Adaptable Pointer Swizzling Strategies in Object Bases: Design, Realization, and Quantitative Analysis *

Alfons Kemper† Donald Kossmann‡

†Lehrstuhl für Informatik
Universität Passau
Innstr. 33
D–94032 Passau, F. R. G.

‡Graduiertenkolleg "Informatik und Technik"
RWTH Aachen
Ahornstr. 55
D–52074 Aachen, F. R. G.

{*kemper*| *kossmann*}*@db.fmi.uni-passau.de*

Abstract

In this paper, different approaches are classified and evaluated for optimizing the access to main memory resident persistent objects—techniques which are commonly referred to as "*pointer swizzling*". To speed up the access along inter-object references, the persistent pointers in the form of unique object identifiers (OIDs) are transformed (swizzled) into main memory pointers (addresses). Pointer swizzling techniques can be directed into two classes: (1) strategies that allow replacement of swizzled objects from the buffer before the end of an application program and (2) those that outrule the displacement of swizzled objects. Whereas the latter class of pointer swizzling methods has received much attention in recent literature, the first class—i.e., techniques that take "precautions" for the replacement of swizzled objects—has not yet been thoroughly investigated. Four different pointer swizzling techniques allowing object replacement were investigated and contrasted them with the performance of an object manager employing no pointer swizzling. The extensive qualitative and quantitative evaluation—only part of which could be presented in this paper—proved that there is no *one* superior pointer swizzling strategy for *all* application profiles. An adaptable object base run-time system was, therefore, devised that employs the full range of pointer swizzling strategies depending on the application profile characteristics which are determined by, e.g., monitoring in combination with sampling, user specifications, and/or program analysis.

*An excerpt of this report was presented at the 9^{th} Int. Conf. on Data Engineering, Vienna, 1993 [KK93]

Contents

1 Introduction

Object-oriented database systems are emerging as the next generation database technology, especially for advanced, so-called "Non-Standard" applications, such as mechanical CAD/CAM, VLSI design, and software engineering. Despite their superior expressive power in comparison to the established relational systems, the acceptance of the object-oriented systems will ultimately depend on their performance. This is all the more true for the technical application domains where database application programs tend to be particularly *computation-intensive* and, at the same time, are highly performance-critical; for example, an interactive CAD application.

In this paper, a class of techniques was investigated for managing references between main memory resident persistent objects which is commonly referred to as "*pointer swizzling*." Pointer swizzling is a measure to optimize the access to persistent objects in a main memory via such references. In object-oriented systems, objects are referenced by their unique object identifier (OID). Each time an object is referenced on the basis of its OID, the system has to locate the object in the main memory by a table lookup, if it is already memory resident; otherwise, it has to be brought into the buffer. The basic idea of pointer swizzling is to materialize the address of a main memory resident persistent object in order to avoid the table lookup. Thus, pointer swizzling converts database objects from an external (persistent) format containing OIDs into an internal (main memory) format replacing the OIDs by the main memory addresses of the referenced objects. This is, of course, particularly important in *computation-intensive* applications—as experienced in, e.g., the CAD/CAM application domain.

In most of the systems that use pointer swizzling, the choice of a specific swizzling strategy appears to have been strongly influenced by the characteristics of the underlying object manager (e.g., the object lookup mechanism) and/or by the characteristics of the programming language (e.g., pointer swizzling in EXODUS version 1.0 is restricted to swizzling the value of local variables only [SCD90]). Moss [Mos92] was the first who abstracted from system and/or object model characteristics and undertook a systematic—though incomplete—classification of pointer swizzling techniques. The work reported here can be seen as an augmentation along a dimension that was not taken into account by Moss: the possibility that swizzled objects can be replaced from the main memory buffer. Moss only analyzed techniques supporting so-called *Load-Work-Save* and *Create-Work-Save* application scenarios without any object replacement before the *Save* phase of the program.

The *Load-Work-Save* application scenario without object replacement from the main memory buffers may be appropriate for certain application profiles that operate on a restricted data volume. However, our experience in the engineering field—particularly mechanical engineering—indicates that there are many applications that operate on a large data volume. Consider, for example a *long* design transaction where the engineer typically browses through large volumes of data, in search of, say, previously constructed similar design objects in between the actual design phases which are restricted to smaller data volumes. It is very advantageous, of course, if the object system can periodically adjust the active working set of swizzled objects since otherwise, the risk of flooding the main memory with obsolete objects would be too great.

Even if object replacement can be precluded for every single application, the scenario investigated by Moss may not be acceptable for the entire workload of a client machine. On the one hand, hot spots that are accessed by almost any application can be observed; therefore, objects should be cached in a swizzled form even after an application has been committed. On the other hand, many objects are only relevant for only a few applications; the object manager should have the opportunity to displace these from the shared main memory buffers of a multi-tasking client machine.

These observations indicate that object replacement is indeed a realistic demand made on general purpose object managers. In this paper, four alternative pointer swizzling strategies were devised that "take precautions" (i.e., maintain control structures) for the replacement of swizzled objects. In the extensive qualitative and quantitative analysis—only part of which could be shown in this paper—these four strategies are compared with an object manager that does not use pointer swizzling. The analysis indicates that there is no *one* best alternative for all possible (yet realistic) application profiles. Rather, each alternative has its pros and cons.

The observation that there is no one superior technique led to design an object manager with adaptable pointer swizzling techniques for our GOM object base system. Depending on the application profile and the characteristics of the object base, the most superior pointer swizzling approach can be chosen. In addition, a method based on monitoring was developed that determines the most efficient swizzling strategy.

The remainder of this paper is organized as follows. In the next section, the setting of the investigation is described. Then, in Section 3, the various pointer swizzling strategies are classified. In Section 4, the architecture of the adaptable object manager is presented. Section 5 defines a cost model, and Section 6 includes performance experiments of our adaptable object manager. On the basis of the cost model, Section 7 outlines how in practice the most efficient swizzling strategy can be determined. The paper is concluded in Section 8.

2 Architectures of Object Management Systems

In this section, the setting of the investigation is described. According to M. Winslett, besides pointer swizzling three crucial architectural decisions must be made when designing an object base system [Win93]:

1. Whether to buffer only pages or to copy objects into an object cache [KBC+88]

2. Whether to use logical or physical object identifiers [KC86]

3. Whether to use an object, page or file server [DFMV90].

Fig. 1 depicts the client-server architecture of an object base system. Pointer swizzling is embodied in the object manager that operates on every client. Given a reference to an object, the object manager carries out lookups and updates of the object and the creation of new objects, during which any I/O activity is carried out implicitly; i.e., transparently to an application. The object manager also provides concurrency control and recovery. Examples of object managers that provide this functionality are the EXODUS storage manager [CDRS86], GemStone [MS87], ORION [KBC+88], Mneme [MS88] and O_2 [VBD89].

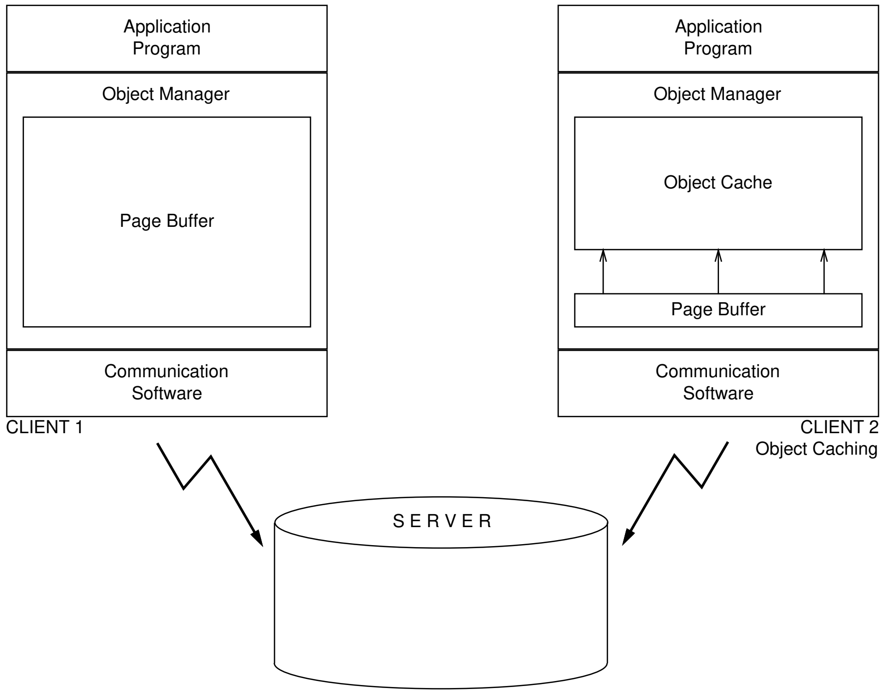


Figure 1: Architecture of an Object Base System

Pointer swizzling takes effect when a resident object is accessed; i.e., an object that is stored in the client's buffer pool. The techniques devised in this work do not influence the mechanism to locate and fetch objects from the server and, therefore, no restrictions are imposed on the implementation of object identity or the server. Vice versa, the techniques can be used equally well with logical as with physical OIDs or with an object, page or file server.

In addition, pointer swizzling can be incorporated in an object manager that only maintains a page buffer (e.g., that of *CLIENT 1* in Fig. 1) and in an object manager that provides an object cache (e.g., that of *CLIENT 2*), again without restricting the design choice. The performance experiments (cf. Section 6), however, indicate that pointer swizzling is usually more effective in a copy architecture because such an architecture improves the locality of an application and reduces some of the swizzling-specific overhead.

3 Pointer Swizzling Techniques

In this section, the approach to pointer swizzling is described. A variety of techniques from literature and existing systems is discussed and a classification scheme is given that allows classifying any technique known to the authors.

3.1 The Method of Pointer Swizzling

When an application accesses an object, it passes a reference to the object manager. In secondary storage, references are implemented as unique object identifiers (OID) [KC86]. On the basis of the OID, the object manager can locate the object in secondary storage and load it into the main memory buffer pool.

Classification of Pointer Swizzling Techniques		
eager	direct	EDS
	indirect	EIS
	optimistic	EOS
lazy	direct	LDS
	indirect	LIS
	optimistic	LOS
no-swizzling		NOS

Table 1: Classification Scheme

If the object is already main memory resident, the object manager has to locate the object in the buffer pool. The *resident object table* (ROT) in which all resident objects are registered realizes the mapping from OIDs to main memory addresses. Each time the object manager accesses a resident object on the basis of its OID, the ROT must be consulted. This is where pointer swizzling takes effect. Pointer swizzling avoids consulting the ROT by materializing the main memory address of the resident objects and converting (*swizzling*) references into main memory pointers.

In conclusion, the advantage of pointer swizzling over no-swizzling is that using pointer swizzling, resident objects can be accessed more efficiently. On the other hand, pointer swizzling induces an overhead by swizzling and unswizzling references (i.e., converting swizzled references back to OIDs).

3.2 Classification of the Techniques

A large variety of techniques have already been incorporated in existing systems. Moss was the first to undertake a systematic classification and thorough analysis of pointer swizzling techniques [Mos92]. However, Moss only investigated techniques that preclude object replacement (referred to as *optimistic* swizzling in our classification scheme). Contrary to this, we concentrated on techniques that provide the necessary control structures for object replacement. In general, pointer swizzling techniques can be classified into two dimensions depending on the following two criteria:

1. When is a reference swizzled?

2. What precautions must be taken for object replacement?

Table 1 summarizes the classification scheme.

3.2.1 Eager and Lazy Swizzling

Eager swizzling insures that all the references in the main memory are swizzled. An object is an aggregation of references and fields representing other atomic or complex values. When an object is loaded from disk, the object is "scanned through" and all the references in the object are immediately swizzled.

