Working with Persistent Objects: To Swizzle or Not to Swizzle

J. Eliot B. Moss

Abstract— **Pointer swizzling[1] is the conversion of database objects between an external form (object identifiers) and an internal form (direct memory pointers). Swizzling is used in some object-oriented databases, persistent object stores, and persistent and database programming language implementations to speed manipulation of memory resident data. Here we describe a simplifying model of application behavior, revealing those aspects where swizzling is most relevant in both benefits and costs. The model has a number of parameters, which we have measured for a particular instance of the Mneme persistent object store, varying the swizzling technique used. The results confirm most of the intuitive, qualitative trade-offs, with the quantitative data showing that some performance differences between schemes are smaller than might be expected. However, there are some interesting effects that run counter to naive intuition, most of which we explain using deeper analysis of the algorithms and data structures.**

Index Terms—**Object store, performance measurement, performance modeling, persistent object store, persistent programming, pointer swizzling, work session.**

I. Introduction

WE ARE concerned with programs that read and write *persistent objects* maintained in a database or object store. Any given persistent object can refer to other persistent objects within the same database. Such references are expressed via *unique object identifiers* (OID's), as in (for example) [1]. We are specifically concerned with systems in which OID's are *not* virtual memory addresses. The central question we explore is whether it is profitable to replace OID references between memory resident persistent objects with direct pointers. Such conversion is called *pointer swizzling*; similar techniques were used in LOOM [2], [3]. The basic trade-off in swizzling is obvious: the conversion costs something up front (and at the end of the program, to convert pointers back to OID's), but saves a little each time a reference is followed. While the qualitative situation is clear, we want to confirm it and describe the trade-off quantitatively. We also want to relate swizzling costs to other CPU costs in a program and to total costs including I/O time.

Manuscript received March 1, 1990; revised March 2, 1992. This work was supported by the National Science Foundation under Grant CCR-8658074, and by Digital Equipment Corporation and GTE Laboratories. Recommended by E.E. Swartzlander.

The author is with the Department of Computer Science, University of Massachusetts, Amherst, MA 01003.

IEEE Log Number 9201614.

[1]We have not been able to establish a definitive etymology of this curious expression.

1.1. Related Work

The cost of swizzling is relevant to persistent programming languages, database programming languages, object oriented database systems, persistent object stores, and object servers. A *persistent programming language* (PPL) is a programming language that includes a persistent memory area (e.g., a heap of objects) that outlives the execution of any individual program. PPL's have most often been designed as extensions of nonpersistent programming languages. PS-Algol [4] introduced the concept of a PPL. The E language [5]–[8] is a more recent PPL-based on C++; Alltalk [9], [10] is one based on Smalltalk [11].

A *database programming language* (DBPL) is similar to a PPL but adds database features such as bulk data (sets or relations) and attribute-based retrieval (queries). Pascal-R [12] is an early DBPL effort based on Pascal that has since moved on to Modula and is now called DBPL [13], [14]. Other DBPL's include O++ [15], [16], ADABTPL [17], FAD [18], Gemstone [19], and CO_2 [20].

E might also be called a DBPL since it has some support for bulk data management; the dividing line between PPL's and DBPL's is fuzzy. General issues of DBPL design for object-oriented languages are discussed in [21], and issues of types and persistence are discussed in [22], which includes a survey of database and persistent programming languages. Swizzling is directly relevant to the implementation of PPL's and DBPL's since such languages support general computation with persistent objects.

Swizzling might also benefit execution of compiled queries in *object-oriented databases* (OODB's), distinguished from traditional databases in that they manage objects having identity, etc. OODB's described in the literature include Exodus [23], DAMOKLES [24], CACTIS [25], VBase [26], Gemstone [19], [27], O_2 [28], [29], Orion [30], and Iris [31]. OODB's generally include a data manipulation language; such a language may be classified as a DBPL if it is rich enough in programming constructs.

Swizzling is also relevant to *persistent object stores* (POS's) and *object servers*, since applications using a store or server might benefit from converting objects from the store/server format to a faster in-memory format. POS's and storage managers include the Exodus storage manager [23], [32], O_2 [29], and Mneme [33]–[35].[2] Mneme is the POS used for this study. There have also been a number of designs related to virtual memory such as [2], [3], [36]–[38]. Object servers include ObServer [39] and Gemstone [40].

[2]Mneme is the Greek word for *memory*; we pronounce it NEE-mee.

We know of no prior studies of swizzling performance, and hence can offer no comparison with directly related work. Published OODB benchmarks and performance studies include [41]–[46].

1.2. Simplifying Assumptions

To study swizzling and obtain clear results, we focus on application behavior patterns most affected by swizzling, i.e., where swizzling is on the critical path. Consider a CAD design tool as an example. In an edit session the tool loads an existing design file (or creates a new one), edits in virtual memory, and then saves the file back to the database. This edit cycle will tend to be especially revealing of swizzling costs, since swizzling typically occurs while data is being loaded, before the application gains control, and likewise unswizzling occurs during the save operation, adding to its length. On the other hand, design file editing (and similar operations such as design rule checking) is computation intensive and reveal the benefits of swizzling too. As described later, we consider and compare *eager* schemes, where all objects needed by the application are loaded and swizzled in advance, and *lazy* schemes, where objects are loaded and swizzled on demand.

We adopted the load-work-save (LWS) cycle as our application model because it is simple yet reveals swizzling effects clearly. To make the impact of swizzling most clear, we make three assumptions.

- Loading and saving perform minimal disk seeking or query processing, so I/O and CPU costs are minimal and swizzling costs most visible.
- There is no significant paging or buffer replacement, for the same reason.
- Concurrency control and recovery impose no significant overhead.

These assumptions are reasonable for significant aspects of interesting applications; CAD design tools are a good example. First, CAD design files provide a natural grouping that can be clustered for fast reading and writing, avoiding query processing or significant disk seeking overhead. Second, CAD design files are usually loaded into virtual memory backed with enough real memory to prevent substantial paging. Third, concurrency control can occur at the level of whole design files or large subsets, adding minimally to the total cost of handling files of significant size. Recovery can also be done at the level of entire files. We grant that logging or checkpointing may be desirable for recovering work if a crash occurs during a long session. Logging will impose comparable costs whether or not swizzling is used. Checkpointing can be treated as a save in the LWS model, except that a load is not required after a checkpoint. While recovery may be of some relevance, we obtained simpler results without it, and our approach can be extended to handle recovery in the future.

We do not claim that these assumptions are realistic for all applications or for all parts of any application. Rather, the assumptions are those conditions that most reveal the costs and benefits of swizzling, and if the assumptions are violated then the costs/benefits are diluted by the additional system overheads. In that sense the assumptions give an upper bound on the costs/benefits of swizzling. Note also we do not claim to offer a general model of PPL, DBPL, OODB, or POS performance, only a model that reveals the maximum impact of swizzling.

1.3. Plan of the Paper

Our study of swizzling proceeds as follows. We define the LWS application model and an analytic cost model suitable for comparing alternatives over a wide range of application characteristics such as the number of objects loaded. We discuss the swizzling alternatives to be compared, describe simple benchmark programs, and present measurements that fix the parameters of the cost model for each swizzling alternative. Finally, we analyze and compare the resulting models.

II. The Application Model

Our LWS model of application behavior uses three concepts: *objects*, *collections* of objects, and *sessions* of work upon collections.

For our purposes an object is a contiguous aggregation of some *slots* and some *bytes*. Each slot may contain a reference to another object, a null reference (to no object), or possibly a nonreference value, if the language (data model) allows tagged quantities. The bytes represent nonreference values such as integers, floating point numbers, and strings. For simplicity we assume the number of slots and bytes in an object does not change after the object is created. Objects model the individual records, arrays, strings, etc., of the application, and hence tend to be "small" (average of 40 to 50 bytes in a Smalltalk system for example). We will be concerned with how slots and bytes add bulk to an object, the swizzling costs of references in slots, and the general costs of manipulating slots and bytes within objects. Note that we model only *structural* properties of objects and do not include any sort of type or class system, inheritance hierarchy, or operation/method invocation mechanism. Our model is adequate because we are not concerned with object semantics but only with implementation costs.

A collection is simply the set of objects used during the execution of an application. As previously mentioned, we assume collections are clustered so that I/O costs are minimal and swizzling costs most apparent. However, 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 experiments we assume the collection is known beforehand and consider the performance effects of eager versus lazy loading. We will characterize collections in terms of the number of objects they contain and average properties of those objects such as size, number of slots, etc.

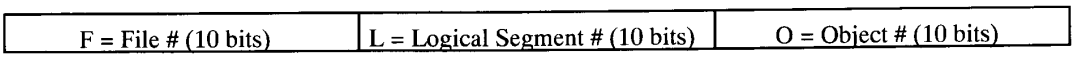

Fig. 1. Mneme object id format.

A session is a create-work-save or LWS cycle. For simplicity we assume sessions either create a collection, or read and possibly modify an existing collection without creating new objects. It would not be difficult to extend the model to allow incremental addition and removal of objects in collections, but would complicate the presentation and analysis. An LWS session breaks down into the obvious three basic activities of *loading*, *working*, and *saving*. Loading and working are separate when all objects are loaded in advance (eager loading), and overlapped in the case of lazy loading. Loading is further broken down into *reading* and *swizzling*. Reading is the fetching of objects from persistent storage into buffers, and swizzling is the conversion (if any) of the objects for the application's use. Saving is similarly separated into *preparing*, unswizzling objects and getting them into storage manager buffers as necessary, and *writing*, the actual transfer of the buffers to persistent storage. In a create-work-save session *creating* replaces loading, and is different in that creating is an in-memory operation, while loading involves persistent storage access and swizzling.

III. Techniques Of Object Management

We now describe the various swizzling and object management alternatives relevant to our investigation. First we describe the basic approaches, then implementation variations on those approaches, and finally algorithmic details of the techniques used in our performance studies. Relevant aspects of Mneme are discussed as necessary.

3.1. Basic Approaches

There are three fundamental approaches to swizzling. First, we can simply not swizzle. Second, we can convert objects between in-memory and on-disk formats *in-place*, in the object manager's buffers. Third, we can swizzle by making a separate in-memory *copy* of the object being swizzled, leaving the buffer resident copy undisturbed. We call these schemes nonswizzling (NS), in-place swizzling (IS), and copy swizzling (CS).

While the trade-off between swizzling and not swizzling is fairly obvious, the relative merits of IS and CS are more subtle. IS avoids making extra copies, and thus has lower CPU and memory cost than CS. On the other hand, IS requires that all objects be unswizzled before the object manager's buffers are written back to persistent storage, whereas CS requires unswizzling only of new and modified objects.

