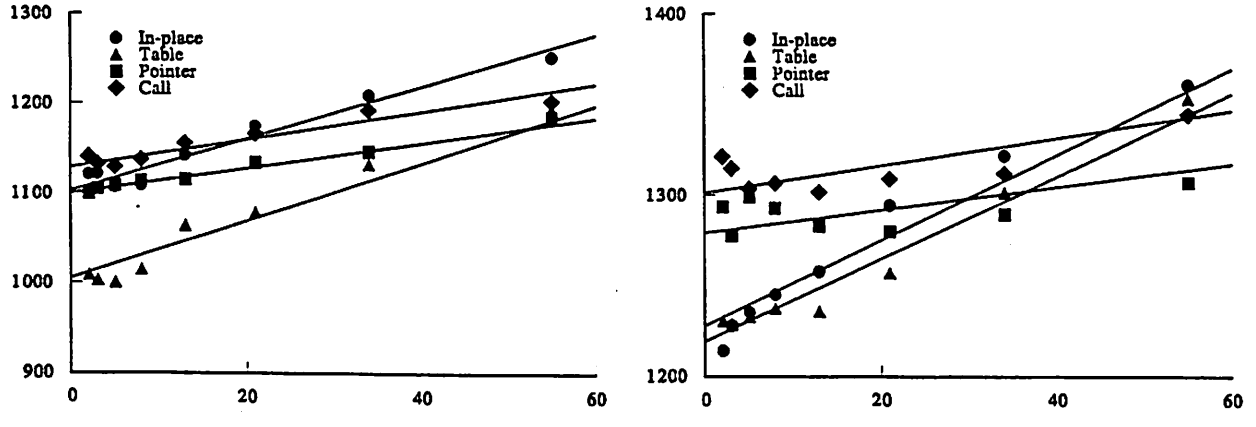
The *incremental* costs of the copy swizzling pointer and call interfaces are likely lower because *all* the slots are copied before swizzling (the cost of which is folded into the constant

Software	Eager		Lazy	
	a	b	a	b
In-place	1103	2.85	1228	2.37
Table	1005	3.17	1219	2.29
Pointer	1100	1.36	1279	0.64
Call	1129	1.51	1301	0.76

Table 11: Load time varying non-empty slots: μs/object $= a + bx$

factor) and hence are likely to be in the cache. As expected, the pointer interface outperforms the call interface. We would expect the copy scheme using the direct interface to be slower than the in-place scheme, because of the copying, but in fact the in-place scheme is slower. This is because the copy scheme always uses a custom table to map object ids to addresses of copies in the heap, and the custom table is faster than the Mneme object location routine. When we use a similar table with the in-place scheme, we get a noticeable improvement in mapping ids of already swizzled objects (i.e., objects that have been entered in the table). As can be seen from the lazy loading graph (Figure 11 (b)), the table helps there as well, though the different overlap of I/O and execution, as well as (we speculate) effects of the processor's instruction cache, give a less dramatic improvement for the table scheme. Even more interesting, the copy scheme with the pointer interface eventually gives better performance than the in-place scheme, a fact for which we have no definitive explanation, only a speculation that the copy scheme may have better code and data access locality than the in-place scheme, which makes Mneme calls to locate resident objects.

Another interesting aspect of these measurements is that the lazy loading costs are noticeably higher than for eager loading. Even if we subtract off the per-object cost of a traversal (which is



(a) eager loading (b) lazy loading

Figure 11: Load time varying non-empty slots: μs/object vs. slots initialized

happening in the lazy case in an interleaved fashion), the lazy scheme is still somewhat slower. We believe that this has to do with poorer code locality in the cache (there were *no* page faults) and with the interleaved code execution causing additional missed disk rotations, resulting in more I/O wait time with no useful work. The clock granularity makes it difficult to develop a detailed and definitive explanation, but it is clear that the interleaved execution of lazy loading hurts performance.