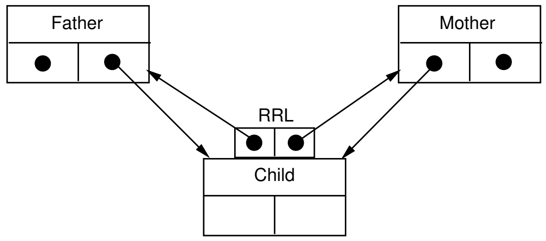


Figure 2: Direct Swizzling and the RRL

In contrast to eager swizzling, *lazy* swizzling only swizzles references on demand. The advantage of lazy swizzling is that no reference is swizzled that is not read and, therefore, cannot be dereferenced by the application. On the negative side, lazy swizzling must handle two different representations of references at run-time: swizzled and non-swizzled references. Software checks that determine the state of a reference (swizzled or not swizzled) need to be included every time an object is accessed.

White and DeWitt distinguished between lazy swizzling *upon discovery* and *upon dereference* [WD92]. Lazy swizzling upon discovery swizzles a reference as soon as it is *read*; e.g., when a reference is copied from a field of an object into a local variable, the system makes sure beforehand that the reference in the field is swizzled and then copies the swizzled reference into the variable. On the other hand, lazy swizzling upon dereference swizzles a reference when it is *dereferenced*.

Lazy swizzling upon discovery carries the danger of swizzling some (but usually very few) references that are never dereferenced. However, in GOM, as well as in EXODUS, swizzling upon discovery is incorporated for all lazy swizzling techniques because lazy swizzling upon dereference often fails to swizzle any inter-object references since references are often copied into local variables before they are dereferenced. A lot of potential is lost, therefore, when using lazy swizzling upon dereference.

3.2.2 Direct and Indirect Swizzling

Depending on whether it is possible to swizzle a reference which refers to an object that is not main memory resident, *direct* and *indirect* swizzling can be distinguished. Direct swizzling requires that the referenced object be resident. A directly swizzled reference contains the main memory address of the object it references. If an object is displaced from the system buffer, i.e., is no longer resident, all the directly swizzled references that refer to the displaced object need to be unswizzled. To unswizzle these references, they are registered in a list called *reverse reference list (RRL)*. Fig. 2 illustrates this scenario.

In the case of eager direct swizzling, simply unswizzling the references is not possible because eager swizzling guarantees that all references in the buffer are swizzled. Instead those references (i.e., their "home objects") must be displaced, too. This may result in a "snowball" effect. On the other hand, a snowball effect can also be observed with eager direct swizzling when an object is brought into the buffer. Non-resident objects that are referenced by this object must be brought into the buffer, too, leading to the preloading of the transitive closure.

Maintaining an RRL can be costly, especially if the fan-in of an object is high. In this context, the *fan-in* of an object is defined as the number of swizzled references that

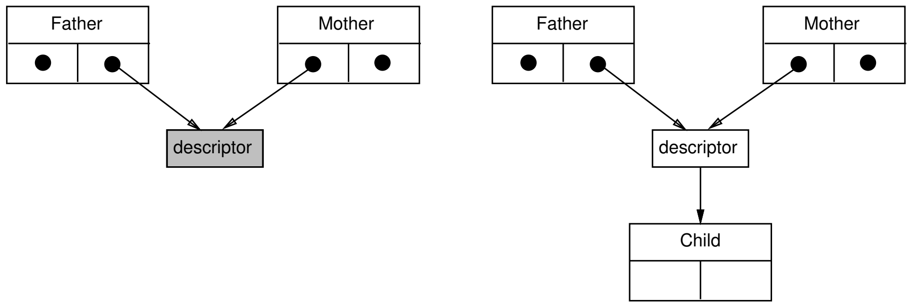


Figure 3: The Scenario of Indirect Swizzling

refer to that object. Let us assume that an attribute of an object is assigned a new value. First, the RRL of the object, the old value of the attribute referenced, must be updated. Then, the attribute needs to be registered in the RRL of the object it now references.

An alternative to setting up an RRL for every resident object dynamically could be to materialize the reverse references and store them persistently in the object base; such an approach was chosen in the design of the materialization of functions [KKM91]. In this case, reverse references would have to be allocated dynamically only for transient structures, e.g., variables. However, structural updates and object replacement would become even more expensive and result in additional I/O activity.

Indirect swizzling avoids the overhead of maintaining RRLs by permitting the swizzling of references that reference non-resident objects. To attain indirect swizzling, a swizzled reference materializes the address of a *descriptor*, i.e., a placeholder of the actual object. If the referenced object is resident, the descriptor stores the main memory address of the object; if not, the descriptor is marked invalid. If an object is displaced, the swizzled references that refer to the object need not be unswizzled; only the descriptor is marked invalid. Fig. 3 depicts this scenario; the shading marks the left-hand descriptor invalid.

For garbage collection, every descriptor keeps a counter that counts the number of indirectly swizzled references referring to the descriptor (i.e., the fan-in). Maintaining a counter is much cheaper than maintaining an RRL when executing updates or when swizzling and unswizzling references. On the other hand, indirect swizzling induces an additional overhead in comparison to direct swizzling when it comes to simple object lookups. Indirect swizzling leads to an additional indirection, due to the descriptor, and to a residency check, a check of whether the descriptor is valid. For direct swizzling, the information that an object is resident is coded in the swizzled reference.

3.2.3 Optimistic Pointer Swizzling in Virtual Memory

Recently, a class of swizzling techniques has been proposed that makes use of the virtual memory facility of the operating system with hardware support [Wil90, WD92]. The goal is to convert the references into direct pointers (as for direct swizzling) but on the other hand to avoid the overhead of maintaining RRLs by simply precluding object replacement and thus the unswizzling of references before the end of an application. The lack of space in a physical main memory is overcome by the swapping facility of the operating system.

However, these swizzling techniques face two crucial disadvantages:

1. Restrictions are imposed upon an application. In principle, an application cannot

System	*Technique Used*	*Reference*
LOOM	eager indirect	[KK83]
PS-algol	lazy optimistic	[ACCM83, CAC+84]
ORION	eager indirect	[KBC+88]
O_2	no-swizzling	[BBB+88]
EXODUS/E	lazy optimistic	[SCD90, WD92]
ObjectStore	eager optimistic	[LLOW91]
GOM	adaptable swizzling	[KK93]

Table 2: Classification of the Swizzling Techniques in Existing Systems

access more data than the size of the virtual memory. For example, in the configuration of our work stations, virtual memory is restricted to 90 MB. This limitation is due to the client's swap capacity. It can be unacceptable for long design transactions.

2. Swizzling in virtual memory usually precludes caching pages after commit of an application in a *hot* form; i.e., so that all references remain swizzled.

3. Additional page faults may occur at commit time of an application because references on pages that were modified need to be unswizzled. If a page that has been modified is swapped out, it has to be brought into physical main memory again, unswizzled, and finally, written back to secondary storage.

Since these techniques assume that an application meets the restriction that object replacement can be precluded, they are referred to as *optimistic* swizzling.[1] A thorough quantitative evaluation was carried out by Moss [Mos92]. He investigated *eager* and *lazy* swizzling in an object manager that copies objects into an object cache as well as in an object manager that swizzles the references *inplace* in the page buffer pool.

One way to implement swizzling in virtual memory is to make use of memory mapping [LLOW91, BGW92, NSS89]. The idea here is to keep the references in secondary storage in a swizzled form. The references, however, usually have to be converted anyhow. In effect, therefore, memory mapping coincides with eager pointer swizzling.

3.2.4 Catalogue of Existing Systems

Pointer swizzling is very effective for computation-intensive applications. Consequently, some form of pointer swizzling is embodied in many of the existing commercial systems and prototypes. In Table 2, some of the systems are classified according to their swizzling facilities. Apart from direct swizzling, every other swizzling technique can be found there.

[1] Only Wilson [Wil90] tries to overcome this restriction by access-protecting all cached pages periodically. However, once a page is swizzled and free to access, its replacement is precluded in his approach, too.

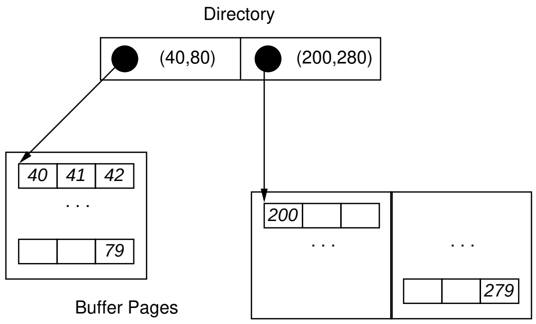


Figure 4: The Directory of a List

3.3 Swizzling in Large Objects

In many applications, large objects are encountered whose size exceeds significantly the size of one page, e.g., a satellite picture or the set of all the citizens of Cairo. One of the strong points of object-oriented database systems is that they allow well-tuned implementations of such data structures. Of course, in these implementations making use fo pointer swizzling is also desirable. In this section, some of the principal difficulties concerning pointer swizzling in the context of large objects are addressed. An *open* architecture that integrates the implementations of various type constructors (e.g., lists, matrices and sets) and which can easily be extended is described in [BKK+92].

From the point of view of pointer swizzling, the following two questions arise when considering large objects:

1. Only a small fraction of a large object is usually main memory resident. What is the main memory address of a large object, i.e., what address is materialized by pointer swizzling? For example, what is the address of a list whose 3^{rd} and 1066^{th} element are resident but are not stored in a contiguous chunk?

2. How are indexes on pointer-valued fields dealt with when references are swizzled?

3.3.1 Large Object Descriptors

The fact that usually only a small fraction of a large object is main memory resident precludes direct (and optimistic) swizzling of references referring to a large object. A descriptor for the large object, called *directory*, must be maintained. Fig. 4 depicts the in-memory implementation of lists in GOM [Kot92]. Each time an element of a list is accessed, the directory of the list must be consulted. This is where swizzling takes effect. Using indirect swizzling the address of the directory will be materialized. On the other hand, under no-swizzling, the ROT has to be consulted before the large object can be accessed via the directory. However, the savings obtained by swizzling are watered down by the relatively high cost of accessing a large object via the directory.

3.3.2 Swizzling and Indexes

A reference may be a key of an index, e.g., of *Access Support Relations* [KM92] or of an index of a large set. It does not make any sense to swizzle these references since, in most cases, they are never dereferenced. Furthermore, swizzling these references could induce a complete reorganization of the index, e.g., a B-tree [Com79] or a hash table [FNPS79]. As a consequence, an index cannot be consulted on the basis of a swizzled reference as key. Before a query can be evaluated, the reference must be "translated" into its non-swizzled format inducing a small additional overhead.

To conclude, large objects in general restrict the use of direct swizzling, and in addition, indexes restrict the use of eager swizzling. The most efficient use of pointer swizzling, however, will always strongly depend on the implementation of the data structure.

4 Adaptable Pointer Swizzling

In the preceding section, the pros and cons of the individual swizzling techniques were discussed. It became apparent that there is no *one* best strategy for *all* situations. In particular, the following observations can be made:

1. The performance of a specific swizzling technique depends strongly on the profile of an application; e.g., for applications with very high locality, direct swizzling will be favorable whereas for applications with very low locality, no-swizzling should be preferred.

2. Large objects restrict the use of pointer swizzling. These restrictions should have no influence on operations with *small* objects.

3. Descriptors and RRLs can consume a significant amount of main memory (cf. Section 5.3). An adaptable object manager is able to control the storage overhead by allowing swizzling only in cases of a high running time speed-up per storage overhead ratio.