If we swizzle, we may do *eager* swizzling or *lazy* swizzling. In eager swizzling we swizzle the entire collection of objects in advance. This requires being able to identify the collection before using it, or at least bounding it. Eager swizzling avoids the overhead of dynamic checks for unswizzled objects. Lazy swizzling, on the other hand, inserts dynamic checks so that the collection can be "discovered" during execution rather than being identified or bounded in advance. In fact, there is a spectrum between pure eager swizzling and pure lazy swizzling. For example, objects might be aggregated into subgroups, with entire subgroups swizzled at a time, and subgroups loaded and swizzled on demand rather than in advance. We consider only the two extremes since intermediate approaches will be intermediate in cost. In sum we have five swizzling approaches: nonswizzling, eager and lazy in-place swizzling, and eager and lazy copy swizzling.

3.2. Variations

If we do not swizzle, then every time we wish to manipulate an object we must present the oid for the object to the object manager (OM), which must then locate the object (if it is resident) or retrieve it (if it is not resident) and perform the manipulation for us. Instead of calling the OM for each manipulation (a *call* interface), we might obtain a pointer to the object in the OM's buffers and then manipulate the object directly (a *pointer* interface). We consider only the more efficient pointer interface here. Note that while it is possible and reasonable for an application using a pointer interface to cache some pointers, e.g., for the duration of a procedure call manipulating an object, such pointers cannot be stored into the objects themselves—that would be swizzling. Note also that retaining direct pointers into buffers requires pinning objects, and hence gets into details of the specific OM's features. If there is a series (one or more) of operations on an object where we can cache a buffer pointer to the object across the entire series, we call that a *visit* of the object. Each visit requires one OID lookup with a pointer interface, but possibly many lookups with a call interface (one for each call).

For eager swizzling, and for unswizzling, there are two fundamental control strategies possible: *iteration* through a set of objects identified in some other way (e.g., "all objects in file X"), and *recursion* through the objects and their references (also called *reachability*). Recursion is more general since it can be expressed in terms of object manipulations; iteration may have performance advantages. We used recursion, for three reasons: it is more general; it likely costs at least as much as iteration, so it gives a better upper bound on swizzling cost; and iteration was not supported by the OM.

Lazy swizzling raises the additional problem of detecting uses of not yet swizzled objects. This problem has many solutions, ranging from hardware support similar to (or built on) virtual memory address translation to software-only approaches. We investigated a software approach because it is most general and gives an upper bound on swizzling cost.

3.3. Algorithms Used

The only algorithm of interest for the nonswizzling scheme is object lookup. Mneme's object lookup technique is similar to virtual memory address translation. The version of Mneme used here splits a 30-bit OID into three 10-bit fields, as shown in Fig. 1. The high-order field indexes a file table to obtain

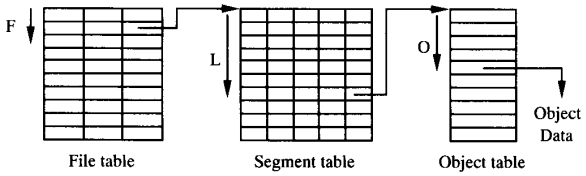

Fig. 2. Object lookup scheme used.

the base of a segment table for that file. The middle field indexes that segment table, giving information about whether the segment is resident, and its memory location if it is. If the segment is not resident, other information is used to locate the segment on disk and retrieve it. In any case, the least significant 10 bits of the OID then index an object table at the beginning of the segment to obtain an offset to the start of the object. This is shown in Fig. 2. To allow more objects to be retrieved at once, several of the segments just described can be grouped together into a single physical unit, always stored contiguously. In Mneme parlance, the physical unit is a *physical segment*, and what we called segments above are termed *logical segments*. Physical segment layout is illustrated in Fig. 3. For more information on Mneme see [35].

Turning to swizzling, Fig. 4(a) sketches the code for eager in-place swizzling. The *Swizzled* and *MarkSwizzled* actions test and set a field containing two bits, encoded as shown in Table I. The encoding is chosen so that the swizzled/unswizzled test examines one bit and the new/modified versus unmodified test looks at the other. These two bits are stored in a 32-bit word at the start of the object. The other 30 bits contain the OID if the object is swizzled (for later use in unswizzling) and are zero otherwise.

Eager copy swizzling differs in important ways from eager in-place swizzling. First, copy swizzling requires some means for locating the in-memory copy of a swizzled object given its OID, similar to the OM lookup routine. Second, CS creates and fills in a (swizzled) copy of the object. Fig. 4(b) sketches the code for eager CS.

Lazy swizzling is more complicated. If object X refers to object Y and both are resident, we want the reference to be a direct pointer. But what if X is resident and Y is not? Can we swizzle X under those circumstances, and if so, what does that mean? One approach is to tag slot contents as to whether they are swizzled: if tagged as *swizzled*, then the slot contains the actual address of the target object, if tagged as *unswizzled*, the OID. We call this *edge marking*. There is a major disadvantage to edge marking. Slot contents can be fetched, passed around, and stored without accessing the target object. Hence, by the time the target is accessed, we may have no idea where the object reference came from, and it is costly to scan through all resident objects to find the sources and swizzle them. One possibility is to use object lookups as necessary and do occasional swizzling scans to amortize the scanning overhead.

Another approach to handling references to nonresident objects is to require that all object references in resident objects be converted to pointers, with small pseudo-objects (we call them *fault blocks*) standing in for nonresident objects, as shown in Fig. 5(a). A fault block contains the OID of the target object, and is distinguishable from an ordinary object. We call this approach *node marking*. When a reference is to be followed, if it refers to a fault block, we locate the target object (retrieving it if necessary) and change the fault block to point to the now-resident object (see Fig. 5 (b)). We call the updated fault block an *indirect block*. If a reference to be followed refers to an indirect block, we thus locate the target object at the cost of an indirection. Similar to edge marking, occasional scanning (perhaps by a garbage collector) can be used to bypass indirect blocks, as shown in Fig. 5(c).

A complete study of lazy swizzling would become involved in details of PPL/DBPL implementation, including available compiler optimizations, garbage collection techniques, etc.—well beyond the scope of this study. We measured the cost of dynamic checking and swizzling, omitting scanning. We carried this out using the following three rules. First, every reference in a swizzled object is a direct pointer (i.e., node marking), to either a swizzled object or a resident, unswizzled object. Second, every reference in an unswizzled object is an OID, which may refer to any object, resident or not. Third, using an object requires that it be swizzled. If an object is unswizzled at the start of a visit, it is immediately swizzled and any nonresident objects it references are made resident (but not swizzled). This approach thus incorporates the effects of dynamic swizzling checks, incremental swizzling, and incremental loading of segments from persistent storage, but not scanning, and it performs some actions in a different order than would a real system. Thus we offer only approximate measures of the real cost of lazy swizzling.

IV. The Cost Model

We desired a simple model of the cost of a session that would consider characteristics of both the object collection and the work performed. Our measure of cost is total elapsed time for a session; this is both simple and arguably the most important performance measure. We desired a small number of variables to describe collections and sessions adequately, and settled on four collection variables and three session variables. These variables are adequate, at least for the simple programs we ran; they also have the virtue that they might be measured in practice for actual programs.

The variables are presented in Table II. Collection variables n and b are obviously relevant, and p is clearly related to swizzling. Variable i is necessary for modeling null/nonnull references. Note that b specifically excludes object headers and other overhead, but includes slots and all other user data. Session variable u is important in considering how much work is done in preparing and writing. Variable v is a measure of the number of times references are traversed (looked up), and thus is important in determining the benefits of swizzling. Finally, w gives us something against which to compare swizzling costs.

Session cost is a function of the variables, determined by a number of *parameters*, where the parameter values depend on the swizzling approach selected and its actual implementation. We measured the parameters for several approaches, and

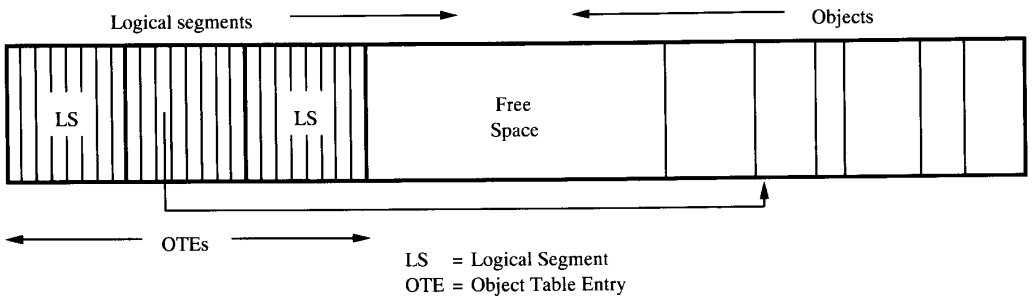

Fig. 3. Physical segment data structure.

```
EagerInPlace (id) returns (pointer)
  p: pointer := OMLookup (id)
  if not Swizzled (p) then
    MarkSwizzled (p, id)
    for x := each slot offset in p do
      if p[x] holds an oid then
        p[x] := EagerInPlace (p[x])
  return p
```

(a)

```
EagerCopy (id) returns (pointer)
  p: pointer := CopyTableLookup (id)
  if p is nil then
    q: pointer := OMLookup (id)
    p := AllocateCopy (SizeOf (q))
    CopyTableEnter (id, p)
    Copy contents of q to contents of p
    MarkSwizzled (p, id)
    for x := each slot offset in p do
      if p[x] holds an oid then
        p[x] := EagerCopy (p[x])
  return p
```

(b)

Fig. 4. Code sketches for eager swizzling. (a) Eager in-place swizzling. (b) Eager copy swizzling.

present and discuss them in some detail later. We use upper case letters for the parameters to distinguish them from the variables. We consider the create, load, work, and save phases separately, devising an equation for each. The equations are weighted sums of products of the variables (the variables were chosen this way). It turns out that the constant terms of the equations are negligible, so for convenience we omit them from the start. Note that *pb* is the average number of bytes devoted to slots in an object and *pbi* the average number devoted to initialized slots.

Creation costs include overhead for each object, CO·*n*, and for initialization: CB·*nb* (per byte costs), CS·*np* (per slot costs), and CI·*ni* (per initialized slot costs). Loading costs are time to read the user data LB·*nb*, overhead for handling each object LO·*n* (including reading headers and other overhead), overhead for handling each slot LS·*np*, and overhead in processing each reference (e.g., swizzling) LI·*ni*. Saving is similar to loading but adds terms depending on the fraction of objects updated, *u* (this is always 1 for a create-work-save cycle). The cost of work is rather different, consisting of a visit overhead V·*nv* (i.e., for following an object reference), and an actual work term *nvw* (*w* is defined so that this term is not parameterized). These are the resulting equations:

TABLE I
ENCODING OF THE "SWIZZLED" FIELD OF OBJECTS

00	new transient (nonpersistent) object
01	unswizzled persistent object
10	swizzled modified persistent object
11	swizzled unmodified persistent object

$$C = CB \cdot nb + CO \cdot n + CS \cdot np + CI \cdot ni$$
$$L = LB \cdot nb + LO \cdot n + LS \cdot np + LI \cdot ni$$
$$W = V \cdot nv + nvw$$
$$S = SB \cdot nb + SO \cdot n + SS \cdot np + SI \cdot ni + UB \cdot nbu + UO \cdot nu + US \cdot npu + UI \cdot niu$$
$$CWS = C + W + S$$
$$LWS = L + W + S.$$

We can combine terms in the overall model to reduce the final number of parameters. Letting M (for *move*) replace L and S, and P (for *produce*) replace C, S, and U (because $u = 1$), we obtain:

$$CWS = PB \cdot nb + PO \cdot n + PS \cdot np + PI \cdot ni + V \cdot nv + nvw$$
$$LWS = MB \cdot nb + MO \cdot n + MS \cdot np + MI \cdot ni + V \cdot nv + nvw + UB \cdot nbu + UO \cdot nu + US \cdot npu + UI \cdot niu.$$

We summarize the model parameters in Table III for future reference.