8 Analysis

We can draw a few conclusions without detailed numeric analysis. First, the Mneme call interface almost always performs worse than the pointer interface and never performs substantially better. Second, swizzling is better than the in-store approach if the amount of work is sufficient to overcome the cost of swizzling. Copy vs. in-place swizzling and lazy vs. eager loading are more complex issues requiring detailed analysis. The experiments we performed allowed us to determine the constants for the predictive performance model presented in Section 4. Before considering the numeric performance model, though, we have some comments regarding interfaces to object stores.

The Mneme call interface was never substantially better than any of the other techniques, and was frequently substantially worse. It is clear that the protective layer of subroutine calls, and perhaps more importantly the need to move data between the objects and scratch buffers to do any computation, cause a substantial performance impact. This is not unlike results in the networking area where it has been found that copying the data part of packets or RPC arguments affects performance noticeably. Similar experience exists in the operating system community (see [8, 9, 10]) concerning copying between user and operating system I/O buffers, and database researchers have also commented upon upon this issue (e.g., [6]). In sum, a call interface makes the in-store approach considerably less attractive than does the pointer interface.

This was confirmed by our tests of the Exodus storage manager (ESM), which allows pointer access for reading but requires that updates be gathered in a scratch area and block transferred into the objects. ESM showed good incremental performance only on reading, and its larger object ids (12 bytes compared with Mneme's 4 bytes) and object headers only made matters worse. ESM performed better only on writing under the measure of kilobytes/sec (it was slower on objects/sec). The point is not a competition between Mneme and ESM (which is part of why we chose not to present detailed results)—ESM was designed to support database access patterns (e.g., sophisticated buffer management) as opposed to a work session model of computation. Rather, the point is that the ESM results confirm that a call interface is not as good for this task as a pointer interface. We also note that the size of ids has a noticeable impact on performance. This is of relevance to persistent programming and applications that conform well to our work session model, but may not carry over well to database applications with short work sessions

and heavier requirements for searching and buffer management.

To understand the tradeoffs further, to discriminate between the various swizzling techniques, and to predict work session performance, we determined the values for the constants of the predictive performance model of Section 4, using the experimental results presented in Section 7. We note that single and multiple regression analyses of the experimental results validate the performance model and leave very little unexplained variance in the data, generally a fraction of 1%.

Recall that the model is expressed by these equations:

$$W = w_1 + w_2 nv + nvw$$
$$M = m_1 + m_2 nb + m_3 n + m_4 npb + m_5 npbi + m_6 nbu + m_7 nu + m_8 npbu + m_9 npbiu$$
$$T = W + M$$

Further, recall that we neglect the constants w_1 and m_1 since they are not important for significant values of n. We will present and discuss in turn quantitative models of eager loading, lazy loading, and sessions that create a collection rather than loading an existing collection.

8.1 Eager loading

For eager loading (and lazy loading, too) we derived the cost model constants for four schemes: IS, in-store; IP, in-place swizzling with Mneme object lookup; T, in-place swizzling with the separate lookup table; and C, copy swizzling. Since the Mneme pointer interface was superior to the call interface, all four schemes use the pointer interface. Table 12 presents the m_i for eager loading under these schemes.

Software	m_2	m_3	m_4	m_5	m_6	m_7	m_8	m_9	w_2
IS	1.8	29	—	—	3.4	—	—	—	21
IP	2.0	17	0.19	1.9	3.4	6.2	—	—	13
T	1.7	20	0.28	2.1	3.4	6.2	—	—	13
C	2.2	17	0.19	1.1	3.6	8.1	-0.09	—	13

Table 12: Cost model constants, eager loading, μs/unit

From this compilation it is clear that m_2, the coefficient of nb (the total size of the data) does not vary much, and neither does m_6, the coefficient of nbu, but the difference may be significant in comparing copy swizzling with other schemes. These terms represent the volume of data read from and written to the store, which does not vary from scheme to scheme; they also include object copying costs, which are relevant to the copy swizzling scheme.