These observations led us to devise an *adaptable* object manager for our object base system GOM that provides the flexibility to choose the most favorable swizzling technique. Specific performance experiments that justify adaptable pointer swizzling are reported in [KK93]. How this adaptability can be achieved with as little additional overhead as possible is discussed in this section.

4.1 Architecture of Adaptable Pointer Swizzling

In an adaptable object manager, a module for each of the swizzling techniques is embodied. The idea is to divide statically (i.e., at compile-time) all references an application dereferences into disjoint granules. All references of the same granule are swizzled uniformly; i.e., exactly one module operates on a specific granule. Each time a reference is dereferenced, the corresponding swizzling module is called accordingly. Fig. 5 depicts this scenario schematically.

It is very important that the mapping of the references to the granules be static for every application. This is the only way to avoid additional software checks every time a reference is dereferenced. For example, if a program chooses to swizzle all the references referring to objects of type *Person* eagerly and directly, the compiler must know this to generate calls of the swizzling module *EDS* for accesses to *Persons*.

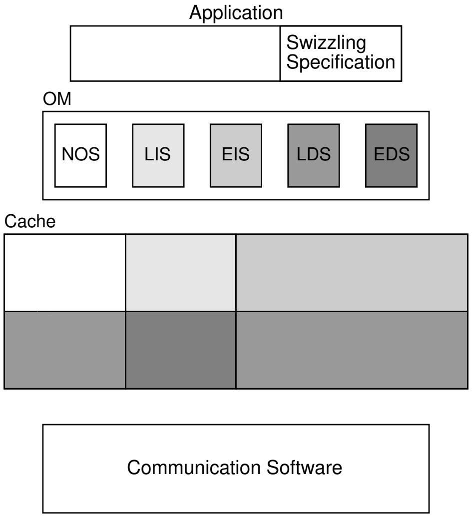


Figure 5: Granularities of Adaptable Pointer Swizzling

To specify the mapping from references to granules, additional code usually has to be generated for an application program. The mapping is discussed in detail in Section 4.2 where four different granularities are proposed.

4.1.1 Conflicting Applications

If two applications conflict with each other, i.e., swizzle some references differently, it is assumed that they run in isolated buffers or are scheduled in such a way that they do not run in parallel. Fig. 6 illustrates why, in general, it is even impossible that two conflicting applications that run in parallel share any objects in the same buffer. Applications *App1* and *App2* traverse the *a* field of object *id1* and then operate on the *b* field of object *id2*. Both applications assume that reference *a* is swizzled directly. However, *App1* assumes that reference *b* is swizzled, too, whereas *App2* assumes that this reference is not swizzled. Consequently, each application must keep its own copy of object *id2* in the desired representation. In addition, each application must also keep its own copy of object *id1*, because this object contains a pointer to object *id2*.

4.1.2 Buffering Pages after Commit

When an application is finished, all the pages remain in the client's buffer pool. This improves performance when several successive transactions use many of the same pages. However, using adaptable pointer swizzling, the cached objects may not have the correct representation for subsequent applications. For example, application *App1* swizzles all references directly. After *App1* is terminated, application *App2* which assumes that all references are swizzled indirectly, is run. Consequently, the references need to be *reswizzled* before they may be dereferenced by application *App2*.

To overcome these difficulties, we proposed a *lazy reswizzling* approach, i.e., references are reswizzled as soon as their "home objects" are accessed for the first time by application *App2*. To realize lazy reswizzling, the objects are protected as long as they have not been accessed. When application *App1* commits, a flag is set in the entries of the ROT and,

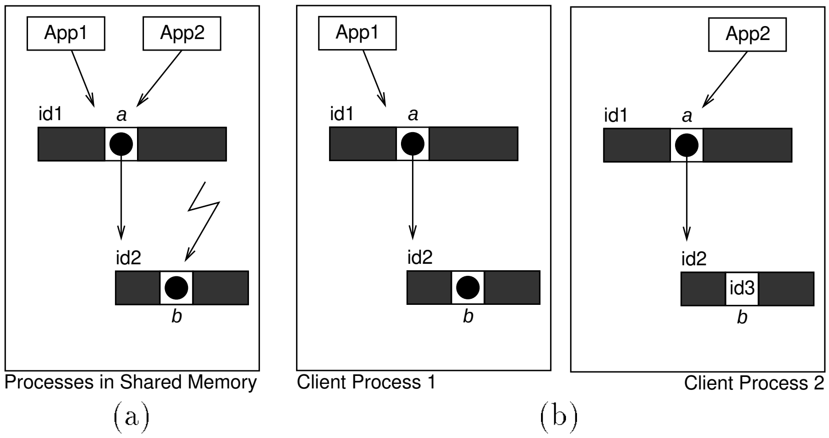


Figure 6: Conflicts due to Adaptable Pointer Swizzling

if existent, in the descriptor of every cached object, indicating that the object is resident but not necessarily in the correct representation. If the object is accessed for the first time, the object manager *snaps* and fixes the representation according to the specification of application *App2*.

Eager direct swizzling can again induce a snowball effect. As soon as an object is accessed, the representation of all objects that are referred to by this object need to be investigated because the object manager cannot snap if a directly swizzled reference is dereferenced. This effect is completely coherent with the eager loading effect of eager direct swizzling observed in Section 3.2.

4.2 Granularities of Pointer Swizzling

4.2.1 Application-Specific Swizzling

The "coarsest" way to realize adaptable pointer swizzling is to swizzle uniformly all the pointers within one application. This approach is called *application-specific* swizzling. The swizzling strategy to follow is specified at the beginning of the execution of an application, thereby activating the corresponding swizzling module. For example, application-specific swizzling allows following lazy direct swizzling for applications that traverse the object base with high locality and eager indirect swizzling for applications that carry out structural updates in addition (Fig. 7).

4.2.2 Type-Specific Swizzling

Usually, an application invokes several operations that access objects with varying profiles. In this scenario, application-specific swizzling is not always the optimum solution as it does not provide the flexibility to choose different swizzling techniques for different references within the same application. One possibility to achieve more flexibility is to swizzle in the *type-specific* mode; i.e., all the references that refer to objects of the same type are swizzled uniformly.

Fig. 8 depicts a typical situation from the OO1 benchmark [CS92]. All references

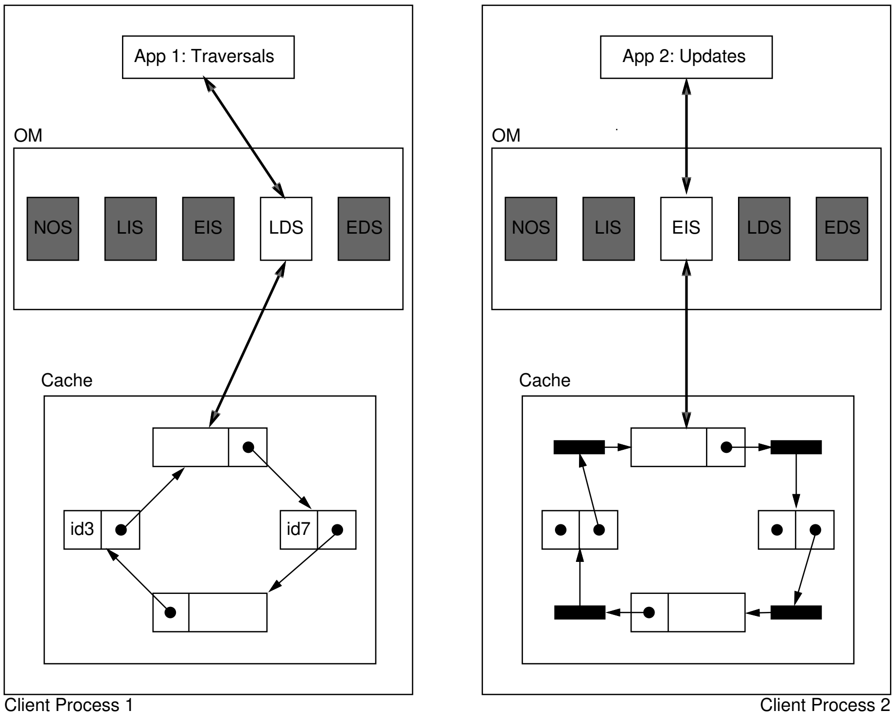


Figure 7: The Scenario of Application-Specific Swizzling

that refer to *Parts* (i.e., those in the *Connections*) are swizzled eagerly and indirectly whereas all the other references are swizzled eagerly and directly. This example does not emphasize that within the same object, references can be swizzled differently; for type-specific swizzling, the type of the *referenced* object (and not the type of the "home object") determines how a reference is swizzled.

The example illustrates how type-specific swizzling allows the use of eager direct (and optimistic) swizzling, the most efficient technique for object lookups. In application-specific mode, this technique usually has to be precluded because it induces a snowball effect when an object is loaded or displaced from the buffer pool. Type-specific swizzling stops this snowball effect when a *Connection* is reached.

Type-specific swizzling exploits strong typing in the programming interface of the object base system (e.g., [KMWZ91]). Only in strongly typed languages, the compiler can determine the type of a reference and generate code accordingly. Furthermore, the property of determining how the references that refer to objects of a specific type are swizzled must be inherited by all subtypes; the refinement of this property must be precluded.

To specify type-specific swizzling, the compiler generates a function for every type. Each time an object is loaded, the corresponding function is called using *late binding*, and swizzling is carried out accordingly. In the following, the code of the functions for the example in Fig. 8 is shown.

```
funct fetch_conn(c:Conn)
begin
    swizzle_INDIRECT(c, from);
    swizzle_INDIRECT(c, to);
end
```

```
funct fetch_part(p:Part)
begin
    swizzle_DIRECT(p, connTo);
end
```

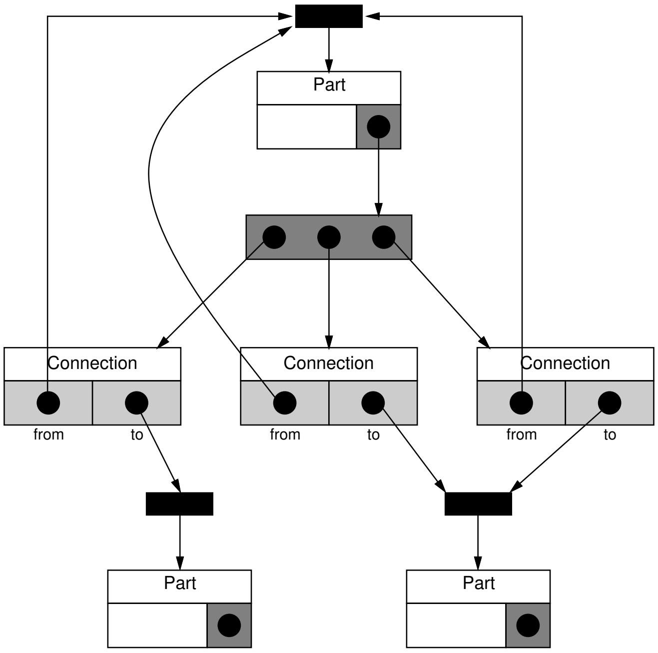


Figure 8: Type-Specific Swizzling in the OO1 Benchmark

Every *Connection* has a *from* and a *to* field that refer to a *Part* each and are swizzled eagerly and indirectly as soon as the *Connection* is brought into a main memory buffer.

In conclusion, type-specific swizzling exploits better the merits of the swizzling techniques than application-specific swizzling. Calling a function using late binding every time an object is loaded, however, induces an additional per-object overhead.