The linearities assumed in this cost model are reasonable, and they are borne out by the experiments we ran. We note that the UB term is an oversimplification for situations more general than those we actually measured. UB covers the costs of writing changed data back to the database, and we assumed the number of *bytes* written back is proportional to the number of *objects* created or updated. This assumption fails if the affected objects are few but are scattered across many segments. The equations given are reasonable if logging is used, if *u* is close to 0 or to 1, or if modified objects are clustered together. This limitation of the cost model can be overcome by adding a new variable capturing the relative clustering of updates; for simplicity we stick with the model presented above.

The model above has 13 parameters to be measured and expresses cost in terms of seven variables. As will be seen, however, some terms drop out for some swizzling approaches, and some turn out to be negligible in practice. More speculatively, some of the collection description variables may tend to fall in narrow ranges, allowing further simplification.

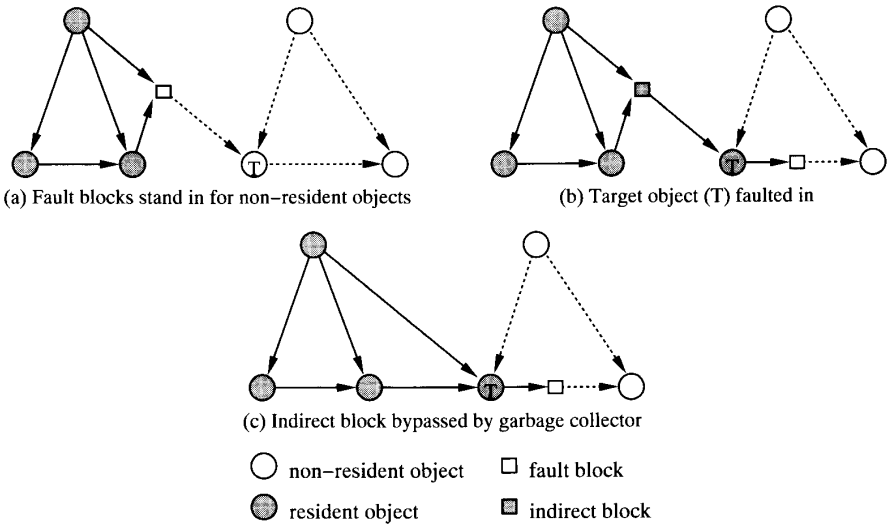

Fig. 5. Node marking approach to lazy swizzling.

TABLE II
COLLECTION AND SESSION DESCRIPTIVE VARIABLES

Collection Variables	
n	number of objects in the collection
p	average number of slots per object ("pointers")
i	average number of slots "initialized" (containing references)
b	average number of user bytes per object, including slots
Session Variables	
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 III
SUMMARY OF THE COST MODEL PARAMETERS

Two letter parameters: xy			
Choices for x		Choices for y	
M (move)	time per unit to load, prepare, and save	B	unit is bytes
U (update)	time per unit to prepare and save modified objects	O	unit is objects
P (produce)	time per unit to create, prepare, and save	S	unit is slots
		I	unit is initialized slots
One letter parameters			
V	time per object visit		

V. EXPERIMENTAL METHODOLOGY AND RESULTS

To make performance measurements we wrote a simple program to create, traverse, and modify collections of objects. It allowed us to control the independent variables of the cost model, such as number and size of objects. We ran the program in the context of a particular hardware/software configuration—a common workstation using the Mneme persistent object store. We now describe the nature of the collections and sessions supported by the test program, the hardware/software environment of the tests, what tests were performed, and the results of those tests.

5.1. Collections

To form collections, we needed a data structure supporting flexible adjustment of the collection variables over wide ranges. We settled on binary trees augmented with extra edges. The parameters of the trees are their height, properties of the internal nodes, and properties of the leaf nodes. Internal nodes have at least two initialized pointers for supporting the tree structure, but may have more pointers, any number of which may be initialized; such initialized pointers are self-loops, for simplicity. Internal nodes may also have any number of bytes in addition to the pointers, to add bulk to the objects. Tree height determines collection variable n, the number of pointers determines p, the number initialized determines i, and the number of bytes determines b (given p). Trees were chosen for simple construction and work traversal (swizzling considers all pointers); binary trees were chosen because they give the slowest growth of n with tree height and the lowest lower bound on p.

Our data structure is general enough because the order of traversal and the detailed pattern of pointers between objects are not important for the comparisons we make, so long as the same order and pattern is used in every case. The fact that additional initialized pointers (beyond those needed for the tree backbone) are self-loops might affect cache residency during swizzling, but locality of data structure argues against our simplification as having a strong effect.

As previously discussed, 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.

TABLE IV

Series	Height	Slots	Bytes	Notes
VS (Vary Slots)	11–17	F(2, 233)	0	total size between 2 and 4 megabytes
VB (Vary Bytes)	11–17	2	F(4, 932)	total size between 2 and 4 megabytes
VH (Vary Height)	14–19	2	0	
VN (Vary n)	12–19		0	slots set for total size of 8 megabytes
SI (Slots Init)	12	2 / 233	0	slots initialized = F(2, 233)
BI (Bytes Init)	12	2 / 0	0 / 932	bytes initialized = F(4, 932)
II (Ints Init)	12	2 / 0	0 / 932	integers initialized = F (1, 233)
WS (Work Slots)	15	2 / 34	0	
WB (Work Bytes)	15	2 / 0	0 / 84	

We also assumed that relatively small objects were the most interesting, extrapolating from the average object size in heap-based programming languages such as Smalltalk, which is on the order of 50 bytes, including overhead, 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.

We designed this new benchmark so that we could control the collection variables and disentangle the results to estimate the model parameters for swizzling. Existing benchmarks such as the Sun Engineering Database Benchmark [41], its more recent simplification [42], and the HyperModel benchmark [45] are designed to test overall object-oriented DBMS (or PPL or DBPL) performance, and do not vary some of the variables significant for our model. In short, we needed tests specialized to our task.

We used a number of series of collections in our measurements, described below so that others can reproduce our test cases. Several Fibonacci series are used in the description: F(1, 233) = {1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233}, F(2, 233) = F(1, 233) - {1}, and F(4, 932) = {4, 8, 12, 20, 32, 52, 84, 136, 220, 356, 576, 932}. Note that for height h there are $2^h - 1$ objects in the tree. If a table entry is written x / y, then x is for internal nodes and y for leaves. Finally, note that object size b is 4 · slots + bytes, rounded up to a multiple of 4 bytes. The header size was 8 bytes for nonswizzling and copy swizzling, and 12 bytes for in-place swizzling, to hold the object id of the object for later unswizzling. These values are for Mneme; other object stores would give different values. Headers are not included in b, though; their processing will show up as per-object rather than per-byte costs.

5.2. Sessions

For each swizzling approach considered, we measured the cost of creating collections, the incremental cost of initializing object contents, the cost of traversing the tree and doing some work at each node, the cost of loading, and the cost of saving. Saving was further broken down into preparing and writing, and we considered saving new collections, unmodified old collections, and modified old collections, since the various schemes have different behavior across these cases. Together these various measurements cover all aspects of work sessions.

To model work on individual objects, we devised activities to capture reading, updating, and initializing of slots, bytes (individually), and integers (aligned groups of 4 bytes treated as integers). We needed activities that were hard to optimize away or turn into block moves or other block operations, since real work usually does not admit such optimizations; Table V shows what we devised.

TABLE V

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	self loops
	Bytes	sequential values from global counter
	Integers	sequential values from global counter

Many of the operations include loop and recursion overheads, but these will be almost the same for each scheme, so differences in measurements will still reveal differences in costs of the various schemes. Further, the control structure of the test program (tree traversal) is about the simplest possible, so it will tend to expose swizzling overheads as desired.

5.3. Hardware and Software Used

All tests were performed on a DECStation[3] 3100 (Mips R2000A CPU[4] clocked at 16.67Mhz) running Ultrix 3.1. The system had 24 megabytes of main memory, 10% of which was used for operating system disk buffers. The disk used for the tests was a relatively empty RZ56 drive (665 megabytes, SCSI, 24.3ms average access time, 16-ms average seek time, 64-Kbyte data buffer); 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

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

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

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. Note, though, that write/save tests completed as soon as all bytes were passed to the operating system, so a significant quantity of data could be in operating system disk buffers. The Mneme segment size was set to 32 Kbytes in all cases.

5.4. Experiments

For each scheme 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. For some activities we measured the cost of nonpersistent C code (NPC), as a general comparison. For nonswizzling measurements, we used the Mneme routines that give direct access to objects in Mneme's buffers. We now describe the individual experiments and their results. We assemble these results into complete models in Section 5.5.

5.4.1. Creation We measured nonswizzling, swizzling, and nonpersistent C. We report in-place and copy swizzling as a single number because both schemes act the same in creation: the objects are always created in the application's heap with Mneme objects created only during the prepare phase. Swizzling is very similar to nonpersistent C, except that nonpersistent C may omit some object header information and need not initialize all slots to nongarbage values. Hence, nonpersistent C is a little faster, but swizzling is still quite fast in this phase. Note also that the eager/lazy distinction makes no sense in creation.

Creation was tested on collection series VS, VB, SI, BI, and II. The VS and VB tests showed that per-object creation costs do not vary with the size of the objects, except for VS in the swizzling scheme since it always initializes all slots explicitly. Non-persistent C avoids this cost entirely, and nonswizzling starts with zero-filled memory, whose initialization cost is very low. The SI, BI, and II collection series allowed us to measure the marginal cost of initializing object contents. Thus we end up with a per-object allocation cost, and per-slot, per-byte, and per-integer initialization costs, all obtained from simple regression fits of elapsed time versus number of objects, or elapsed time per object versus number of slots, bytes, or integers initialized.[5] Each of SI, BI, and II give an estimate of the per-object cost. These estimates are close to each other; we present the BI estimates, an arbitrary choice.