The tradeoff between the explicit table and Mneme lookup forms of the in-place scheme is made more clear: the explicit table costs more per object (m_3), per slot (m_4), and per initialized

slot (m_5), but saves a little on per byte costs. Since the measurements for some of the results are from objects consisting mostly of slots, there is probably not a lot of significance is the shift of costs between bytes and bytes devoted to pointers, etc. As we saw in Section 7, the explicit table does speed things up some, but the relative speedup is not dramatic. That is, the IP and T schemes offer similar performance. Since, as previously argued, IP is probably more realistic than T, we will not consider the T scheme further in this analysis.

We now compare IS and IP to understand the tradeoff involved in performing swizzling at all. It is clear that IS will cost more if v is large enough. (Note that the nvw term can be neglected since all schemes we are considering provide direct manipulation of objects, either in the Mneme buffers or in the application heap, and hence that aspect of work costs the same in all schemes of interest.) IS and IP have the same performance when:

$$v = 1.17 + 0.76u + 0.025b + 0.26pb$$

Note that we are setting i to 1 under the assumption that most slots actually contain pointers and that most of the pointers are not null. This will tend to stress all the schemes the most, and it helps the analysis to remove variables where we can. Now since $0 \leq u \leq 1$, it is clear that u does not have a major impact on the comparison. Therefore we fix u at 1, giving $v = 1.93 + 0.26b(p + 0.10)$. We plot $\log_{10} v$ against b for several values of p in Figure 12.

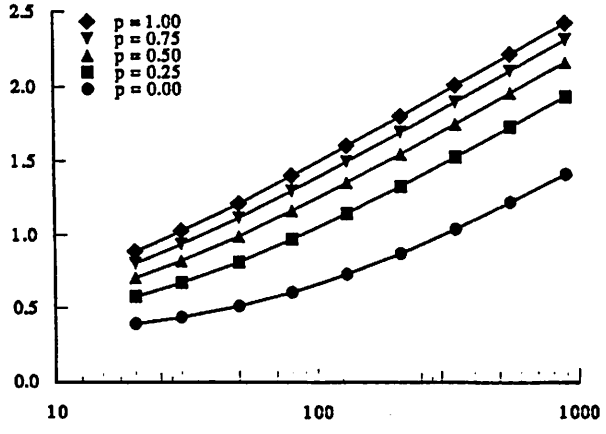


Figure 12: Locus of IS = IP (eager): $\log_{10} v$ vs. b for selected values of p

Examining the figure, we see that, depending on b and p, the number of visits required before IP is cheaper than IS varies from 2 to 200, and increases as either b or p increases. If we take $b = 50$ as "typical", then 10 ± 5 visits is where the cost functions cross. While application characteristics do determine which scheme performs better, the crossover is frequently realized with a relatively small number of visits (but more than one or two). Another observation is that the costs of the schemes are not usually drastically different, so optimal choice is important only in extreme cases.

We now compare the two swizzling techniques. First, note that v does not affect their relative performance, while b, p, and u do. This means that we can expect the same work performance from either swizzling scheme, and the differences will be in loading and saving time. The load/save times (μs/object), setting $i = 1$, are given by these equations:

$$\begin{aligned} \text{IP} &= 17.4 + 2.0b + 2.09pb + u(6.16 + 3.4b) \\ \text{C} &= 17.0 + 2.2b + 1.32pb + u(8.12 + 3.6b - 0.091pb) \end{aligned}$$

From these equations, we can derive curves for IP $= k$C in (p, u) space, given specific values for b and k. In Figure 13 we show such curves for $k = 0.95, 1.00, 1.05$. This shows that under a wide range of conditions, the performance of load/save for the two schemes is within 5 to 10%. The graph also indicates under what conditions one scheme will outperform the other, but we regard the closeness of their time performance as being more significant, indicating that is does not matter all that much, on the basis of time performance, which approach you take. The copy approach requires more space, but allows more separation between the buffer manager and the application, at a possible additional expense of re-fetching buffers to merge changes, an expense we have not modeled.