4.2.3 Context-Specific Swizzling

An even finer granularity can be achieved by the so-called *context-specific* swizzling mode. Here, a reference is swizzled according to the *context* it is stored in. For example, Fig. 9 depicts a situation within the OO1 benchmark in which the references in the *to* fields are swizzled eagerly whereas the references in the *from* fields are swizzled lazily.

Again, to specify context-specific swizzling, a function is generated for every type that is called when an object is loaded and induces the same overhead as for type-specific swizzling. Subsequently, the code is given for *Parts* and *Connections* in our example. In the function *fetch_conn* only the *to* field is swizzled indirectly. The *from* field remains untouched at this point, as lazy swizzling is specified for the references in these fields.

funct fetch_conn(c:*Conn*)
begin
    swizzle_INDIRECT(c, *to*);
end

funct fetch_part(p:*Part*)
begin
    swizzle_DIRECT(p, *connTo*);
end

The finer granularity provided by context-specific swizzling has, in some cases, to be paid for by additional overhead. For example, consider the following assignment:

myConn.to := myConn.from;

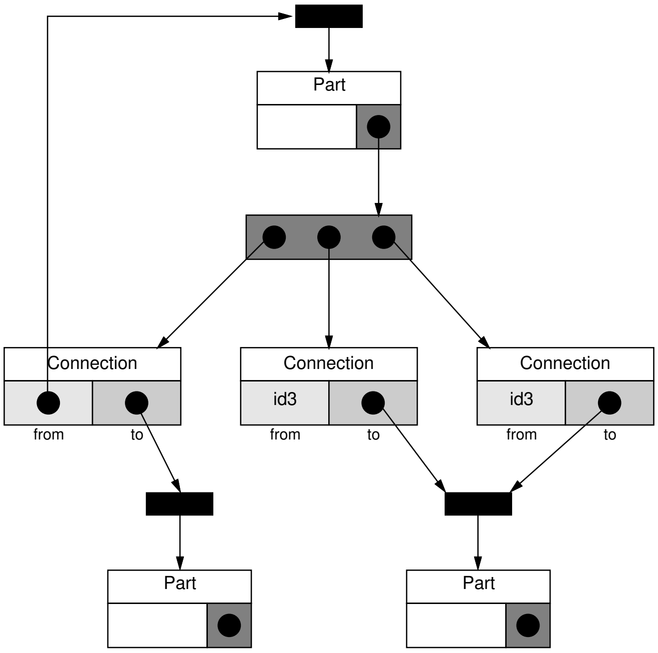


Figure 9: Context-Specific Swizzling in the OO1 Benchmark

If the *to* fields are swizzled directly and the *from* fields are swizzled indirectly, the indirectly swizzled reference of *myConn.from* has to be "translated" before it is copied into *myConn.to*. Translations may also have to be carried out to evaluate Boolean expressions such as *myConn.from* = *yourConn.to*. In application as well as type-specific swizzling, translations are never necessary.

A very specific class of context is defined by variables. In our notion, the identifier of each variable defines its own context. Often a special treatment of variables is followed, e.g., in the first version of EXODUS [SCD90] or in [Kos91]. Incorporating such a technique is a very obvious application of context-specific swizzling.

4.2.4 Reference-Specific Swizzling

The finest granularity conceivable is to swizzle every reference individually. However, this approach is rarely promising. Due to aliasing, the same reference can be converted several times when an application is run. For example, in the OO1 benchmark a *Part* may be referenced by *Connection_A* and *Connection_B*. Of course, this situation cannot be known at compile-time and the decision to swizzle the reference in the *Part* that is referred to by *Connection_A* indirectly and to swizzle the reference in the *Part* that is referred to by *Connection_B* directly results in a conflict that has to be resolved at run-time at high cost.

Only in very obvious cases, e.g., the repetitive computation of a path expression in a loop, can reference-specific swizzling be profitable. However, in these cases, compile-time techniques, e.g., code motion [MR79], will probably do a much better job. Consequently, reference-specific swizzling was not investigated any further.

Session variables	
l	Number of lookup operations performed.
u	Number of update operations performed.
$\overline{fi}$	Average fan-in of an object; i.e., the number of swizzled references that refer to an object.
$m(st)$	Number of references that are swizzled. Depends on whether eager or lazy swizzling is used.

Table 3: Session Variables

Cost functions	
SW	Cost to swizzle a reference.
US	Cost to unswizzle a reference.
LO	Cost to carry out a lookup.
UP	Cost to carry out an update.

Table 4: Cost Functions

5 Performance Analysis

In this section, a cost model is devised for application, type and context-specific swizzling. With the help of this cost model the following two decisions can be made, provided that the profile of an application and the characteristics of the object base are known exactly: (1) in what granularity to swizzle and (2) what technique to use for every granule. Section 7 outlines how to make the right swizzling decision in practice with the help of this cost model.

On the basis of the cost model, best and worst case analyses are carried out. In addition, the storage overhead of pointer swizzling is analyzed.

5.1 A Cost Model for Application-Specific Swizzling

The cost of executing an application is determined by I/O activity and CPU time. However, we are only interested in swizzling specific costs and, therefore, focus on the CPU time that is spent to access objects in main memory and to convert references. For application-specific swizzling these costs, depending on the technique used—denoted *st*—can be summarized by equation (1). This function is an adaptation of the cost model used by Moss [Mos92]. Table 3 describes the session variables and Table 4 describes the *SW*, *US*, *LO*, and *UP* functions.

$$C(st) = m(st) * (\mathrm{SW}(st, \overline{fi}) + \mathrm{US}(st, \overline{fi})) + l * \mathrm{LO}(st) + u * \mathrm{UP}(st, \overline{fi}) \tag{1}$$

Aside from the swizzling technique used, *SW, US* and *UP* depend on the fan-in of an object. Unfortunately, it is not possible to determine in general the exact fan-in of an object. Only an average fan-in $\overline{fi}$ can be determined as a rule. Obviously, considering

Lookup	*TC*	*EDS*	*LDS*	*EIS*	*LIS*	*NOS*
int	1.0	3.6	4.0	4.3	4.7	23.4
reference	0.9	6.7	7.1	7.4	7.8	26.4

Table 5: Object Lookups in μs

SW+US	$fi = 0$	$fi = 1$	$fi = 2$	$fi = 3$	$fi = 8$
direct	85.1	59.2	63.0	67.8	85.0
indirect	62.2	33.6	33.6	33.6	33.6

Table 6: Swizzling and Unswizzling a Reference in μs

the average fan-in only is a source of inaccuracy. Nevertheless, the experiments reported in [KK93] showed that this approximation works very well in almost any case.

5.1.1 Determining the Functions of the Cost Model

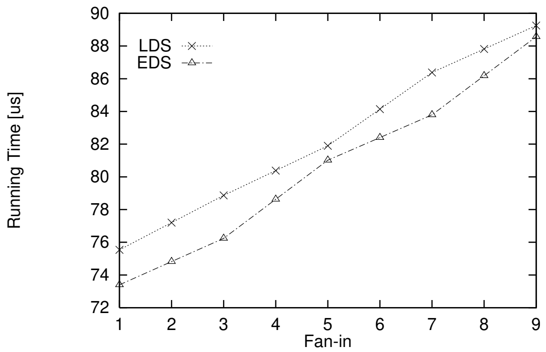
Table 5 lists the costs to read a field of an object containing an **int** (4 bytes) or a reference (8 bytes).[2] We concluded that eager direct swizzling has the potential of being 6.5 times more efficient than no-swizzling. The results in [Mos92] that lazy swizzling is up to 10 % inferior to eager swizzling for pure lookups were confirmed.

Transient C (called *TC*) was also measured. The tremendous gap between transient C and eager direct swizzling (which is also observed for the pointer swizzling techniques in [WD92]) is due to two factors: in the first place, the object manager was based on the EXODUS storage manager which provides user-descriptors that cause an additional indirection; then, each time an object was accessed, the object manager set a flag to carry out an *LRU* replacement strategy. This procedure was omitted in *TC*. The gap was even wider for lookups on fields that contain references. In transient C, pointers that are 4 bytes long are copied, whereas in a persistent object manager, swizzled and non-swizzled references that are 8 bytes long are copied.

Table 6 lists the costs of swizzling and unswizzling a reference. For direct swizzling, these costs depend on the fan-in, as the RRL of the referenced object must be maintained. For $fi = 0$, the costs are particularly high, as in this case, an RRL (or a descriptor) is allocated when the reference is swizzled, and destroyed when the reference is unswizzled again. With increasing fan-in, the cost to swizzle and unswizzle a reference directly grow proportionally, whereas the cost to swizzle and unswizzle a reference indirectly are constant.

Similarly, the cost of direct swizzling grows linearly with increasing fan-in for an operation that updates a field that contains a reference (cf. Figure 10a). On the other hand, the cost for indirect swizzling is constant. The time to modify a field that contains an **int** was also measured (cf. Figure 10b). In this case, direct swizzling outperforms

[2]See Section 6.1 for details of the benchmark environment.



(a)

Update	*reference*	**int**
TC	1.3	1.3
EDS	–	29.4
LDS	–	29.7
EIS	32.1	30.1
LIS	33.3	30.4
NOS	48.7	46.6

(b)

Figure 10: Object updates in μs: (a) for direct swizzling if a field that contains a reference is modified, and (b) if a field that contains a reference or an **int** is modified.

indirect swizzling by approximately the same difference as for lookups since no RRL needs to be maintained.

The gap between transient C and pointer swizzling is even greater for updates than for lookups. This is due to the fact that the object manager has to mark the updated objects. This overhead, which is very expensive, is precluded under transient C.

5.1.2 Best and Worst Case Analysis

Considering, equation (1) and the quantities of the cost functions, Table 7 compares the techniques in the best cases. For example, in an application that carries out lookups only, lazy direct swizzling (*LDS*) outperforms no-swizzling (*NOS*) by a factor of 5.9 (equivalent to 83% savings). On the other hand, in an application that dereferences every reference only once, swizzling will not be worthwhile and no-swizzling outperforms lazy direct swizzling by a factor of 6.8. Analyzing this worst case of direct swizzling, it was assumed that $\overline{fi} = 25$.

The drawback of eager swizzling can become arbitrarily high (∞) due to the fact that references are swizzled unnecessarily, i.e., without being dereferenced. When comparing eager direct and eager indirect swizzling, the fact that eager direct swizzling may cause additional I/O activity was disregarded.

In realistic applications, these extremes will hardly ever be reached. However, they underline impressingly the need to use pointer swizzling carefully in an adaptable system.