Table VI shows the costs; columns labeled ± give 95% confidence intervals throughout the paper. Note that the marginal initialization costs are almost exactly the same for the various schemes; the only significant variation is in the per-object cost. Recall, though, that this cost includes the loop/recursion overhead of the test program. Taking the difference between the costs of Table VI and the cost of no-work traversal (presented later) gives a better measure of the cost of allocation by itself, though since the loops are different, the comparison is only approximate.

[5]The swizzling per-slot cost was obtained from the VS series since the SI series does not vary the total number of slots, and swizzling always initializes all slots.

TABLE VI
OBJECT CREATION AND INITIALIZATION COSTS

Scheme	Per Object	Per Slot	Per Byte	Per Integer
	μs (±)	μs (±)	μs (±)	μs (±)
NS	33.7 (.3)	0.394 (.009)	0.374 (.002)	0.401 (.007)
IS/CS	10.5 (.4)	0.393 (.005)	0.368 (.002)	0.399 (.008)
NPC	9.1 (.3)	0.387 (.007)	0.367 (.002)	0.385 (.005)

TABLE VII
TRAVERSAL WITH NO WORK

Scheme	Per Object	Ratio to NPC
	μs (±)	
NS	20.15 (.02)	1.70
NS (adjusted)	17.75 (.02)	1.50
IS/CS (with check)	13.14 (.03)	1.11
IS/CS (no check)	12.04 (.02)	1.02
NPC	11.83 (.03)	1.00

5.4.2. Work Again we measured nonswizzling, swizzling, and nonpersistent C. We expected nonpersistent C and swizzling to have essentially identical cost, and nonswizzling to cost more. As a baseline, we measured no-work traversal on VS and VB. This was done with and without object residency checks, to compare between eager and lazy swizzling when objects are actually resident. We also varied, individually, the reading and writing of slots (using WS), and of bytes and integers (using WB), to get a sense of the incremental cost of manipulating fields of objects, though those times are not part of our cost model, except indirectly through the w factor.

The no-work results were obtained using collection sets VS and VB, which gave nearly identical results. We performed linear regression of CPU time versus number of objects visited to obtain per-object costs. The numbers for VS are presented in Table VII. We see that the cost of a residency check for a resident object is $1.10 \pm .05\mu$s, less than 10% of the cost for no-work traversal. Also, nonpersistent C is apparently 2% faster than swizzling without residency checks. We attribute this small difference to slight variations in the code generated by the C compiler.

Mneme object id lookup costs $8.32 \pm .05\mu$s more than a C pointer dereference (NS − NPC), which includes the cost of a second call to Mneme to release the object. The release call takes approximately 2.4μs, so lookup takes 5.9μs by itself. The release call may or may not be needed depending on the approach taken for buffer management, concurrency control, and noting of updates. Henceforth we will assume the release call is *not* needed. Therefore, we will use the adjusted

TABLE VIII
TRAVERSAL WITH WORK

Scheme	Slot Read	Byte Read	Int. Read	Slot Update	Byte Update	Int. Update
	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)
NS	0.55 (.02)	0.32 (.00)	0.35 (.01)	0.45 (.02)	1.03 (.00)	0.44 (.02)
IS/CS	0.55 (.02)	0.32 (.00)	0.35 (.01)	0.46 (.02)	1.03 (.00)	0.44 (.02)
NPC	0.55 (.02)	0.32 (.00)	0.35 (.01)	0.45 (.02)	1.03 (.00)	0.44 (.02)

TABLE IX
EAGER LOADING AND SWIZZLING

Scheme	Per Byte	Per Object	r^2	F
	μs (±)	μs (±)		
CS (measured)	2.17 (.09)	29.8 (1.0)	0.9903	3357
IS (measured)	2.04 (.05)	29.4 (0.8)	0.9962	8816
CS (predicted)	2.17 (.09)	25.5 (1.0)		
IS (8 byte headers)	2.04 (.05)	21.2 (1.0)		

NS values, and will adjust any other results as necessary to remove the cost of calls to the release routine. Recent revisions to Mneme have reduced the lookup costs; we discuss the implications later.

The results for traversal with work are shown in Table VIII. These were calculated by linear regression of time per object versus number of slots, bytes, or integers read or written. There is virtually no difference between the schemes, which is only to be expected since they all manipulate object contents directly via pointers. These results confirm that performance has not been sacrificed in some surprising way.

5.4.3. Loading First we consider eager loading. This applies only to the swizzling schemes since nonswizzling is inherently lazy. We used collection sets VS and VB, and combined their data. We analyzed the data via multiple regression of elapsed time versus number of user bytes and number of objects loaded. The per byte and per object costs are given in Table IX, along with the correlation coefficients (r^2) and the F statistic.[6] Copy swizzling costs more per byte and per object, as might be expected. However, in-place swizzling costs almost as much per object, partly because its headers are 4 bytes larger, increasing costs by about 8μs per object. The table does not include the cost of looking up each *initialized* slot (i.e., the slots beyond those used for the tree structure). We saw in Section 5.4.2 that that cost is 5.7μs/slot, independent of swizzling scheme.

The benchmark program used fairly simplistic processing for building and searching the resident object table required for copy swizzling. On the one hand, a realistic table organization would cost an estimated 4.5μs/object more for each id lookup and 5.0μs more for entering each object in the table. On the other hand, a more clever approach would handle whole groups of objects at a time and obtain considerable savings, an estimated *reduction* of 4.3μs per object of CPU time from the current values. It could be that not all of this would show up in elapsed time reductions, since CPU costs may currently be overlapped with I/O. Still we use 4.3μs as our best estimate, as shown on the "predicted" line of the table. We do not believe that more clever approaches would help in-place swizzling comparably. This means that because of its larger headers, in-place swizzling actually costs *more* than copy swizzling for objects with up to 30 bytes of user data. If there were some way to avoid the extra 4 bytes required by in-place swizzling then in-place would always be faster than copy swizzling, as shown by the last line of the table. Though there is an element of estimation involved, we will carry the copy swizzling adjustment through to the cost model so as to give comparisons between how the schemes would be implemented in a complete system rather than how they were coded in the benchmark program.

[6]High values of F indicate that most of the variance is accounted for by the linear regression, since F is (roughly speaking) the ratio of the variance explained by the regression to the remaining variance.

TABLE X
LAZY LOADING AND SWIZZLING, WITH COMPARISON

Scheme	Per Byte	Per Object	r^2	F
	μs (±)	μs (±)		
LCS (measured)	2.56 (.10)	49.4 (1.1)	0.9951	6962
LIS (measured)	2.37 (.09)	44.3 (1.0)	0.9953	7272
NS (measured)	1.77 (.02)	42.8 (0.4)	0.9993	26 832
LCS (predicted)	2.56 (.10)	42.7 (1.1)		
LIS (repeated)	2.37 (.09)	44.3 (1.0)		
ECS + traversal	2.17 (.09)	37.5 (1.0)		
EIS + traversal	2.04 (.05)	41.4 (0.8)		
NS (adjusted)	1.77 (.02)	40.4 (0.4)		

In another test, using data set VN, which keeps the data set size fixed but varies the number of objects, we found that eager swizzling cost 28.3±0.8μs per object. This matches reasonably well with the results in Table IX. The nonswizzling read time was 3.57±0.01μs per byte, giving a rate of about 280 000 bytes per second.

Lazy loading with swizzling, measured as for eager loading, revealed higher per byte and per object costs, though we must keep in mind that lazy swizzling includes traversal time, whereas eager swizzling does not. The multiple regression results for lazy swizzling are shown in Table X. The nonswizzling results are also shown there, which were computed by performing a multiple regression on results from data sets VH and VN. The predicted lazy copy swizzling cost deducts the 4.3μs previously discussed, and an additional 2.4μs for a call used in the benchmark that can be removed by a more clever implementation. Lazy in-place swizzling outperforms lazy copy swizzling for objects having more than 8 bytes of user data. In the table we also include the eager costs (ECS and EIS) with one no-work traversal (without residency checks) added in, and the nonswizzling results, for comparison.

TABLE XI
LOADING COST COMPARISON FOR VARIOUS SIZE OBJECTS

Scheme	Time(μs/object)				Ratio to NS (%)				
	Number of user bytes				Number of user bytes				
	8	24	50	200	8	24	50	200	∞
LCS	63	104	171	555	115	125	133	141	145
LIS	63	101	163	518	115	122	126	131	134
ECS	55	90	146	472	100	108	113	120	123
EIS	58	90	143	449	105	108	111	114	115
NS	55	83	129	394					

To summarize the loading results, copy swizzling, whether lazy or eager, is slightly faster for small objects, but in-place swizzling's lower cost per byte eventually wins out. This inversion results from the extra 4 bytes per object required to implement in-place swizzling. If an alternative implementation for unswizzling could be devised then in-place swizzling would always be faster than copy swizzling. Laziness always costs more than eagerness. Not swizzling is the cheapest approach for objects with at least 8 bytes of user data. Eager copy swizzling is predicted to be a little faster for extremely small objects because swizzling whole segments of objects at a time in tight loops will save a little compared with nonswizzling's uniform use of general id lookup.

While the results give the intuitive orderings (lazy > eager, copy > in-place, and swizzling > nonswizzling) for medium sized and large objects, it is a bit surprising that the costs for small objects are so similar and that there are some inversions of the expected orderings. In Table XI we show the expected times for each scheme for several object sizes, and their ratios to the nonswizzled times, to give a better sense of the relative performance of loading in each scheme.

5.4.4. Writing For measuring writing, we used collection series VH and VN. The idea was to determine clearly how collection size and number of objects each affect writing cost. Write measurements were done for the nonswizzling scheme; since we did not overlap preparing and writing, the write costs for swizzling schemes are the same as for nonswizzling. We found that the number of objects had very little effect and that writing takes 3.38±0.09μs/byte; this is about 296 000 bytes/s. These numbers include header bytes, so per object costs must be calculated based on the number of header bytes in each scheme.

5.4.5. Preparing We measured preparing for in-place and copy swizzling using the same collections as for writing. There are three subcases: new collections, unmodified old collections, and modified old collections, as previously discussed. In all cases the number of objects and the number of slots were seen to be significant cost factors. The size of the objects was significant only for new collections (because new objects must always be copied into store objects) and for copy swizzled objects that are modified (because they require copying back).

The results, presented in Table XII, show several things immediately. First, all byte copying has essentially the same cost, 0.20μs per byte. Second, per slot costs are very close, about 0.40μs when copying and 0.77μs when not. (This is not anomalous. Recall that slot costs are over and above their per byte cost, so the total cost per slot in the copying cases is about 1.2μs.) Third, in the new and old objects cases, copy and in-place swizzling perform similarly, with copy swizzling costing slightly more per object. The only major difference between the swizzling cases is the handling of modified objects, where copy swizzling definitely costs more. The copy swizzling per object costs for the new and modified cases have been adjusted to remove 2.4μs for an unnecessary call.