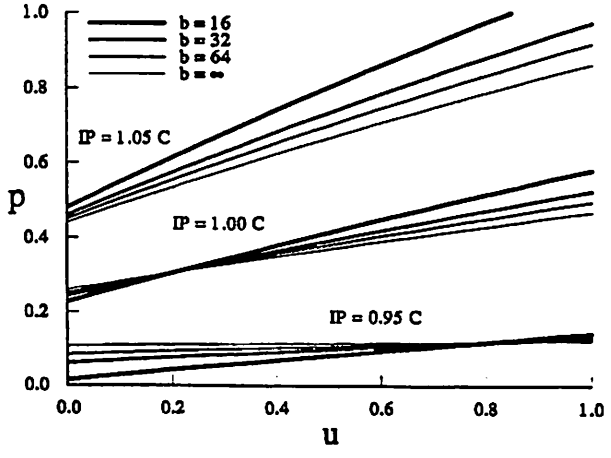


Figure 13: Curves for IP $= k$C (eager)

8.2 An example

Let us consider a specific case to see how dramatic the differences between the schemes are. Suppose we have $n = 100,000$ objects, with an average object size b of 40 bytes including overhead,[12] and an average of 5 slots per object ($p = .5$), with $i = 1$. Let us further assume that $u = .1$, corresponding to a moderate amount of editing.

[12]The in-store method actually saves 4 bytes per object since it does not store the object id in the objects, but this difference is an artifact of the current prototype so we will not penalize the other approaches because of it. Further, though it would change some details, it would not affect the general nature of the results.

Then the load plus save time is predicted to be 9.3 seconds for IS, 14.6 for C, and 15.3 for IP, taking into account the visit avoided in the IS scheme, which would have cost 2.1 seconds. Now if we add work of $w = 20$ μs/object/visit, and 10 visits, the work cost is 39 seconds for IS and 33 for the swizzling schemes, giving totals of 48 seconds for each of the three schemes. This illustrates that the differences are not dramatic in a "typical" case, even though extreme cases exist. If we increase the update fraction to $u = .5$, then we obtain load/save times of 14.8 seconds for IS, 20.7 for C, and 21.0 for IP, giving total times of 54 seconds each.

It is clear that combined load and save time (9 to 21 seconds) is on the order of 10 times the cost of a no-work traversal (1 to 2 seconds) in this example. Further, the load and save times, while not really fast, are not unreasonable periods for a user to wait at the beginning of an edit session that will last for a while, though larger data sets lead to less pleasant delays. Although the copy swizzling scheme offers some advantages for small u, the ability to get more user data into the same memory space probably tips the balance in favor of the in-place scheme, and for any very interesting amount of work on an object collection, the swizzling approach appears better than the in-store approach.

8.3 Lazy loading

Let us consider lazy loading. Performance is determined by the same parameters as the previous cost model, though some of the constants require adjustment (specifically, m_2 through m_5), and the nv term should be replaced by $n(v-1)$ in the swizzling cases to account for the fact that loading is accomplishing a traversal. We are assuming that additional work, w_3, cannot be effectively overlapped with loading, i.e., that combining would not reduce the total time. This is reasonable since the number of objects and other cpu time related factors have substantial bearing on load time, but the assumption needs to be verified in the future, especially in light of the apparently complex interactions resulting from overlap of cpu and I/O in the lazy loading case. The new cost factors for the various schemes appear in Table 13 (the in-store numbers are actually the same as before; the in-store approach is intrinsically lazy).