5.2 Cost Models for Type and Context-Specific Swizzling

For type-specific swizzling, equation (1) is adapted in two ways. One, the costs induced by every granule are summed up. Accordingly, equation (1) is applied to every type t considering the type-specific parameters, $m_t(st)$, l_t and u_t, and the type-specific technique, st_t. Two, the cost to call the type-specific function for every object that is accessed must be considered. Here, o is defined as the number of objects that are accessed by the application and FC as the cost to call this function using late binding. Any swizzling

worst / best	*NOS*	*LIS*	*EIS*	*LDS*	*EDS*
NOS	1	2.9	∞	6.8	∞
LIS	5	1	∞	5.1	∞
EIS	5.4	1.1	1	5.3	5.3
LDS	5.9	1.2	∞	1	∞
EDS	6.5	1.3	1.2	1.1	1

Table 7: Speed-up of the Techniques in the Best Case

that is carried out within the type-specific function is encountered in the $m_t(st_t) * \mathrm{SW}(st_t)$ terms.

$$C = o * FC + \sum_{t \in \mathcal{T}} m_t(st_t) * (\mathrm{SW}(st_t, \overline{fi}) + \mathrm{US}(st_t, \overline{fi})) + l_t * \mathrm{LO}(st_t) + u_t * \mathrm{UP}(st_t, \overline{fi}) \quad (2)$$

Context-specific swizzling may induce an additional overhead due to the translations of references, referred to as function TL in equation (3). This cost depends on the number of translations and on the techniques involved; e.g., it is much cheaper to translate an indirectly swizzled reference into a non-swizzled reference than the other way around. Furthermore, for context-specific swizzling, more granules (referred to as $\mathcal{C}$) need to be considered.

$$C = TL + o * FC + \sum_{c \in \mathcal{C}} m_c(st_c) * (\mathrm{SW}(st_c, \overline{fi}) + \mathrm{US}(st_c, \overline{fi})) + l_c * \mathrm{LO}(st_c) + u_c * \mathrm{UP}(st_c, \overline{fi}) \quad (3)$$

5.2.1 Determining the Additional Cost Functions

In the benchmark environment described in Section 6.1, the overhead for late binding when calling the type-specific function, referred to by FC in equations (2) and (3), is 33.2 μs. Table 8 summarizes the cost to translate a reference into a different layout. It is assumed that the object that is referred to is resident and that a descriptor has been allocated. If two values are given for translating references following lazy swizzling, the first value is the value if the reference is swizzled; "–" means that no translation is necessary.

5.2.2 Best and Worst Case Analysis

It is always possible to specify context-specific swizzling so that no translations need to be carried out. Consequently, the best and worst case analysis of application vs. type and application vs. context-specific swizzling coincide.

The worst case of type and context-specific swizzling is an application that browses through a large number of objects, thereby accessing every object only once. For such an application, no-swizzling is preferable for all references. Equation (4) demonstrates how the speed-up of application-specific swizzling is computed for this extreme case.

l_1 \ l_2	*EDS*	*LDS*	*EIS*	*LIS*	*NOS*
EDS	–	–	2.8	2.8	2.8
LDS	2.3 / 21.1	–	2.8 / 19.1	2.8 / 2.3	2.8 /2.3
EIS	2.8	2.8	–	–	2.8
LIS	2.8 / 21.1	2.8	2.3 / 19.1	–	2.8 / 2.3
NOS	20.7	–	18.0	–	–

Table 8: Time in μs to Translate a Reference from Layout l_1 to Layout l_2

$$\frac{C(typ)}{C(appl)} = \frac{o * FC + o * LO(\mathrm{NOS})}{o * LO(\mathrm{NOS})} = \frac{33.2 + 23.4}{23.4} = 2.42 \qquad (4)$$

The best case of type and context-specific swizzling is an application that dereferences some references with very high locality and others only once. For such an application, type and context-specific swizzling allow eager direct swizzling where it is profitable and no-swizzling where it is not. On the other hand, application-specific swizzling faces a dilemma between no-swizzling and lazy indirect swizzling, if eager swizzling is not profitable because too many references are swizzled unnecessarily, and direct swizzling is outperformed because $\overline{fi}$ is too high.

Equation (5) computes the potential speed-up in the extreme case that only a few objects are accessed, one reference is dereferenced l times, and m other references are dereferenced once. Application-specific swizzling has a break-even point between no-swizzling and lazy indirect swizzling for $m = l * 18.7/43.5$; at this point, the speed-up of type and context-specific swizzling reaches its maximum.

$$\begin{aligned}
\frac{C(appl)}{C(typ)} &= \frac{l * LO(\mathrm{NOS}) + m * LO(\mathrm{NOS})}{o * FC + SW + US + l * LO(\mathrm{EDS}) + m * LO(\mathrm{NOS})} \\
&= \frac{l * LO(\mathrm{NOS}) + l * 18.7/43.5 * LO(\mathrm{NOS})}{o * FC + SW + US + l * LO(\mathrm{EDS}) + l * 18.7/43.5 * LO(\mathrm{NOS})} \\
&\xrightarrow{l \to \infty} \frac{LO(\mathrm{NOS}) + 18.7/43.5 * LO(\mathrm{NOS})}{LO(\mathrm{EDS}) + 18.7/43.5 * LO(\mathrm{NOS})} \\
&= 2.45 \qquad (5)
\end{aligned}$$

5.3 Storage Overhead

As stated in Section 4, storage economy is one reason for using pointer swizzling in an adaptable system. In principle, there are three sources of additional needs of main memory due to pointer swizzling:

1. the descriptors induce a *per object* overhead:

$$o * SD$$

   where SD denotes the size of one descriptor.

2. every entry in an RRL induces a *per reference* overhead:

$$10 * \left\lceil \frac{m}{10} \right\rceil * SR$$

where SR denotes the size of one RRL entry. For running time efficiency, blocks of 10 RRL entries are always allocated and, therefore, internal off-cuts in the blocks have to be accounted for as well.

3. pointer swizzling may result in a larger object code of the application program.

The amount of additional code is not significant in our implementation. Only for type and context-specific swizzling can a small overhead be observed for the *fetch* functions. On the other hand, every RRL entry consumes $SR = 12$ bytes and a descriptor is $SD = 24$ bytes long in our object manager [Kot92, Kos91].

Considering the data structures of the OO1 benchmark, 43% of the main memory has to be invested for each descriptors or RRLs when using eager indirect swizzling or eager direct swizzling, respectively. Here, it must be kept in mind that the OO1 benchmark can be seen as the worst case for pointer swizzling; the objects are very small and contain references with a high density. Furthermore, RRLs may be swapped out since they are rarely used; but descriptors are hot spots because a descriptor is involved every time an indirectly swizzled reference is dereferenced.

6 Performance Experiments

In this section, the analytical results of the last section are confirmed with the help of the OO1 benchmark [CS92]. Diverse application profiles as well as various object base configurations and the influence of object caching and clustering are investigated. Experiments that investigate specifically the break-even points between the swizzling techniques and the influence of the fan-in are presented in [KK93].

6.1 The Benchmark Environment

6.1.1 Software and Hardware Used

All the experiments were carried out on a Sun SPARC station 1 plus with a 40 MB main memory under SunOS 4.1.3. If not stated otherwise, the size of the buffer pool was restricted to 1000 pages at 4096 bytes. In addition, extra main memory was reserved to allocate RRLs and descriptors. Every application had its own private stack and heap for transient data, e.g., local variables.

Access to persistent data was effected by calls to the GOM object manager. The GOM object manager is based on Version 1.3 of the EXODUS storage manager [CDRS86]. As stated in the preceding section, the GOM object manager embodies no-swizzling and each of the four swizzling techniques that take precautions for object replacement. The GOM object manager facilitates application, type and context-specific swizzling.

To read or manipulate the state of an object, a reference to the object is passed to the object manager. For no-swizzling, references are *logical* OIDs consisting of 8 bytes. Directly and indirectly swizzled references are also 8 bytes long.

6.1.2 The OO1 Benchmark

The benchmark used was derived from the OO1 (Sun) benchmark [CS92]. The data structures are defined as follows:

type *Part* **is**
  [*part-id*: **int**;
  type: **string[10]**;
  x, y: **int**;
  built: **int**]
  connTo : {*Connection*}]
end type *Part*

type *Connection* **is**
  [*from*: *Part*;
  to: *Part*;
  type: **string[10]**;
  length: **int**]
end type *Connection*

Considering a 4-byte-alignment, every *Part* was 36 bytes long and a *Connection* consumed 32 bytes. If not stated otherwise, object bases consisting of 20000 *Parts* and 60000 *Connections* were measured. In GOM that maps logical OIDs to physical addresses using a linear hash table with separators [Lar88], such a database consumed 8.9 MB.

The *Parts* and *Connections* were initialized as described in [CS92]; i.e., *part-ids* were numbered from 1 through 20000 and for every *Part*, 3 *Connections* existed whose *from* field referenced that *Part*. To support *Traversals*, references to these 3 *Connections* were materialized in a set that was referred to by the *connTo* field of a *Part*. To obtain locality, the *to* fields of the *Connections* were initialized so that 90% of the *Connections* connected a *Part* with a *Part* that was within the 1% closest.

The benchmark measured applications composed of the following operations:

Lookup: Selecting a random *Part* and reading its x, y and *type* field and calling a null procedure.

Traversal: Finding all *Parts* connected to a randomly selected *Part*, or to a *Part* connected to it, and so on up to a certain depth (by default 7) and reading the x, y and *type* field of every *Part* and calling a null procedure, thereby making use of the sets referred to by the *connTo* field of every part.

Reverse Traversal: Beginning at a randomly selected *Part*, finding all *Parts* it is connected to, and *Parts* these *Parts* are connected to, and so on. This operation is far more complex than the *Forward Traversal*, because in every iteration, the *Connections* whose *to* fields refer to the particular *Part* have to be selected from the set of all *Connections*; references to these *Connections* are not materialized.

Update: Swap twice the values of the *to* fields of two randomly selected *Connections*.

The creation of new objects was not measured. Although usually, the implementation of this operation will depend on the swizzling technique used, there is no swizzling specific cost in creating an object. However, newly created objects will usually be immediately initialized. This initialization can be seen as a sequence of operations that has the same profile as the *Lookup* and *Update* operation.

6.2 Lookups

The *Lookup* operation was measured on a 10000 *Parts* and 30000 *Connections* object base; i.e., an object base in which all *Parts* and *Connections* fit into the main memory buffers

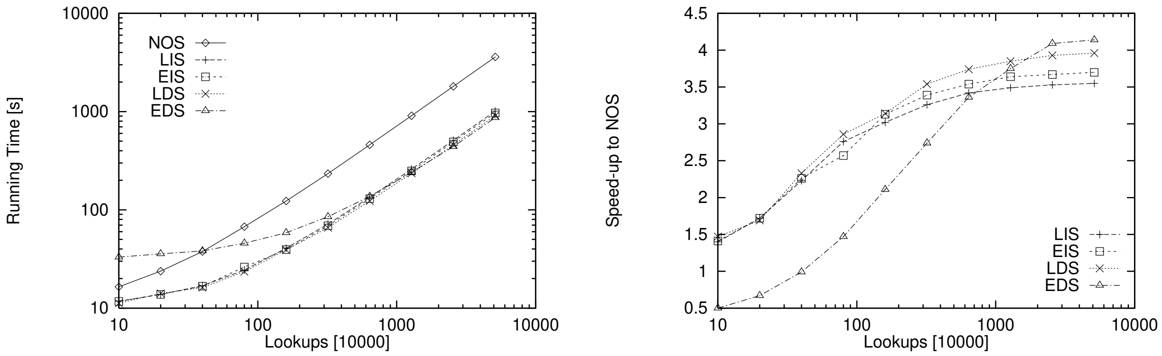


Figure 11: Measuring the *Lookup* Operation

so that eager direct swizzling was a reasonable approach. With increasing computation intensity, swizzling becomes more and more attractive; i.e., the speed-up of swizzling against no-swizzling increases (Fig. 11).

The maximum speed-up that can be achieved is about 4.5. In Section 5, the maximum speed-up was calculated to 6.5. However, here and in most realistic applications, the results are diluted, e.g., by calls of the function "random" or by I/O activity. The maximum possible (theoretic) speed-up, therefore, cannot be achieved.

If very few *Lookups* are carried out, eager direct swizzling (*EDS*) is outperformed dramatically by any other technique because *EDS* loads the transitive closure of a *Part* as soon as the *Part* is accessed and thus induces additional I/O. With an increasing number of *Lookups*, more *Parts* are brought into the buffer anyway, and *EDS* catches up and finally, outperforms any other technique.

Only application-specific swizzling was considered for this experiment. The profile was too homogeneous to exploit the merits of type and context-specific swizzling.