TABLE XII
PREPARING

Scheme	Per Object	Per Slot	Per Byte	r^2	F
	μs (±)	μs (±)	μs (±)		
CS New	29.6 (0.08)	0.40 (0.01)	0.20 (0.01)	0.9999	362679
IS New	29.1 (0.13)	0.43 (0.02)	0.19 (0.01)	0.9999	181722
CS Old	4.6 (0.02)	0.77 (0.01)		0.9998	200728
IS Old	4.4 (0.03)	0.77 (0.01)		0.9996	86 504
CS Modified	11.9 (0.07)	0.40 (0.01)	0.20 (0.01)	0.9997	87 147
IS Modified	6.6 (0.04)	0.77 (0.01)		0.9996	94 829

TABLE XIII
PREPARE COST PER INITIALIZED SLOT

Scheme	New	Old	Modified
	μs (±)	μs (±)	μs (±)
CS	1.81 (0.01)	1.80 (0.01)	1.81 (0.01)
IS	1.89 (0.01)	1.90 (0.01)	1.89 (0.01)

The reason there are *any* prepare costs for old objects in the copy swizzling case is that we used traversal of the object structure to find modified objects. If we devised some means other than traversal to find the modified objects, then we could avoid most of the prepare costs for old objects when copy swizzling.

To measure per-pointer prepare costs, we used data set SI, which varies the number of initialized slots. The results are shown in Table XIII. We see that copy swizzling consistently costs about 1.8μs per slot, and in-place costs 1.9μs.

5.5. Cost parameter values

The preceding tables allow us to calculate the parameters of the cost model for each scheme. Here we present the resulting cost models for both load-work-save and create-work-save sessions; Section VI analyzes the models, compares the schemes, and derives results.

5.51. Load-work-save sessions See Table XIV for the cost model parameter values for nonswizzling, and lazy and eager copy and in-place swizzling. We calculated the parameters as follows.

[MB] This comes straight from the per byte costs in Tables IX and X.

[MO] The two primary components of MO are the per object loading costs (Tables IX and X) and preparing costs (Table XII). The lazy schemes (NS, LCS, LIS) are adjusted by subtracting the cost of one no-work traversal, since they

TABLE XIV
LOAD-WORK-SAVE PARAMETER VALUES

Scheme	Parameter							
	MB	MO	MS	MI	V	UB	UO	US
	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)
LCS	2.56 (.10)	26.5 (1.0)	0.77 (.01)	7.7 (.06)	13.14 (.03)	3.58 (.10)	34.3 (0.8)	−.37 (.01)
LIS	2.37 (.09)	27.8 (0.9)	0.77 (.01)	7.8 (.06)	13.14 (.03)	3.38 (.09)	42.8 (1.5)	
ECS	2.17 (.09)	22.4 (0.9)	0.77 (.01)	7.7 (.06)	12.04 (.02)	3.58 (.10)	34.3 (0.8)	−.37 (.01)
EIS	2.04 (.05)	26.0 (0.8)	0.77 (.01)	7.8 (.06)	12.04 (.02)	3.38 (.09)	42.8 (1.5)	
NS	1.77 (.02)	25.1 (0.4)			17.75 (.02)	3.38 (.09)	27.0 (0.7)	

include a traversal. We made one more adjustment. There must be an initialized pointer referring to each object visited, which causes the first visit to the object. Our experiments for measuring MO included the cost of processing that initialized pointer, so the MI parameter will result in double counting. To prevent that, we adjust MO by subtracting one MI from it. We adjusted the confidence interval on MO so that the overall model gives the proper confidence interval.

[MS] This comes straight from the per slot costs in Table XII.

[MI] This has two components, loading (Section 5.4.3) and preparing (Table XIII).

[V] This comes from Table VII.

[UB] There are two components, preparing (Table XII) and writing (Section 5.4.4). The prepare cost is the *difference* between the modified and old costs, since MB takes the old cost into account.

[UO] This has a preparing component, the per object costs of Table XII (a difference, as for UB), and a writing component consisting of the per byte writing costs times the number of header bytes (12 for in-place swizzling, 8 otherwise).

[US] This is the difference between the modified and old prepare costs. It may seem strange that it is negative, but the modified case includes per byte costs that the old case does not; once you add in the cost of the 4 bytes to hold the slot, the net is positive.

[UI] Since it is always zero in the measurements, we do not display UI in the tables.

5.5.2. Create-work-save sessions Table XV presents the cost model parameter values for create-work-save sessions. Note that we assume all slots and bytes are initialized. The parameters were calculated as follows:

[PB] This has three sources: creating (Table VI), preparing (Table XII), and writing (Section 5.4.4).

[PO] The sources are the same as for PB, but there is an adjustment similar to that for MO: one PI is subtracted to avoid double counting initialized slots.

[PS] This has two sources, creating and preparing.

[PI] Preparing is the only source for this parameter (Table XIII).

[V] As in load-work-save sessions, this comes from Table VII.

TABLE XV
CREATE-WORK-SAVE PARAMETER VALUES

Scheme	Parameter				
	PB	PO	PS	PI	V
	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)
LCS	3.95 (.11)	65.3 (1.2)	0.79 (.02)	1.81 (.01)	13.14 (.03)
LIS	3.94 (.10)	78.3 (1.6)	0.82 (.03)	1.89 (.01)	13.14 (.03)
ECS	3.95 (.11)	65.3 (1.2)	0.79 (.02)	1.81 (.01)	12.04 (.02)
EIS	3.94 (.10)	78.3 (1.6)	0.82 (.03)	1.89 (.01)	12.04 (.02)
NS	3.75 (.09)	60.7 (1.0)	0.39 (.01)		17.75 (.02)

VI. ANALYSIS AND DISCUSSION

In analyzing the cost models our goal is to determine criteria for choosing an approach to swizzling (or choosing not to swizzle) based on performance. We begin with specific comparisons using the cost models for Mneme presented in the previous section. In particular, we compare lazy versus eager swizzling, copy versus in-place swizzling, and swizzling versus not swizzling, which is in order of complexity of the analysis. We then consider what effect faster and slower id lookup would have. Finally, we offer some limited extrapolations beyond Mneme to other styles of object managers.

6.1. Lazy versus Eager Swizzling

Our experiments showed that lazy swizzling always costs more than eager swizzling. The only load-work-save cost model parameters that are different between lazy and eager swizzling are MB, MO, and V; the only create-work-save difference is V, which is the same as for load-work-save. We show the differences (lazy − eager) and the ratios (lazy / eager) of these parameters in Table XVI, for both copy and in-place swizzling.

Laziness costs up to 16% or 18% more in loading (perhaps a bit more, given the confidence intervals). This effect will be watered down by other costs of loading, especially the overhead of swizzling initialized pointers (MI), and by writing (if there are updates), so in practice it will not be as strong. We are not sure why there should be any effect of laziness on MB, but suspect that it comes from I/O or cache effects that our elapsed time measurements cannot reveal. Slight effects on MO are reasonable since the algorithms vary a little, though the magnitude in the copy swizzling case is a little surprising. This comes about partly because the MB effect of the object headers is added into MO.

The effect on V is the most expected: laziness involves continual dynamic checks that eagerness avoids. We see that in our benchmark setup the overhead is 9%. Since there is little code in the benchmark traversal beyond the recursion itself, programs that do real work would effectively reduce this overhead. While it would take us too far off the subject here, we note that it may be possible to use compiler techniques and program annotations to substantially reduce the number of object residency checks performed at run time [6]–[8], [47],

TABLE XVI
DIFFERENCES AND RATIOS OF LAZY AND EAGER SWIZZLING PARAMETERS

	Parameter					
	MB		MO		V	
	Difference	Ratio	Difference	Ratio	Difference	Ratio
Scheme	μs (±)	Value (±)	μs (±)	Value (±)	μs (±)	Value (±)
LCS versus ECS	0.39 (.19)	1.18 (.10)	4.1 (1.9)	1.18 (.10)	1.1 (.05)	1.09 (.00)
LIS versus EIS	0.33 (.14)	1.16 (.07)	1.8 (1.7)	1.07 (.07)	1.1 (.05)	1.09 (.00)

[48]. This suggests that software- versus hardware-mediated object residency checks may not be a substantial performance issue, and that the strongest argument in favor of page trapping techniques for object faulting is that they do not require compiler support to be transparent to the programmer. The reason that residency checking in software may be acceptable is that current CPU cycle times are very much faster than I/O retrieval times. We will have more to say on this point later.

6.2. Copy versus In-Place Swizzling

Copy swizzling costs more per byte (MB, UB) but less per object (MO, UO). The higher cost per byte is expected (because of copying) and is 6% or 8%. However, the difference is about as large as the confidence interval. We can be reasonably certain there is a difference, but its magnitude is more questionable. At any rate, it is not a huge difference and affects only the swizzling and unswizzling phases of the model, not the work phase.

More surprising is in-place swizzling's higher cost per object. As previously discussed, this comes from its need for an additional 4 header bytes, to support unswizzling. If the cost of loading and writing those extra bytes is subtracted out, then in-place swizzling is indeed a little faster than copy swizzling. Perhaps it would have been better to build a separate table mapping object addresses back to their ids. This would avoid reading and writing the extra bytes, though it would add some CPU overhead for building and searching the table. If that overhead is small enough, then in-place swizzling would be more attractive.

As it is, copy swizzling can be up to 14% to 18% faster for sessions involving update and 5% to 14% faster for read-only sessions. This would happen only for very small objects, and even then is diluted a bit. On very large objects, in-place swizzling has an advantage of up to 6% or 8% (the per byte difference discussed above). If we eliminated the extra 4 header bytes, in-place's per object advantage would be 45±10% in the lazy case and 26±10% in the eager case, which would be watered down by other factors, and reduced substantially by the additional table maintenance.

For create-work-save sessions, copy and in-place swizzling cost about the same, except for the per object costs (PO). Again, the extra 4 header bytes make in-place swizzling cost more (20±5%), and if the header bytes could be removed it would cost slightly less (4%, but less than the confidence interval so not statistically significant).

If we could avoid unswizzling costs for unmodified old objects in the copy swizzling case, then the relationship of copy and in-place swizzling would change in interesting ways. The copy swizzling model parameters would be affected as follows: MO would drop by 4.6μs, MS by 0.77μs, and MI by 1.8μs, while UO would go up by 4.6μs and US by 0.77μs. Also, UI would have to be reintroduced and set to 1.8μs. We would get the differences and ratios between copy and in-place swizzling shown in Table XVII. Because the differences are still positive for MB and UB, as object size increases, eventually the copying cost will outweigh the benefits of copy swizzling, but unless a session is update intensive, copy swizzling will usually perform better than in-place swizzling. The tables in the swizzling versus nonswizzling discussion give further projections of these effects.