Software	m_2	m_3	m_4	m_5
IS	1.8	29	—	—
IP	2.3	30	0.28	1.7
T	2.2	31	0.37	1.6
C	2.5	33	0.28	0.8

Table 13: Cost model constants, lazy loading, μs/unit

We still have the same general tradeoffs: IS is worse for any significant amount of work, C swizzling is better than IP depending on u, and so forth. As with eager loading, we can develop

equations determining IS, IP, and C costs per object in terms of b, p, u, and v (ignoring w, which is the same for all cases), setting $i = 1$:

$$\begin{aligned}
\text{IS} &= 8 + 1.8b + 21v + u(3.4b) \\
\text{IP} &= 17 + 2.3b + 1.9pb + 13v + u(6.2 + 3.4b) \\
\text{C} &= 21 + 2.5b + 1.1pb + 13v + u(8.1 + 3.6b - 0.09pb)
\end{aligned}$$

Comparing IS and IP as we did for eager loading yields Figure 14, which is quite similar to Figure 12. The primary difference is that the range of v values is a bit narrower as p varies.

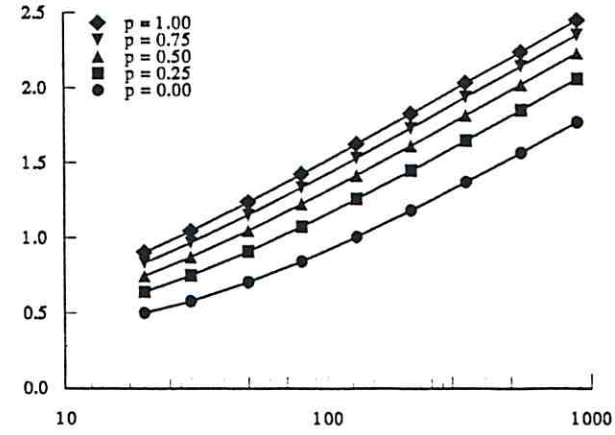


Figure 14: Locus of IS = IP (lazy): $\log_{10} v$ vs. b for selected values of p

Likewise, we compare IP and C in Figure 15 and see results similar to those for eager loading.

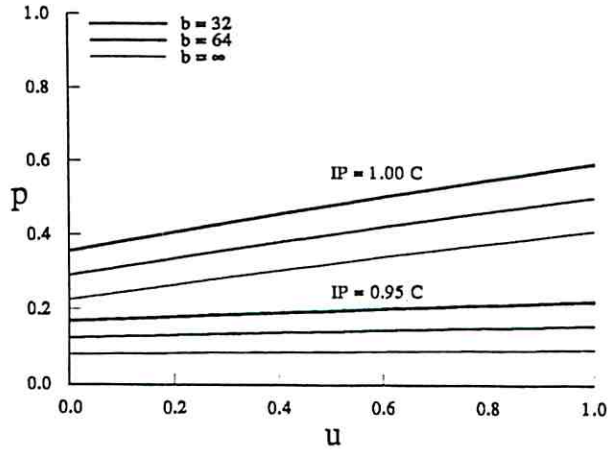


Figure 15: Curves for IP = kC (lazy)

8.4 Comparing eager and lazy loading

Now we compare overall eager and lazy loading performance. Of course, IS does not vary; the table below combines previous tables and equations to show the IS, IP, and C cost equations