6.3 Traversals

The *Traversals* were carried out *cold*, *warm* and *hot* on a 20000 *Parts* and 60000 *Connections* object base. The cold *Traversals* are I/O bound. In this case, it does not make much difference whether references are swizzled or not, as the running time of the application is dominated by I/O activity rather than in-memory object lookups. Fig. 12a shows the running time of every technique[3] and Fig. 12b the savings obtained by the swizzling techniques as compared to no-swizzling.[4] Only eager indirect swizzling dropped behind due to the unnecessary swizzling of references; it caught up with increasing depth, since with increasing depth, the effect of cycles in the object graph became more apparent.

The warm *Traversals* were carried out in the following way: first, a *Traversal* was carried out using no-swizzling and then, the running time was measured to carry out the same *Traversal* using individual swizzling techniques. This experiment investigated the behavior of the system when pages were buffered in the client after commit of an applica-

[3] *EDS* had to be precluded, as the size of the object base exceeded the size of the main memory buffers.

[4] The savings are defined by the following ratio: $\frac{NOS-SWIZZ}{NOS}$.

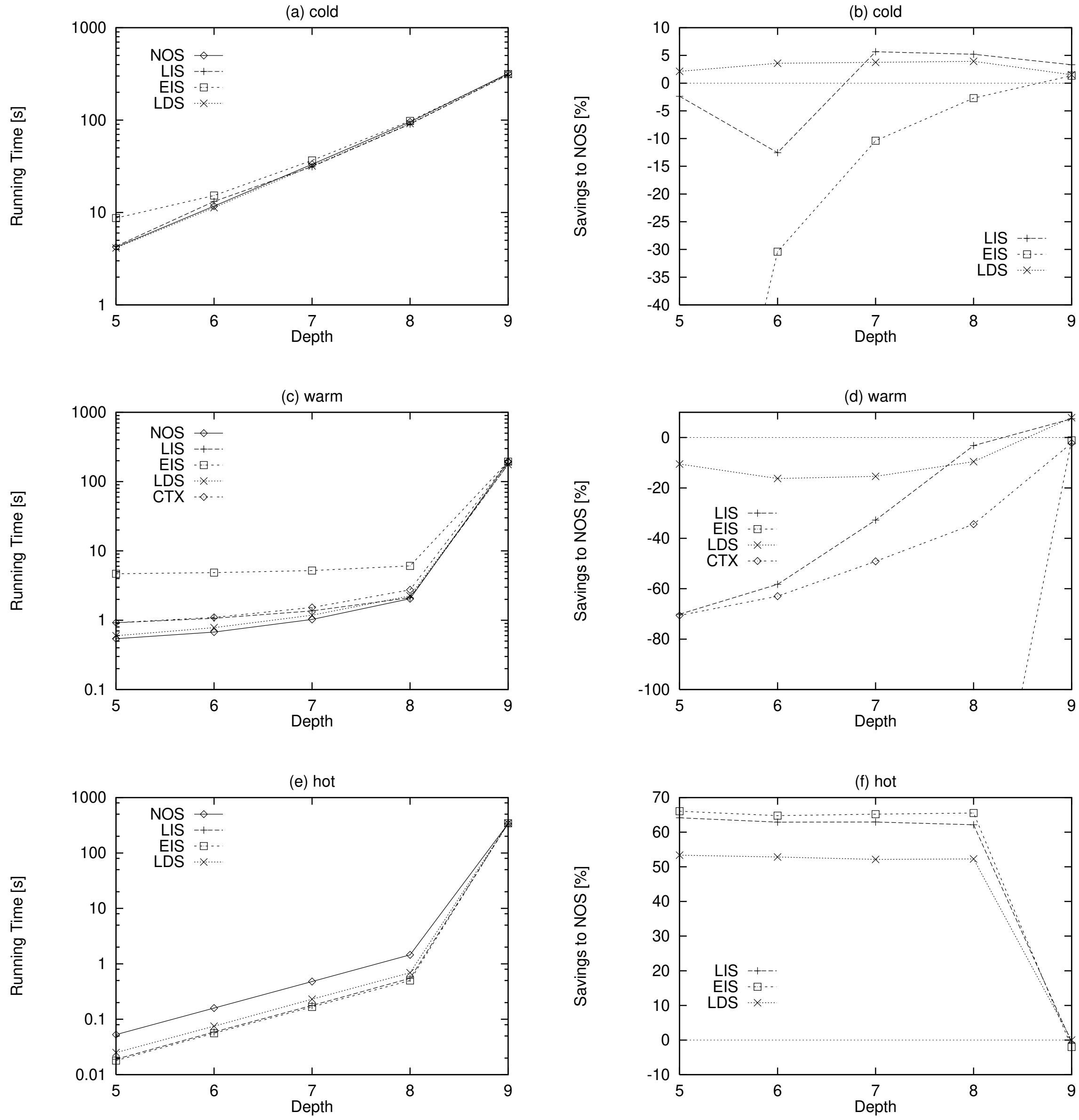


Figure 12: Measuring *Traversals*

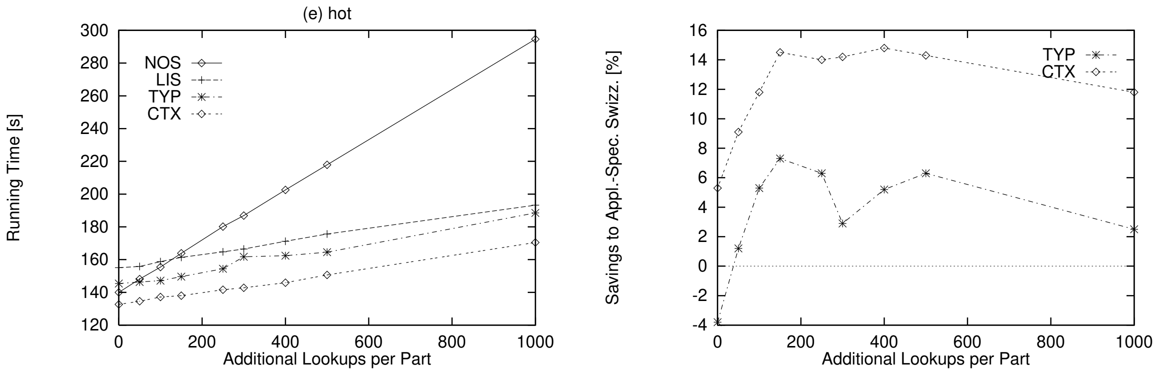


Figure 13: *Traversals* in Combination with *Lookups* (warm)

tion (see Section 4.1); many objects were main memory resident but they were not in the right representation. Again, the running time (Fig. 12c) and the comparison (Fig. 12d) between the swizzling techniques and no-swizzling were reported. Here, context-specific swizzling (referred to as *CTX*) was also investigated to demonstrate how large the losses can become due to calls of the *fetch_part* and *fetch_connection* functions each time an object was loaded or the representation of a resident object was altered the first time it was accessed.

To investigate computation-intensive applications, Figures 12e and 12f summarize the experimental results for hot *Traversals*. Here, a *Traversal* was executed once considering a specific swizzling technique and then, the running time was measured to carry out the same *Traversal* again. Consequently, many objects were main memory resident and, in addition, in the correct representation when the benchmark was executed.

Up to a depth of 9, swizzling outperforms no-swizzling dramatically. Beginning from a depth of 9, so many objects are accessed that most of them are paged out already within the *warm-up* run and the same results are obtained as for cold *Traversals*.

In this experiment, lazy direct swizzling suffered from two phenomena: (1) due to paging many references had to be unswizzled and swizzled again because they were dereferenced again subsequently; and (2) due to the recursion to realize a depth first search, many local variables were involved which led to large RRLs. Even using type and context-specific swizzling, these drawbacks could not be compensated so that eager indirect swizzling turned out to be the most effective technique with up to 70% savings as compared to no-swizzling.

To demonstrate the effectiveness of type and context-specific swizzling, an operation mix of *Traversals* and *Lookups* was measured. Beginning at a randomly selected *Part*, the object base was traversed. With every iteration, the x, y, and *type* fields of the *Part* that was reached were read a number of times in addition. In this experiment, the client was "warmed up" by carrying out beforehand a *Traversal* using no-swizzling.

In this experiment, application-specific swizzling faced a dilemma. On the one hand, no-swizzling should have been used to carry out the warm *Traversal* efficiently; on the other hand, the additional *Lookups* provided enough locality to justify direct swizzling of references referring to *Parts*. Type and context-specific swizzling overcame this dilemma

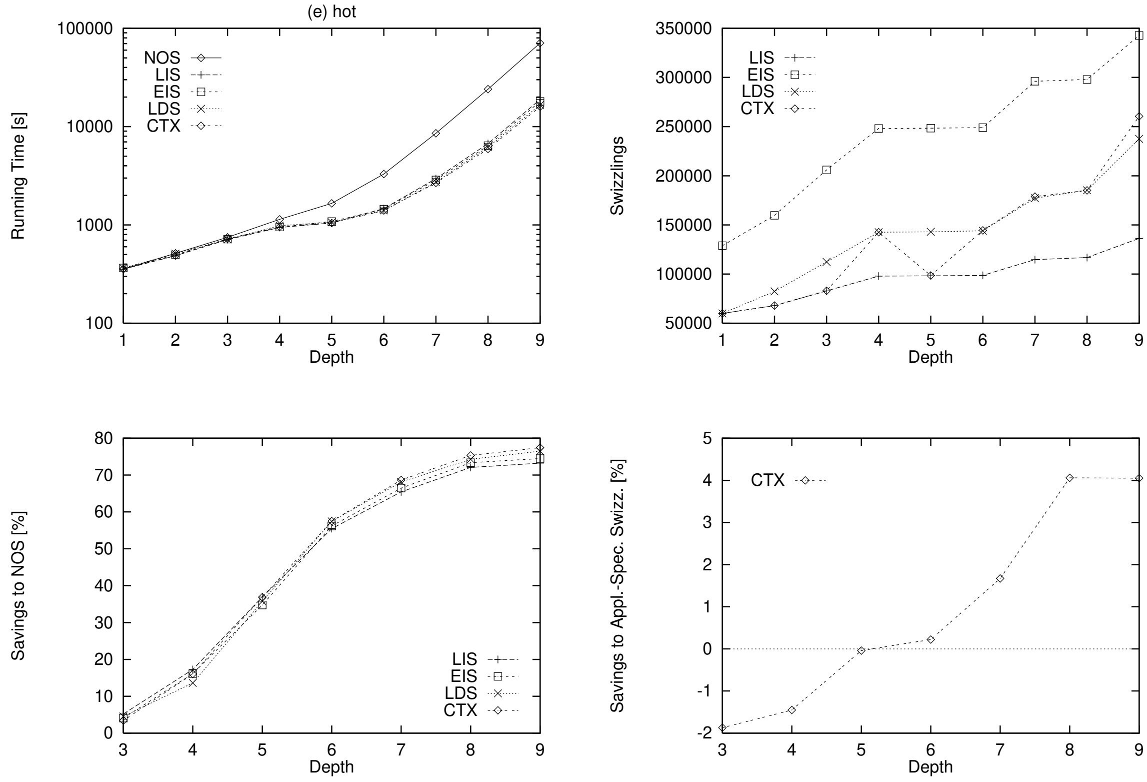


Figure 14: Measuring *Reverse Traversals*

and outperformed application-specific swizzling by savings of up to 16% (Fig. 13).

6.4 Reverse Traversals

The major difference between *Traversals* and *Reverse Traversals* is that the *Traversals* are supported by materializing the *Connections* that are traversed in every iteration in the *connTo* fields of the *Parts*. On the other hand, *Reverse Traversals* must "find" their way through the object graph. This makes a *Reverse Traversal* a much more computation-intensive operation.