Another point is that we did unswizzling using a recursive walk through the objects, rather than iteration over an object table. The iteration is likely to be faster, since it can avoid per object procedure call costs; we estimate it will cost about 1μs, which is 3.4 to 3.6μs less than the benchmark program's cost. This applies to both copy and in-place swizzling, but it is most important if we are trying to skip over unmodified objects when copy swizzling. Further, iteration would actually be required in practice, since changes might make some objects unreachable via the recursive walk, and we would fail to unswizzle those objects properly. (Note that even though the objects might be unreachable from the paths we traverse, they may still be reachable from elsewhere in the store, so we cannot simply claim they are globally unreachable and hence garbage to be reclaimed.)

6.3. Swizzling versus Not Swizzling

In all cases there are net per byte, per object, and per slot overheads to swizzling, and an advantage on each visit. We show the actual differences in Table XVIII. The rows labeled LIS8 and EIS8 give lazy and eager in-place swizzling values adjusted to remove the extra 4 header bytes, for comparison. The lines labeled LCSP and ECSP give the projected costs for copy swizzling that avoids traversal to find modified objects. This assumes that there is no cost for unmodified objects, and that the total cost for each modified object is as before. In fact, there will be either a small iteration cost per unmodified object (e.g., the estimated 0.8μs for iteration that tests a modified flag in each object) or additional cost when working (to record in a table which objects are modified, so the unmodified objects

TABLE XVII
CURRENT AND PROJECTED DIFFERENCES OF COPY AND IN-PLACE SWIZZLING PARAMETERS

	Parameter, μs (±)							
Scheme	MB	MO	MS	MI	UB	UO	US	UI
LCS-LIS now	0.19 (.19)	−1.3 (1.9)		−0.10 (.02)	0.20 (.01)	−8.5 (2.3)	−.37 (.01)	
projected	0.19 (.19)	−5.9 (1.9)	−.77 (.01)	−1.90 (.01)	0.20 (.01)	−3.9 (2.3)	0.40 (.01)	1.80 (.01)
ECS-EIS now	0.13 (.14)	−3.6 (1.7)		−0.10 (.02)	0.20 (.01)	−8.5 (2.3)	−.37 (.01)	
projected	0.13 (.14)	−8.2 (1.7)	−.77 (.01)	−1.90 (.01)	0.20 (.01)	−3.9 (2.3)	0.40 (.01)	1.80 (.01)

TABLE XVIII
SWIZZLING MINUS NON-SWIZZLING PARAMETER VALUES (LOAD-WORK-SAVE)

	Parameter								
	MB	MO	MS	MI	V	UB	UO	US	UI
Scheme	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±)	μs (±) LCS	μs (±)
LCS	0.79 (.12)	1.4 (1.4)	0.77 (.01)	7.7 (.1)	−4.6 (.1)	0.20 (.19)	7.3 (1.5)	−.37 (.01)	
LCSP	0.79 (.12)	−3.2 (1.4)		5.9 (.1)	−4.6 (.1)	0.20 (.19)	11.9 (1.5)	0.40 (.01)	1.80 (.01)
LIS	0.60 (.11)	2.7 (1.3)	0.77 (.01)	7.8 (.1)	−4.6 (.1)		15.8 (2.2)		
LIS8	0.60 (.11)	−6.8 (0.9)	0.77 (.01)	7.8 (.1)	−4.6 (.1)		2.3 (1.8)		
ECS	0.40 (.11)	2.7 (1.3)	0.77 (.01)	7.7 (.1)	−5.7 (.0)	0.20 (.19)	7.3 (1.5)	−.37 (.01)	
ECSP	0.40 (.11)	−7.3 (1.3)		5.9 (.1)	−5.7 (.0)	0.20 (.19)	11.9 (1.5)	0.40 (.01)	1.80 (.01)
EIS	0.27 (.07)	0.9 (1.2)	0.77 (.01)	7.8 (.1)	−5.7 (.0)		15.8 (2.2)		
EIS8	0.27 (.07)	−7.3 (1.0)	0.77 (.01)	7.8 (.1)	−5.7 (.0)		2.3 (1.8)		

need not be touched at all). Also, an iterative approach will likely reduce the total cost for modified objects. However, the projection gives a sense of the maximum possible performance of copy swizzling for read-only sessions.

Though some MO differences are negative, note that, since there must be at least one initialized pointer pointing at each object, each object has at least one MI cost, and the average object size must be at least 4 bytes. Taking these facts into account, the MO advantage of swizzling never overcomes the MI and MB cost, so the net per object cost of swizzling is always positive. The same kind of argument pertains to US, since each US cost involves 4·UB.

What are the extremal cases of swizzling versus nonswizzling? It is easy to calculate the maximum possible advantage to swizzling in each scheme, by taking the limit as the number of visits goes to infinity. Lazy swizzling can take as little as 74% the time of nonswizzling, and eager swizzling can take as little as 68%. The downside of swizzling is much worse, and occurs as we allow object size to go to infinity, with the objects full of initialized pointers. Table XIX shows these limit ratios. Note that in these limits, per object costs do not matter, so the header size issue of in-place swizzling does not arise. If all objects are updated, the maximum ratio is diluted to about 1.5.

Of course, one is more likely interested in typical rather than extremal cases, but there is not a lot of hard evidence (yet) as to what might be typical for persistent object systems and object-oriented databases. We can, however, extrapolate from existing object systems such as Smalltalk and Trellis[7] [49]. In Smalltalk the average object size exclusive of headers

[7]Trellis, a registered trademark of Digital Equipment Corporations, was formerly called Trellis/Owl.

TABLE XIX
WORST-CASE LIMITS OF SWIZZLING/NON-SWIZZLING TIME

Scheme	LCS	LIS	ECS	EIS
Ratio (±)	2.64 (.10)	2.55 (.09)	2.42 (.09)	2.36 (.07)

is about 40 bytes, half of which are devoted to slots. This suggests that about half of object contents are pointers, but that would be an overestimate since many slots in Smalltalk objects contain integers. A better estimate would be that half of the slots (i.e., one quarter of the space in objects) is devoted to slots containing pointers, the vast majority of which are initialized. Every Smalltalk object also includes a pointer to its class object, but we will not count that since special, faster techniques might be used to handle type/class markers in object headers. In sum, then, we might assume an average of 2 initialized pointers per object in 2.5 slots per object. A remaining question is whether or not the number of pointers per object will generally scale with object size or stay relatively fixed. Rather than taking a position on this issue, we will consider two "typical" models, one with a fixed number of pointers per object (2 initialized pointers in 2.5 slots) and one with a varying number (20% of user data containing initialized pointers, with 25% of user data being slots).

Under the fixed model, MO subsumes MS and MI, and under the varying model MB subsumes them. Similarly, UO (or UB) subsumes US and UI. The net result is that we have reduced the number of model variables to be considered and can make more concrete comparisons. Specifically, we can determine the number of visits to each object required to pay back the cost of swizzling each byte in the object and the per object swizzling cost. That is, we can determine the number of visits v required to break even (or achieve other levels

TABLE XX
VISITS FOR BREAK-EVEN, FIXED NUMBER OF POINTERS PER OBJECT

Scheme	Parameters		v for various values of b (±)			
	c (±)	a (±)	b = 20	b = 40	b = 100	b = 200
LCS	4.06 (.38)	0.17 (.03)	7.5 (.9)	10.9 (1.5)	21 (3)	38 (6)
LCSP	1.87 (.35)	0.17 (.03)	5.3 (.9)	8.7 (1.5)	19 (3)	36 (6)
LIS	4.39 (.36)	0.13 (.03)	7.0 (.9)	9.6 (1.4)	17 (3)	30 (6)
LIS8	2.33 (.26)	0.13 (.03)	4.9 (.8)	7.5 (1.3)	15 (3)	28 (5)
ECS	2.56 (.27)	0.07 (.02)	4.0 (.7)	5.4 (1.1)	10 (2)	17 (4)
ECSP	0.79 (.25)	0.07 (.02)	2.2 (.7)	3.6 (1.0)	8 (2)	15 (4)
EIS	3.23 (.26)	0.05 (.01)	4.2 (.5)	5.1 (0.8)	8 (2)	13 (3)
EIS8	1.80 (.21)	0.05 (.01)	2.7 (.5)	3.7 (0.7)	7 (2)	11 (3)

TABLE XXI
VISITS FOR BREAK-EVEN, VARYING NUMBER OF POINTERS PER OBJECT

Scheme	Parameters		v for various values of b (±)			
	c (±)	a (±)	b = 20	b = 40	b = 100	b = 200
LCS	0.30 (.31)	0.27 (.03)	5.6 (.9)	10.9 (1.5)	27 (3)	53 (6)
LCSP	−.69 (.30)	0.24 (.03)	4.0 (.9)	8.7 (1.5)	23 (3)	46 (6)
LIS	0.59 (.29)	0.23 (.03)	5.1 (.8)	9.6 (1.4)	23 (3)	46 (6)
LIS8	−1.47 (.19)	0.23 (.03)	3.0 (.7)	7.5 (1.3)	21 (3)	44 (5)
ECS	−.47 (.23)	0.15 (.02)	2.4 (.6)	5.4 (1.1)	14 (2)	29 (4)
ECSP	−1.28 (.22)	0.12 (.02)	1.2 (.6)	3.6 (1.0)	11 (2)	23 (4)
EIS	0.16 (.21)	0.12 (.01)	2.6 (.5)	5.1 (0.8)	13 (2)	25 (3)
EIS8	−1.27 (.17)	0.12 (.01)	1.2 (.4)	3.7 (0.7)	11 (2)	24 (3)

of relative performance) given the object size b; the form of the equation is $v = c + a \cdot b$, with c and a determined by the model and our fixed or varying number of pointers assumptions. Note that c is the number of visits required to pay back the per object costs and a is the number of visits per byte required to overcome the per byte costs. Tables XX and XXI give the values of c and a for the various schemes, and show v for an illustrative range of object sizes b. The tables give the calculations for read-only sessions; sessions involving update generally take a few more visits to pay back the total swizzling and unswizzling cost.