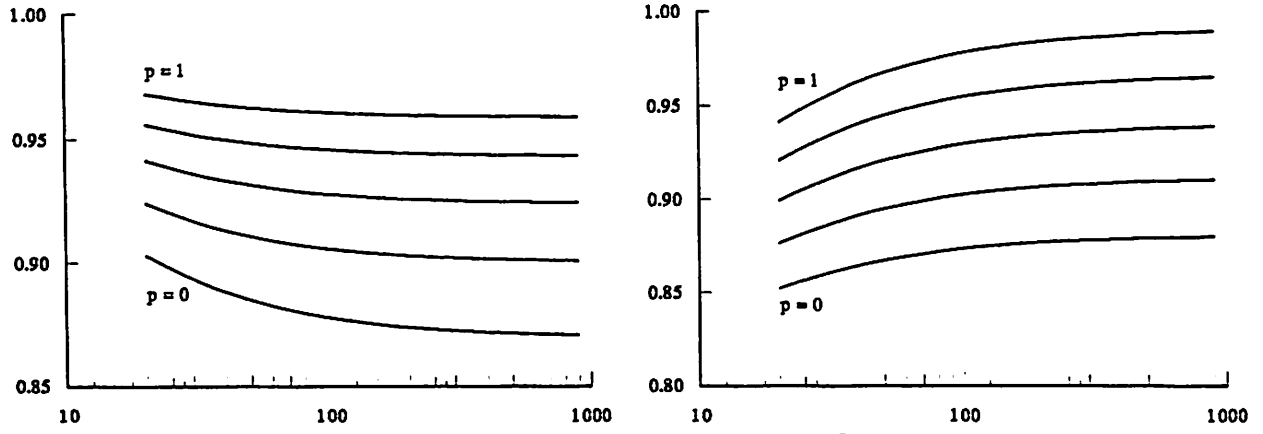
(μs/object) in one place. (We use subscripts E and L for *eager* and *lazy*.)

$$
\begin{aligned}
\text{IS} &= 8 + 1.8b + 21v + u(3.4b) \\
\text{IP}_E &= 17 + 2.0b + 2.1pb + 13v + u(6.2 + 3.4b) \\
\text{IP}_L &= 17 + 2.3b + 1.9pb + 13v + u(6.2 + 3.4b) \\
\text{C}_E &= 17 + 2.2b + 1.3pb + 13v + u(8.1 + 3.6b - 0.09pb) \\
\text{C}_L &= 21 + 2.5b + 1.1pb + 13v + u(8.1 + 3.6b - 0.09pb)
\end{aligned}
$$

To compare eager and lazy loading, it is helpful to consider the load time part of the equations alone:

$$
\begin{aligned}
\text{IP}_E &= 13.0 + 2.0b + 1.43pb \\
\text{IP}_L &= 12.7 + 2.3b + 1.28pb \\
\text{C}_E &= 12.4 + 2.2b + 0.68pb \\
\text{C}_L &= 16.2 + 2.5b + 0.41pb
\end{aligned}
$$

Since these equations depend only on p and b, it is easy to plot ratios of eager to lazy load time (we have already compared IP to C in each case), which are shown in Figure 16. Eager loading is faster in both cases, by 2 to 15%. For $p = 0.5$ eager loading is about 7% faster.



(a) $\frac{\text{IP}_E}{\text{IP}_L}$ vs. b for various p (b) $\frac{\text{C}_E}{\text{C}_L}$ vs. b for various p

Figure 16: Ratios of eager and lazy load times

We note that these comparisons take into account the fact that lazy loading allows a traversal to be overlapped and thus reduces by one the number of application visits to each node. That lazy loading is slower overall is surprising, since we would expect the opportunity for cpu and I/O overlap would improve performance rather than degrade it. As previously mentioned, we have no detailed explanation, but suspect that lazy loading has less cache locality, and that the additional cpu time during loading may cause more missed disk rotations. The magnitude of the difference could probably be explained by either effect alone, and the coarse granularity of the system clock makes it difficult to resolve the issue. In any case, the effect is not a major one, though perhaps significant for some applications or users.

Let us consider the earlier example. The resulting times are summarized in the following table, which clearly shows the equivalence of the schemes for this data set and work parameters:

Phase	$u = .1$			$u = .5$		
	IS	IP	C	IS	IP	C
Eager load/save	9	15	15	15	21	21
Lazy load/save	9	16	16	15	22	22
"Work" (v, w terms)	39	33	33	39	33	33
Eager total	48	48	48	54	54	54
Lazy total	48	49	49	54	55	55

8.5 Creation

As described at the end of Section 4, the creation model is quite similar to the load/work/save model, except that we fix $u = 1$, so we require only m_2 through m_5. The work costs are the same as for a load session, and of course creation substitutes for loading. However, we must also use appropriate prepare phase measurements: the ones for new objects. The measurements of Section 7 result in the constants presented in Table 14. We note that IP and T are the same (since there is no loading), which is why T is not presented separately in the table.