Reverse Traversals can be seen as iteratively performing a join of the set of *Connections* on itself. We tried implementing this join with a nested-loop but the running time was not acceptable because not all the *Connections* fitted into the main memory buffers. To improve locality, the set of all *Connections* was partitioned into several disjoint and equally-sized subsets so that every subset fitted into the buffers. Then, iteratively a subset was loaded and as much as possible of the *Reverse Traversal* was executed based on this subset. This approach reduced the number of page faults dramatically and—as will be seen—provided enough locality to make swizzling worthwhile. It is correct in the sense that it computes the same number of *Parts* that are connected to a specific *Part*; however, it does not meet the demand to follow a depth first search strategy.

To reduce the running time of the benchmark, the object base as well as the buffer pool were scaled down. Fig. 14 summarizes the experimental results for a 10000 *Parts* and 30000 *Connections* object base executed on a buffer pool of 500 pages. With increasing

NOS	*LIS*		*EIS*		*LDS*		*EDS*		*TYP, CTX*	
μs	μs	%	μs	%	μs	%	μs	%	μs	%
225	113	49.8	96	57.3	289	-28.4	299	-32.9	74	67.1

Table 9: Running Time in μs and Savings in % of Pointer Swizzling for the *Update* Operation (hot)

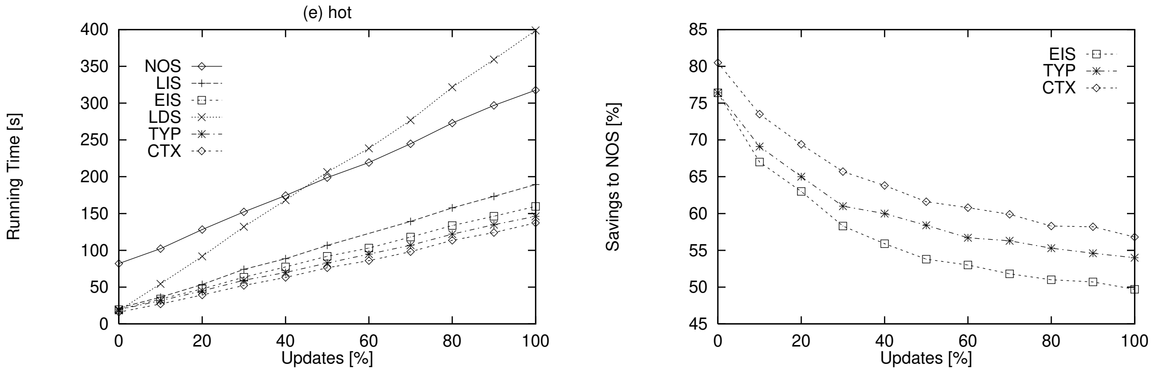


Figure 15: Operation Mix: Updates and Lookups (hot)

depth the running time increased exponentially. However, locality increased as well since each time a subset was loaded, it was operated upon more intensively and thus, swizzling became more attractive. With increasing depth, context-specific swizzling became more attractive, too, as it had the opportunity to exploit eager direct swizzling.

This experiment also makes clear that a tremendous number of swizzlings can be afforded in such a computation-intensive application and, at the same time confirms that swizzling is still worthwhile. Again, lazy direct swizzling carries out more swizzling due to *unlucky* object replacements. However, lazy direct swizzling can compensate for this by its more efficient object lookup mechanism and consequently, outperform indirect swizzling.

6.5 Structural Updates

Table 9 contains the running time for the *Update* operation and the trade-off between swizzling and no-swizzling, if the buffers are hot; i.e., after a *Traversal* had been executed using the same swizzling technique. As expected, direct swizzling causes tremendous overhead due to maintaining the RRLs, and indirect swizzling outperforms no-swizzling by savings of more than 50% since it avoids consulting the ROT. The best performance, however, can be achieved by type and context-specific swizzling since these strategies allow swizzling the references referring to *Connections* directly (thus providing fast accesses to the *Connections*), and not swizzling the references that are redirected at all (thus avoiding any overhead for maintaining RRLs or descriptors). 23% of the running time are saved for this operation when using type or context-specific swizzling against eager indirect swizzling which is the most efficient technique for application-specific swizzling.

Fig. 15 summarizes the experiments for *Updates* in combination with *Lookups*. With an increasing number of *Updates*, the savings of swizzling compared to no-swizzling decreased because *Updates* are relatively more expensive than *Lookups*. On the other hand, type-specific swizzling became more and more attractive than eager indirect swizzling, the most efficient technique considering application-specific swizzling because it is able to make use of direct swizzling. Context-specific swizzling outperformed type-specific swizzling, because it could make use of eager direct swizzling without risking swizzling references unnecessarily.

6.6 The Influence of Locality on Pointer Swizzling

The most decisive parameter for any swizzling technique is the application's *locality*; i.e., the probability that an application accesses an object that is already main memory resident. The higher the locality, the higher the ratio l/m in equation (1) on page 17 and the more profitable swizzling becomes.

Locality is not only determined by the operations an application executes; it is also influenced by the state of the object base and by other optimization techniques, e.g., clustering. It also makes a difference whether the objects are copied from the page buffer into an object cache.

6.6.1 Varying Topological Locality

In the original definition of the OO1 benchmark, 90% of the *Connections* are randomly selected among the 1% of *Parts* that are "closest." This parameter was scaled so that object bases were also investigated in which, for example, all (100%) or no (0%) *Connections* satisfied this demand. This parameter is referred to as the *topological locality* of the OO1 object base.

In Fig. 16, the savings over no-swizzling of the overall best and overall worst swizzling strategies are presented when running *Traversals* and *Reverse Traversals* up to a depth of 7 against object bases with varying topological locality. With increasing topological locality the performance of pointer swizzling improves. The effect is more striking for *Traversals*; it is only beginning at a topological locality of 80% that pointer swizzling becomes worthwhile even for *hot* traversals. *Reverse Traversals* are so computation-intensive that in any case, no-swizzling is outperformed by any swizzling technique.

6.6.2 Object Caching

Locality can be improved dramatically if objects are copied from the page buffer pool into a separate object cache [KBC+88]. In such a copy architecture, most of the *Parts* and *Connections* that are accessed when a *Traversal* is carried out can be cached. On the other hand, in an architecture that buffers only pages, a lot of main memory is wasted with objects that are never accessed, but are stored on the same page together with objects that are accessed.

To investigate the influence of object caching on pointer swizzling, the *Traversal* portion of the OO1 benchmark was run *hot* in a copy architecture (referred to as *OC*) with a 2.46 MB object cache (equivalent to 600 pages) and a page buffer with a capacity of 200 pages (0.82 MB). The results were contrasted with a run of the benchmark in an

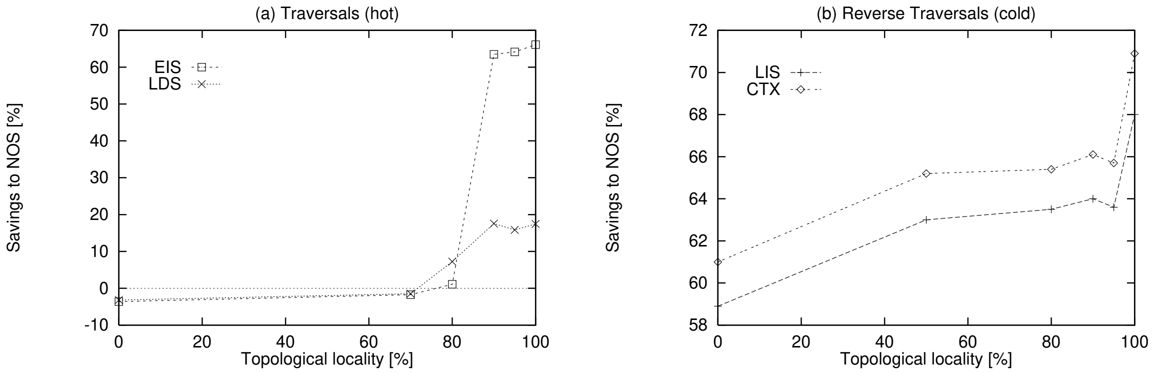


Figure 16: Varying Topological Locality

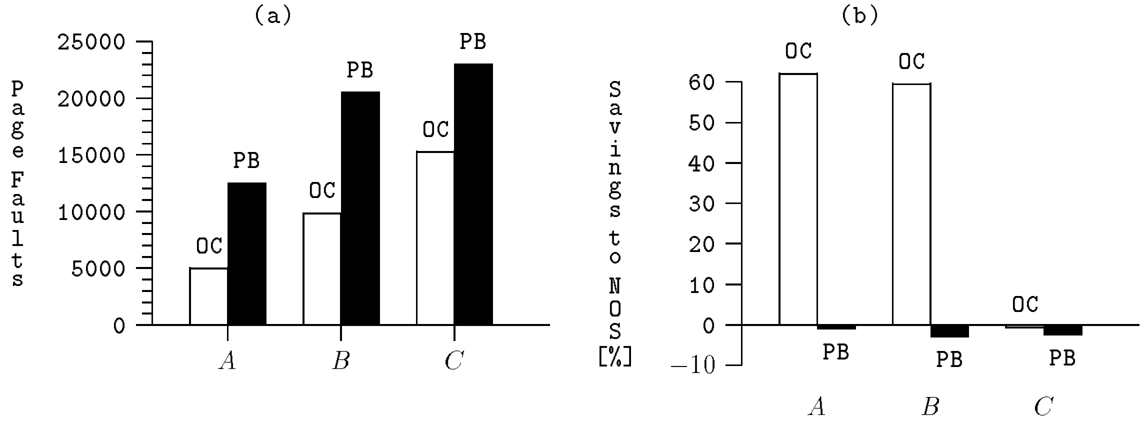


Figure 17: Measuring hot *Traversals* in an Object Cache (OC) and in a Page Buffer (PB)

architecture that buffers up to 800 pages (*PB*) without copying objects. Furthermore, to demonstrate how the benchmark scales against the size of the objects and the number of objects, the *Traversals* were run against three different object base configurations:

Configuration	*Number Parts*	*Number Conns.*	*Objects per Page*	*Size of the DB*
A	20000	60000	100	8.9 MB
B	100000	300000	100	36.3 MB
C	20000	60000	9	42.5 MB

Fig. 17a illustrates how, due to object caching, the number of page faults can be decreased significantly. For example, for the object base configuration *A*, fewer than half the faults occurred under the *OC* architecture. Fig. 17b illustrates how this affected the performance of pointer swizzling; the savings attained by the best swizzling technique (in most cases this was application-specific lazy indirect swizzling) compared to no-swizzling are depicted. In configurations *A* and *B*, most of the objects could be cached in the copy architecture and pointer swizzling was up to 60% superior to no-swizzling. On the other hand, the page buffer did not provide enough locality to make swizzling worthwhile.