Because of the wide range of values, there is no simple answer to whether or not swizzling is a good idea—it depends strongly on the application's characteristics, both in terms of the objects it manipulates and in terms of the work it performs on them (number of visits, etc.). While the break-even numbers are useful, it is also important to know how sensitive the

TABLE XXII
SWIZZLING/NON-SWIZZLING FOR ONE VISIT, FIXED NUMBER OF POINTERS PER OBJECT

Scheme	Fixed Number of Pointers			
	b = 20	b = 40	b = 100	b = 200
LCS	1.38 (.06)	1.40 (.06)	1.42 (.07)	1.43 (.07)
LCSP	1.25 (.05)	1.31 (.06)	1.38 (.07)	1.41 (.07)
LIS	1.35 (.05)	1.35 (.06)	1.34 (.06)	1.34 (.06)
LIS8	1.23 (.05)	1.27 (.05)	1.30 (.06)	1.32 (.06)
ECS	1.22 (.05)	1.22 (.05)	1.22 (.06)	1.22 (.06)
ECSP	1.09 (.05)	1.13 (.05)	1.18 (.06)	1.20 (.06)
EIS	1.23 (.04)	1.21 (.04)	1.18 (.04)	1.17 (.04)
EIS8	1.13 (.03)	1.14 (.04)	1.14 (.04)	1.15 (.04)

TABLE XXIII
SWIZZLING/NON-SWIZZLING FOR ONE VISIT, VARYING NUMBER OF POINTERS PER OBJECT

Scheme	Varying number of pointers			
	b = 20	b = 40	b = 100	b = 200
LCS	1.27 (.05)	1.40 (.06)	1.54 (.07)	1.61 (.07)
LCSP	1.18 (.05)	1.31 (.06)	1.46 (.07)	1.53 (.07)
LIS	1.24 (.05)	1.34 (.06)	1.46 (.06)	1.52 (.07)
LIS8	1.12 (.04)	1.27 (.05)	1.42 (.06)	1.49 (.07)
ECS	1.11 (.05)	1.22 (.05)	1.34 (.06)	1.40 (.07)
ECSP	1.01 (.05)	1.13 (.05)	1.26 (.06)	1.32 (.06)
EIS	1.12 (.04)	1.21 (.04)	1.30 (.04)	1.34 (.04)
EIS8	1.02 (.03)	1.14 (.04)	1.26 (.04)	1.32 (.04)

relative costs are if the number of visits is varied. We checked this and found that, in the region near the break-even point, increasing (decreasing) the number of visits by a factor of 2 decreased (increased) the cost ratio (swizzling/nonswizzling) by about 10%. That is, for swizzling to take 90% the time of nonswizzling, you need twice as many visits as indicated in the tables; for swizzling to be within 20% of nonswizzling, one needs only one quarter of the visits in the tables, etc. Thus the cost ratio is not extremely sensitive to the number of visits. We also see in Tables XXII and XXIII that the worst case (namely $v = 1$) for swizzling "typical" objects is usually within 20%–40% of the nonswizzling cost, which is readily reduced if there is any significant work being done or there are more visits. Though we cannot dismiss the costs as being insignificant, they are small enough that other factors might override the potential performance advantage or disadvantage of swizzling.

The general nature of the results is the same for create-work-save sessions. The main difference is that, since persistent objects must be created in all cases, the relative costs are closer and swizzling pays back a little sooner.

6.4. *Understanding Swizzling Costs*

It is a matter of viewpoint whether or not one views the plus or minus 30% range of swizzling advantage/disadvantage as "large" or "small." Many real applications will add enough work to dilute the costs or benefits, too, so it may not matter all that much. However, at first glance one might expect swizzling

to cost a lot less than it actually does—after all, it consists merely of storing the results of object id lookups performed in the nonswizzling case, does it not? That naive view overlooks at least two important factors. First, swizzling examines and converts *all* pointers in each object, whereas an application may use only some of the pointers. Further, since objects are likely loaded in groups, and because there may be some economies of scale to swizzling a set of objects all at once, swizzling may tend to touch and convert more *objects* than a nonswizzling approach. The second factor is that one must also be able to *un*-swizzle objects, which requires maintaining some kind of reverse mapping. We saw that storing the inverse map in the objects for in-place swizzling led to noticeable degradation; it may be possible to get better performance with a separate table, but it will cost more than the EIS8/LIS8 projections shown in previous tables.

Why does lazy swizzling cost more than eager swizzling? One reason is that lazy swizzling, in our implementation, involves continual residency checks avoided by eager swizzling. As previously mentioned, we believe most of these can be eliminated. Even if they are, there appears to be some residual overhead. We hypothesize that laziness does not achieve as much I/O overlap, and hence misses more disk revolutions, and also that laziness, because it leads to more switching back and forth between user code and swizzling code, gives poorer instruction and data cache performance. Unfortunately our measurement approach could not confirm or deny these hypotheses, so we have no definitive explanation of the higher cost of laziness.

We have been over the issue of copy versus in-place swizzling several times. It seems to surprise some people that copy swizzling is not drastically more expensive than it is. One reason copy swizzling may work well is that the necessary copying is quite efficient in the architecture we used, and may be less so in other machines. In particular, the data cache on the MIPS does not load missing blocks before a store proceeds, so copying results in fewer cache misses than on some other machines. The copying routine (memcpy) is also very tightly coded. Another point is that a significant amount of the overhead in swizzling and unswizzling is in the control part of the algorithm: recursing from object to object, looping through the pointers in each object, etc. As previously noted, we believe that we can reduce costs by using iteration rather than recursion. Also, copy swizzling has the advantage of not needing to unswizzle unmodified objects. By providing efficient means to find the modified objects and avoid work on the unmodified ones, we can expect useful algorithmic performance improvement too.

6.5. Swizzling in the Future

There are several projections we can try to make based on the present study. First, what if object id lookup was faster (or slower)? All the schemes depend on id lookup, and hence it would speed swizzling. However, it would also speed nonswizzling by the same absolute amount, so the performance gap would widen because the nonswizzling V parameter would decrease. Also, the sensitivity of swizzling costs to object size, specifically to the number of initialized pointers (MI), would decrease, since the MI parameters would decrease. Overall one would expect nonswizzling to be a little more attractive. In systems where id lookup costs are higher, swizzling would tend to be more attractive than in our study.

What effect will faster CPU's have? While it is hard to predict the effects of future cache implementation strategies such as two-level caches, if CPU speeds increase faster than disk I/O speeds (a combination of seek time and transfer rate since we cannot expect always to obtain sequential reads) then swizzling will be less of an issue because I/O time will be a larger fraction of total time.

What are the prospects for reducing the overheads of swizzling? First, compile-time techniques can reduce the number of residency checks required for lazy swizzling. Also, operating system improvements can make page trapping more attractive for detecting object faults, even though some systems use it already (because they demand transparency). In sum, we believe that laziness will not be a significant issue, and that the eager swizzling measurements give a better sense of the future of swizzling. Second, we can use program annotations and compiler modifications to swizzle some pointers but not all. Given adequate profiling, and program behavior that is predictable enough, this will allow us to obtain the advantages of swizzling and nonswizzling, as well as eager and lazy approaches, simultaneously. Third, compiler generated type-specific code, i.e., swizzling routines customized to each type, may offer improvement, by avoiding loops and unnecessary object header examination and interpretation. Fourth, we can use iteration rather than recursion, when it is possible and reasonable. This will significantly reduce the per-object overheads of swizzling and unswizzling. Finally, we can avoid processing unmodified objects in the copy swizzling case.

If all of these steps are taken, and the benefits are as we expect, then we should achieve performance close to that of the projected eager copy swizzling case shown in the tables. If that is true, then swizzling would usually be advantageous if the number of visits is more than just a few, and the maximum penalty would be about 30%. Eager in-place swizzling would be almost as good, especially if most objects are updated.

VII. Summary and Conclusions

We articulated a model of working with objects, introducing the notions of *collections* of objects, and *work sessions*, and breaking work sessions down into distinct phases for measurement and analysis. We described several methods for managing objects, namely lazy versus eager and copy versus in-place swizzling, as well as not swizzling. We designed and performed experiments to evaluate the schemes and predict their performance. The experimental results support a simple and clear work session performance model, which allows us to draw some conclusions about swizzling, especially in the context of the Mneme object store.

7.1. Conclusions

First, the results support the general expectation that swizzling saves time if one does enough computation with the

swizzled objects. The time savings of swizzling are up to 25% to 35% for applications that do a lot of pointer chasing but not much work with individual objects. In extreme cases swizzling can cost more than twice as much as not swizzling, but a more reasonable expectation is that if objects are visited only once, swizzling costs 20%–40% more than nonswizzling, with the cost of swizzling recovered in anywhere from a few visits to a few tens of visits. It is clear that if an application visits objects only once or a few times, swizzling is unlikely to help and may hurt some. However, swizzling does not result in orders of magnitude impacts on performance, and it is reasonable to choose a swizzling scheme (or not to swizzle) based on factors other than performance. We mentioned a number of possible improvements that reduce swizzling cost and the potential disadvantages of swizzling; the potential advantages of swizzling are limited by the difference between traversing a swizzled or unswizzled data structure, which is not related to the swizzling improvements.

Second, the cost of swizzling goes up with increasing object size and increasing number of pointers to be considered and swizzled. Thus the nature of the collections of objects used in an application determine the extent to which swizzling is beneficial. If we knew more about the collections used in various applications, then we could make more definitive judgments as to whether swizzling is desirable.

Third, as expected, lazy swizzling costs more than eager swizzling, though our measurements do not admit explanation of all of the difference. We speculate that lazy swizzling does not achieve the same degree of CPU-I/O overlap and has poorer cache locality. Similarly, copy swizzling costs more than in-place swizzling, for objects of the same size, when objects are updated. It is interesting that our implementation of in-place swizzling is *slower* than copy swizzling for small to medium sized objects, because we use 4 additional bytes in the disk representation of objects for in-place swizzling. (The 4 bytes are reserved to hold the object's id for unswizzling.) This experience demonstrates how important it is to control per-object space overheads, because of their impact on time performance. We also saw that copy swizzling is generally faster than in-place swizzling in sessions with few updates, if we can avoid processing unmodified objects when unswizzling.

Fourth, since copy swizzling does not cost much more than in-place swizzling, and frequently less, we conclude that given adequate main memory one should use copy swizzling rather than in-place swizzling, because copy swizzling allows much more general transformations between disk and memory formats. Both styles of swizzling can handle issues such as byte order and floating point format, but copy swizzling is more suited to extracting or constructing the fields of objects needed by a specific application, and can help mask schema differences by admitting change of representation of objects in the swizzling/unswizzling process.

Fifth, the cost of object residency checks is not large relative to total cost. This suggests that hardware-versus software-mediated residency checking is not an important performance issue. We note in this context that we believe one can eliminate a significant number of residency checks through compiler optimization and partial eagerness (swizzling most, but not all, pointers eagerly, in the knowledge that the application is very likely to traverse them).

Sixth, the work session model of application behavior allowed us to construct a simple performance model, using parameters of the data set that are relatively easy to measure. Our experiments showed that both the work session model and the performance model derived from it, are reasonable in practice, at least for certain kinds of application behaviors.

Finally, to summarize the conclusions: the costs or benefits of swizzling depend substantially on an application's use of objects, both in the sense of the size and "shape" of the objects (number of pointers, etc.) and in the sense of how the application visits and works on the objects. However, the costs and benefits are not usually large, so others factors may determine which swizzling approach to use. We suggest that, given adequate memory, copy swizzling may be the best approach because of its flexibility.