Software	m_2	m_3	m_4	m_5	w_2
IS	3.4	33	—	0.10	21
IP	3.6	37	0.21	0.47	13
C	3.6	40	0.20	0.46	13

Table 14: Cost model constants, creation, μs/unit

It is clear that swizzling leads to higher creation costs if work is ignored. This is to be expected, since swizzling creates objects first in the application heap and then must copy them into the object store. However, since object reclamation undoubtedly costs more within the object store than in the transient heap area, any practical scheme is likely to create objects in the transient heap and thus effectively act like a swizzling scheme for creation. We first note that IP and C are within 1 to 2.5% of each other, with IP slightly better. Experimental error was small, but it is fair to say that the difference between IP and C is not of great significance. Comparing IS and IP is more interesting. If we consider just the create/save costs, IP is 6 to 18% faster than IP, as shown in Figure 17. Figure 18 shows how doing work affects the relative costs of IS and IP, and shows that, depending on p and b, it takes 1 to nearly 200 visits before IP's performance matches that of IS. For $b = 50$ the range is 2 to 4 visits.

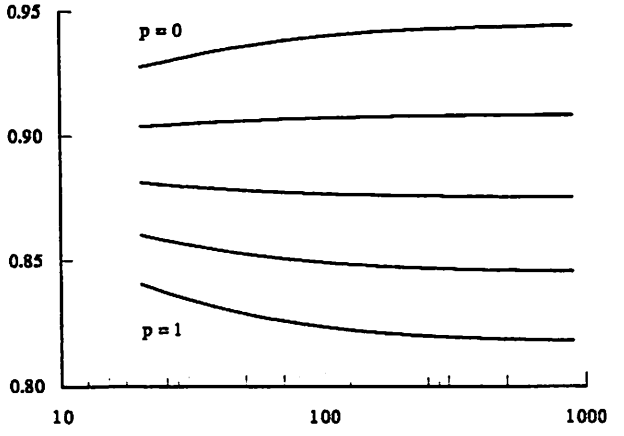


Figure 17: $\frac{IS}{IP}$ vs. b for various p (create/save only)

9 Summary and conclusions

We articulated a model of working with objects, introducing the notions of *collections* of objects, and *work sessions*, consisting of distinct phases with particular activities in each. We described several methods for managing objects, and experiments designed to gather data so as to evaluate their behavior and predict their performance. The experimental results support a simple and clear work session performance model, which allows some general statements to be made about object management and the Mneme object store.

First, direct access to the object manager's I/O buffers improves performance considerably; keeping object identifiers short helps, too. Second, swizzling approaches outperform in-store approaches unless the amount of work performed in a session is rather small; on the other hand, the in-store approach using the Mneme object store's direct access features is not dramatically more expensive during work than the swizzling approach. Third, the difference between swizzling in place and making copies does not have a strong effect on total work session time (we

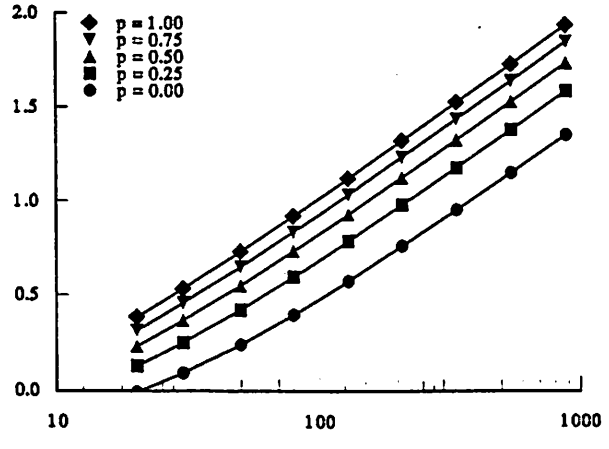


Figure 18: Locus of IS = IP (create): $\log_{10} v$ vs. b for selected values of p