7.2. Future Work

Considerable work remains on these topics. The work session model of applications needs additional evaluation to determine its applicability to important practical settings. The performance model can be strengthened by further validation. The models, experiments, and object management techniques can be extended to a client/server or other distributed model. Constants in the performance model might be determined for other hardware and software combinations. By studying actual object collections we can establish typical values for the independent variables of the performance model. The question of lazy versus eager swizzling needs further investigation. A cost model allowing objects to be created and destroyed while accessing an existing collection would be more general.

ACKNOWLEDGMENT

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. A number of students and colleagues gave useful comments on various drafts of the paper. Finally, I thank the editor and anonymous referees for their thoughtful criticisms and suggestions, through which the paper was greatly improved.

REFERENCES

[1] S. N. Khoshafian and G. P. Copeland, "Object identity," in [50], pp. 406–416.
[2] T. Kaehler and G. Krasner, "LOOM—large object-oriented memory for Smalltalk-80 systems," in *Smalltalk-80: Bits of History, Words of Advice*, G. Krasner, Ed. Reading, MA: Addison-Wesley, pp. 251–270, 1983, ch. 14.
[3] T. Kaehler, "Virtual memory on a narrow machine for an object-oriented language," in [50], pp. 87–106.
[4] M. Atkinson, K. Chisolm, and P. Cockshott, "PS-algol: An Algol with a persistent heap," *ACM SIGPLAN Not.*, vol. 17, pp. 24–31, July 1982.
[5] J. E. Richardson and M. J. Carey, "Programming constructs for database system implementations in EXODUS," in [51], pp. 208–219.

[6] ——— "Persistence in the E language: Issues and implementation," *Computer Sciences Tech. Rep. 791*, Univ. Wisconsin, Madison, WI, 1988.
[7] J. E. Richardson, "E: A persistent systems implementation language," Ph.D. dissertation, Computer Sciences Dept, Univ. Wisconsin, Madison, WI, Aug. 1989. Available as *Computer Sciences Tech. Rep. 868*.
[8] ———, "Compiled item faulting: A new technique for managing I/O in a persistent language," in [52], pp. 3–16.
[9] S. Riegel, F. Mellender, and A. Straw, "Integration of database management with an object-oriented programming language," in [53], pp. 317–322.
[10] A. Straw, F. Mellender, and S. Riegel, "Object management in a persistent smalltalk system," *Software: Practice and Experience*, vol. 19, pp. 719–737, Aug. 1989.
[11] A. Goldberg and D. Robson, *Smalltalk-80: The Language and its Implementation*. Reading, MA: Addison-Wesley, 1983.
[12] J. W. Schmidt and M. Mall, "Pascal/R report," *Rep. 66*, Fachbereich Automatik, Univ. Hamburg, Hamburg, Germany, Jan. 1980.
[13] J. W. Schmidt, H. Eckhardt, and F. Matthes, "DBPL report," *Tech. Rep. DBPL-Memo 111-88*, Fachbereich Informatik, Johann Wolfgang Goethe-Universität, Frankfurt, Germany, 1988.
[14] F. Matthes and J. W. Schmidt, "The type system of DBPL," in [34], pp. 255–260.
[15] R. Agrawal and N. H. Gehani, "ODE (Object Database and Environment): The language and the data model," in *Proceedings of the 1989 ACM SIGMOD Int. Conf. Management of Data*, pp. 36–45, 1989.
[16] ———, "Rationale for the design of persistence and query processing facilities in the database language O++," in [34], pp. 25–40.
[17] D. Stemple, A. Socorro, and T. Sheard, "Formalizing objects for databases using ADABTPL," in [53], pp. 110–128.
[18] F. Bancilhon, T. Briggs, S. Khoshafian, and P. Valduriez, "FAD: A powerful and simple database language," in *Proceedings of the Thirteenth International Conference on Very Large Data Bases*, pp. 97–105, 1987.
[19] G. Copeland and D. Maier, "Making Smalltalk a database system," in *Proc. 1984 ACM SIGMOD Int. Conf. Management of Data*, pp. 316–325, 1984.
[20] F. Bancilhon *et al.*, "The design and implementation of O_2, an object-oriented database system," in [53], pp. 1–22..
[21] T. Bloom and S. B. Zdonik, "Issues in the design of object-oriented database programming languages," in [54], pp. 441–451.
[22] M. P. Atkinson and O. P. Buneman, "Types and persistence in database programming languages," *ACM Computing Surveys*, vol. 19, pp. 105–190, June 1987.
[23] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita, "Object and file management in the EXODUS extensible database system," in *Proc. Twelfth Int. Conf. Very Large Databases*, pp. 91–100, 1986.
[24] K. R. Dittrich, W. Gotthard, and P. C. Lockemann, "DAMOKLES—A database system for software engineering environments," in *Proc. Int. Workshop on Advanced Programming Environments*, June 1986.
[25] S. Hudson and R. King, "CACTIS: A database system for specifying functionally-defined data," in [55], pp. 26–37.
[26] T. Andrews and C. Harris, "Combining language and database advances in an object-oriented development environment," in [54], pp. 430–440.
[27] D. Maier, J. Stein, A. Otis, and A. Purdy, "Development of an object-oriented DBMS," in [50], pp. 472–482.
[28] C. Lécluse, P. Richard, and F. Velez, "O_2, an object-oriented data model," in *Proc. 1988 ACM SIGMOD Int. Conf. Management of Data*, (Chicago, Illinois, May 1988), pp. 424–433, 1988.
[29] F. Velez, G. Bernard, and V. Darnis, "The O_2 object manager, an overview," in [34].
[30] W. Kim *et al.*, "Integrating an object-oriented programming system with a database system," in [56], pp. 142–152.
[31] D. H. Fishman *et al.*, "Iris: An object-oriented database management system," *ACM Trans. Office Information Syst.*, vol. 5, pp. 48–69, Jan. 1987.
[32] M. J. Carey *et al.*, "Storage management for objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. H. Lochovsky, Eds. New York: ACM Press/Addison-Wesley, ch. 14, pp. 341–369, 1989.
[33] J. E. B. Moss and S. Sinofsky, "Managing persistent data with Mneme: Designing a reliable, shared object interface," in [53], pp. 298–316.
[34] R. Hull, R. Morrison, and D. Stemple, eds., *Proc. Second Int. Workshop on Database Programming Languages*, 1989.
[35] J. E. B. Moss, "Design of the Mneme persistent object store," *ACM Trans. Inf. Syst.*, vol. 8, pp. 103–139, Apr. 1990.
[36] P. Cockshott, "Stable virtual memory," in *Proc. Second Int. Workshop on Persistent Object Systems: Their Design, Implementation, and Use*, pp. 470–476, 1987.
[37] R. Connor *et al.*, "The persistent abstract machine," in [57], pp. 353–366.
[38] J. L. Keedy and J. Rosenberg, "Support for objects in the MONADS architecture," in [57], pp. 392–405.
[39] A. Skarra, S. B. Zdonik, and S. P. Reiss, "An object server for an object oriented database system," in [55], pp. 196–204.
[40] A. Purdy, B. Schuchardt, and D. Maier, "Integrating an object server with other worlds," *ACM Trans. Office Inform. Syst.*, vol. 5, pp. 27–47, Jan. 1987.
[41] W. R. Rubinstein, M. S. Kubicar, and R. G. G. Cattell, "Benchmarking simple database operations," in [51], pp. 387–394.
[42] R. G. G. Cattell, "Object-oriented DBMS performance measurement," in [53], pp. 364–367.
[43] J. Duhl and C. Damon, "A performance comparison of object and relational databases using the Sun benchmark," in [56], pp. 153–163.
[44] V. Benzaken and C. Delobel, "Enhancing performance in a persistent object store: Clustering strategies in O_2," in [52], pp. 403–412.
[45] T. L. Anderson *et al.*, "The Tektronix HyperModel benchmark specification," *Tech. Rep. 89-05*, Computer Research Laboratory, Tektronix Laboratories, Aug. 1989.
[46] D. J. DeWitt *et al.*, "A study of three alternative workstation-server architectures for object oriented database systems," in *Proc. Sixteenth Int. Conf. Very Large Databases*, pp. 107–121, 1990.
[47] A. L. Hosking and J. E. B. Moss, "Toward compile-time optimisations for persistence," in [52], pp. 17–27.
[48] ———, "Compiler support for persistent programming," *COINS Tech. Rep. 91-25*, Dept. Computer and Information Science, Univ. Massachusetts, Mar. 1991.
[49] C. Schaffert *et al.*, "An introduction to Trellis/Owl," in [50], pp. 9–16.
[50] *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, Sept. 1986, *ACM SIGPLAN Not. 21*, Nov. 1986.
[51] *Proc. 1987 ACM SIGMOD Int. Conf. Management of Data*, May 1987, *ACM SIGMOD Rec. 16*, Dec. 1987.
[52] A. Dearle, G. M. Shaw, and S. B. Zdonik, Eds., *Proc. Fourth Int. Workshop Persistent Object Systems*, Sept. 1990, published as *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990.
[53] K. R. Dittrich, Ed., *Proc. Second Int. Workshop Object-Oriented Database Systems*, vol. 334 of *Lecture Notes in Computer Science*, (Bad Münster am Stein-Ebernburg), Germany, Sept. 1988, *Advances in Object-Oriented Database Systems*, Springer-Verlag, 1988.
[54] *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, Oct. 1987, *ACM SIGPLAN Not. 22*, Nov. 1987.
[55] K. Dittrich and U. Dayal, Eds., *Proc. Int. Workshop on Object-Oriented Database Systems*, IEEE Computer Society Press, Washington, D.C., Sept. 1986.
[56] *Proc. Conf. Object-Oriented Programming Systems, Languages, and Applications*, Sept. 1988, *ACM SIGPLAN Not. 23*, Nov. 1988.
[57] J. Rosenberg and D. Koch, Eds., *Proc. Third Int. Workshop on Persistent Object Systems*, Springer-Verlag, 1990.

J. Eliot B. Moss (S'76–M'81) received the S.B., S.M., E.E., and Ph.D. degrees in electrical engineering and computer science from the Massachusetts Institute of Technology in 1975, 1978, 1978, and 1981, respectively.

He is an Associate Professor in the Department of Computer Science, University of Massachusetts, Amherst, MA. He was a staff programmer at the U.S. Army War College from 1981–1985. His primary research interests are in programming languages, databases and their unification, and relevant architectural, operating system, and distributed system issues. His focus is on the incorporation of new features into object-oriented programming languages, and performance evaluation and improvement of implementations of such languages. He has continuing interest in database systems implementation issues stemming from his participation in the CLU and Argus projects at MIT. Current projects include the Mneme persistent object store, Persistent Smalltalk, and Persistent Modula-3; he also directs the Objects Systems Research Group.

Dr. Moss received a National Science Foundation Presidential Young Investigator Award in 1987. He is a member of the Association for Computing Machinery and the Sigma Xi Research Society of America.