assume no virtual memory paging occurs). Fourth, the object store delivers a significant fraction of the available disk bandwidth (on the order of 80%). Fifth, in our tests incremental loading of a collection performed a bit worse than preloading the collection, for which we have no definitive explanation. Sixth, and perhaps most importantly, the work session model of application behavior allows a simple performance model to be constructed, using parameters of the data set that are relatively easy to measure. Obtaining reasonable values for the work session parameters v and w may be more difficult.

There is considerable important work remaining on these topics. The work session model of applications needs qualitative evaluation to determine its applicability to important practical settings. The performance model would be strengthened by further validation. The models, experiments, and object management techniques could be extended to a client/server or other distributed model. Constants in the performance model could be determined for other hardware and software combinations. Studies could be done of actual object collections to determine interesting values for the independent variables of the performance model. The question of eager vs. lazy loading needs further investigation. A cost model allowing objects to be created and destroyed while accessing an existing collection would be more general. Finally, we made a number of assumptions and took some minor shortcuts in our programs. The effects of these on the tradeoffs between object management techniques should be explored in more detail.

10 Acknowledgments

Steven Sinofsky implemented the first running version of Mneme, which Tony Hosking has substantially improved, adding the direct access interface (among other things). Bojana Obrenić coded the original tree creation and traversal program and its test script driver. Farshad Nayeri assisted in the data analysis by preparing draft graphs. Tony Hosking provided the work session model figure. Finally, we are grateful to the Exodus group at the University of Wisconsin for their assistance with the Exodus storage manager, well beyond the usual professional courtesy.

References

[1] *J. E. B. Moss* and S. Sinofsky, "Managing persistent data with Mneme: Designing a reliable, shared object interface," in *Advances in Object-Oriented Database Systems*, vol. 334 of *Lecture Notes in Computer Science*, pp. 298–316, Springer-Verlag, Sept. 1988.

[2] J. E. B. Moss, "Addressing large distributed collections of persistent objects: The Mneme project's approach," in *Second International Workshop on Database Programming Languages*, (Gleneden Beach, OR), pp. 269–285, June 1989. Also available as University of Massachusetts, Department of Computer and Information Science Technical Report 89–68.

[3] J. E. B. Moss, "The Mneme persistent object store," COINS Technical Report 89-107, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, Oct. 1989. Submitted for publication.

[4] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "Object and file management in the EXODUS extensible database system," in *Proceedings of the 12th International Conference on Very Large Databases*, (Kyoto, Japan), pp. 91–100, ACM, Sept. 1986.

[5] M. J. Carey, D. J. DeWitt, D. Frank, G. Graefe, J. E. Richardson, E. J. Shekita, and M. Muralikrishna, "The architecture of the EXODUS extensible DBMS: A preliminary report," Computer Sciences Technical Report 644, University of Wisconsin, Madison, WI, May 1986.

[6] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "Storage management for objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications* (W. Kim and L. Frederick H, eds.), Frontier Series, ch. 14, pp. 341–369, ACM Press, New York, NY: Addison-Wesley, 1989.

[7] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[8] R. Fitzgerald and R. F. Rashid, "The integration of virtual memory management and interprocess communication in Accent," *ACM Trans. Computer Systems*, vol. 4, pp. 147–177, May 1986.

[9] D. R. Cheriton, "The V kernel: A software base for distributed systems," *IEEE Software*, vol. 1, pp. 19–42, Apr. 1984.

[10] N. C. Hutchinson and L. L. Peterson, "The x-kernel: An architecture for implementing network protocols," Tech. Rep. 89-1a, Department of Computer Science, University of Arizona, Aug. 1989.