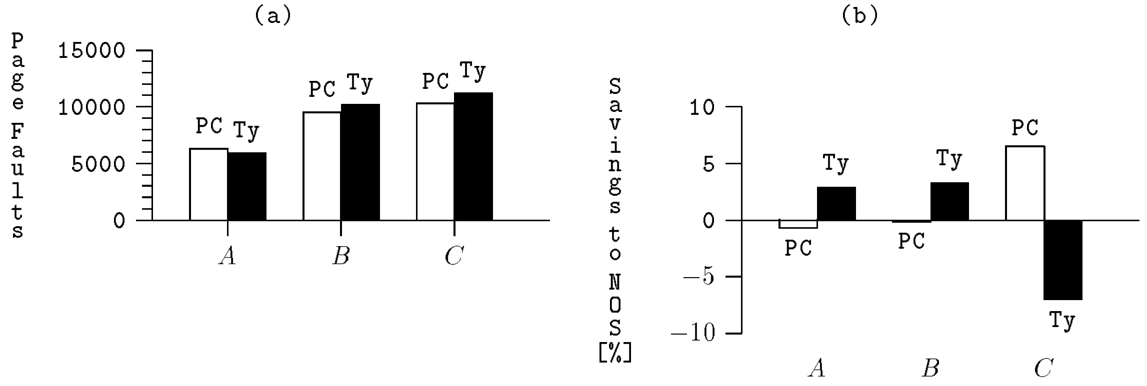


Figure 18: Measuring Cold *Traversals* in a *Part-to-Connection* (PC) and in a *Type-based* (Ty) Clustered Object Base

In configuration C, even the copy architecture could not cache enough objects from the *warm-up* run to make swizzling profitable.

6.6.3 Clustering

The same effect as obtained by object caching can be achieved by clustering. In Fig. 18, *type-based* (*Ty*) clustering (i.e., storing all the *Connections* in a segment and all the *Parts* in a different segment) is contrasted to clustering a *Part* together with the three *Connections* that originate in the *Part* on the same page (referred to by *PC*). The *Traversals* were run cold up to depth 7 on the three object base configurations described above. It became apparent that good clustering can also make the difference between no-swizzling and swizzling.

7 Making the Right Choice

In Section 5, a cost model that permits deciding which swizzling strategy is the most profitable was described. In this section, how, in practice, the variables of the cost model can be derived is outlined, and as a result, the most profitable swizzling strategy can be determined. Some of the ideas were borrowed from *clustering* in object bases [GKKM92b]; there, the same problem of combining characteristics of the object base and the application's profile was addressed.

7.1 Monitoring the Variables of the Cost Model

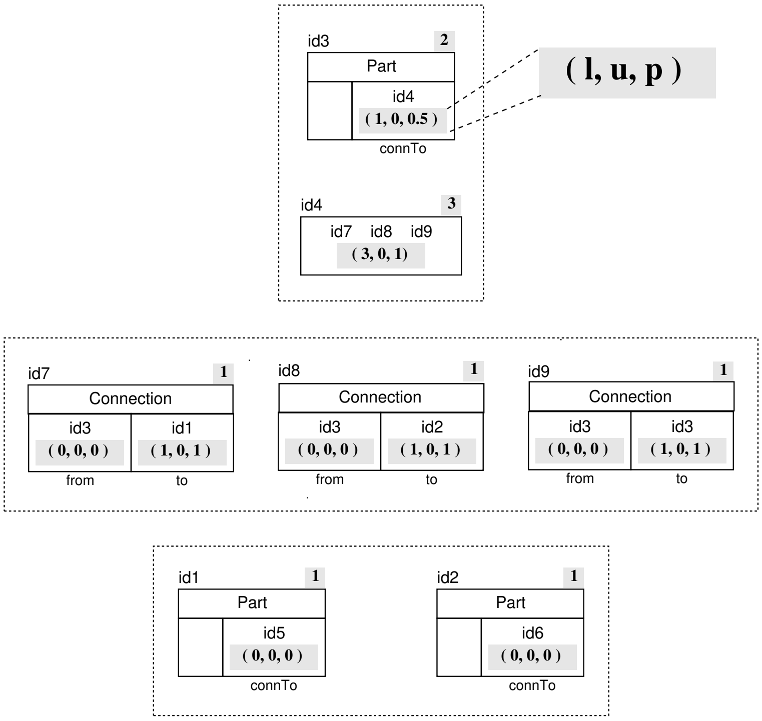
The application's profile can be determined by monitoring. The application is executed in the *training mode* using no-swizzling.[5] Thus, variables o, l, u, and $m(st)$ are computed for every context, i.e., the smallest granularity, by adding up the contribution of every object and reference.

Figure 19 illustrates this approach with the help of a *swizzling graph* (cf. Figure 19b) and the trace of a *Traversal* with depth 1 (cf. Figure 19a). The trace keeps a record of

[5]It is also possible to carry out a trace-driven simulation [DYC93].

```
id3  connTo  r
id4    --    r
id7    to    r
id1  connTo  r
id4    --    r
id8    to    r
id2  connTo  r
id4    --    r
id9    to    r
id3  connTo  r
```

(a)



(b)

Figure 19: Generating a Swizzling Graph (b) from an Object Trace (a)

every call of the object manager; for every access to an object, it records the OID of the object, registers the attribute if the object is tuple-structured, and records whether the object was read (r) or modified (w).

The swizzling graph is an adaptation of the *clustering graph* [TN91]. For every object (represented as a node), the number of times that the object is faulted by the object manager is recorded; i.e., is accessed without being currently registered in the ROT. For example, if two pages can be buffered, no pages are cached beforehand, and an LRU buffer replacement policy is followed; object *id4* is faulted three times and, therefore, assigned weight 3. In Figure 19b, pages are depicted as dotted boxes. To obtain realistic results, the application should be trained in a cache in which other applications were run beforehand.

Every reference (depicted as an edge in the swizzling graph) is assigned 3 weights: l, u, and p. References that are stored in a set are assigned these three weights collectively because in sets individual references cannot be distinguished; u counts the number of times the reference is redirected; i.e., the number of w records in the trace. l sums up the number of times a reference is dereferenced. Here, the number of times is counted that a reference is read and the referenced object is accessed within the next 10 records of the trace, e.g., in Figure 19 the *connTo* field of *Part id3* is dereferenced only once although it is read twice.

To determine the number of references per granule that are swizzled following lazy swizzling upon discovery ($m(lazy)$), p computes for every reference the probability that the reference is read when its "home object" is brought into the buffer pool. When an object is loaded, a flag is allocated for all the fields that contain a reference. If a field is read, the corresponding flag is set. When the object is displaced from the buffer pool, the flags are evaluated recomputing p for every reference.

The cumulative weights of the swizzling graph determine the variables of the cost model. For the example depicted in Figure 19, $o = 10$ and the following values apply:

	l	u	p	$m(lazy)$	$m(eager)$	
from	0	0	0	0	3	*CTX*
to	3	0	1	3	3	
connTo	1	0	1/4	1	4	
$\{Conns\}$	3	0	1	3	3	
Parts	3	0	1/2	3	6	*TYPE*
Conns	3	0	1	3	3	
$\{Conns\}$	1	0	1/4	1	4	
	7	0	7/13	7	13	*APPL*

$\overline{fi}$ is approximated by $m(st)/o$; e.g., 1.5 for *Parts* following eager swizzling. The inaccuracy can be reduced by reference counting when monitoring the application; e.g., executing the application using lazy and eager indirect swizzling and recording the counters kept in the descriptors. For this example, the conclusion was reached that no-swizzling in application-specific mode is the most efficient strategy.

The variables can of course be computed "on-the-fly" when the application is monitored. It is not necessary to store the weights of all the objects and all the references.

Consequently, the storage overhead is $\mathcal{O}(g)$, g representing the number of granules (e.g., attribute identifiers) recorded.

7.2 Reconsidering Eager Direct Swizzling

As stated in Section 3.2, eager direct swizzling can induce a snowball effect, and as a result, cause additional I/O activity. This characteristic was not taken into account in the last section (incremental loading was assumed in any case) and, therefore, for some granules, the decision to swizzle eagerly and directly has to be reconsidered.

To find those granules automatically that may be swizzled eagerly and directly, thus guaranteeing that no additional I/O activity would be induced, the following greedy algorithm is applied:

1. Sort all granules[6] that are supposed to be swizzled eagerly and directly in an *intention_list* in decreasing order according to $C(EDS) - C(LDS)$ for every granule; i.e., placing first the granule that benefits most from eager direct swizzling.

2. Assume for the following that all granules in the *intention_list* are swizzled lazily.

3. Consider the first granule of the *intention_list*. By means of simulation and sampling of the object base, ascertain whether this granule can be swizzled eagerly and directly without causing additional I/O. If so, swizzle this granule eagerly and directly; otherwise, swizzle the granule lazily and directly. Take this decision into account in all further calculations. Repeat this step with the next granule from the *intention_list* and so on until it is empty.

The algorithm works well for type and context-specific swizzling and for the data structures of the OO1 benchmark (cf. Figure 8 on page 15). However, it is restrictive since it precludes eager direct swizzling as soon as the simulation discovers that a snowball effect may occur. Consequently, eager direct swizzling is not always fully exploited. The system should, therefore, also allow the user to specify what granules may be swizzled eagerly and directly. The user can estimate best the risk of this technique, i.e., flooding the memory with the transitive closure for a particular application.

7.3 Alternatives

7.3.1 Optimizing the Overall Throughput

If many short applications are run on a client, it is profitable to optimize the overall throughput rather than optimizing every single application in isolation. Here, it must be kept in mind that pages are buffered *hot* after commit of an application (i.e., all references remain swizzled), and reswizzling should be avoided (see Section 4.1).

In this case, monitoring is not applied to a single application but to a *snapshot* of all the activities on a client over a certain period (e.g., one day). Then, the swizzling decision is carried out as described above, and the object manager in the client is configured accordingly. Any conflicts between two applications are precluded, as all the applications that run on the same client have to follow the same swizzling strategy.

[6]In the application-specific mode, there is only one granule; the algorithm, nevertheless, is valid in the same way.

7.3.2 Decapsulation Based Swizzling

The method of monitoring applications as described in Section 7.1 faces several problems. First, training can be very costly; e.g., training a long design transaction that will run for months. Furthermore, to achieve representative results, the application must be executed several times in the training phase or a long period must be monitored. Second, the state of the object base can change rapidly; as a consequence, the results obtained by training may be outdated very soon.

To overcome these difficulties, the object of current investigation is to discover how *decapsulation*, a tool that carries out program analysis [KSK93], can be applied to making the right swizzling decision. Decapsulation was developed in previous work on application profile dependent optimization [KKM91, KMS92, GKKM92a]. By extracting all the reference chains (path expressions) that are possibly traversed by an application, decapsulation characterizes the profile independently from the state of the object base. In addition, the running time of decapsulation is negligible (usually less than a second).

8 Conclusions

In this work, alternative pointer swizzling techniques for a persistent object manager that facilitates object replacement during an application run were investigated. The qualitative and quantitative analysis indicated that there is no *one* superior pointer swizzling technique. Rather, they all have their distinctive pros and cons depending on the characteristics of the application profile. This led to the development of an adaptable object manager that allows varying the pointer swizzling technique employed according to the particular profile. Four different granularities were devised: application, type, context, and reference-granules were devised within which the pointer swizzling techniques can be varied. Thus, the object manager provides flexible pointer swizzling along two dimensions: (1) it uses four swizzling techniques (lazy direct/indirect, eager direct/indirect) and no-swizzling and (2) it allows adjusting the particular swizzling technique used within the four above-mentioned granularities.

A technique based on monitoring the application profile in combination with sampling the object base was outlined to determine the most appropriate swizzling technique and adjustment granularity for a given application. In future work in this area, we will concentrate on static program analysis in combination with sampling the current state of the object base. As stated above, we intend to augment a program analysis method called decapsulation.

Acknowledgements

This work was partially supported by the German Research Council (DFG). Axel Kotulla helped to implement the swizzling strategies. Michael Steinbrunn and Andreas Zachmann participated in the design of the classification scheme. We thank Judith Kossmann for improving the presentation.

References

[ACCM83] M. P. Atkinson, K. J. Chisholm, P. Cockshott, and R. Marshall. Algorithms for a persistent heap. *Software—Practice and Experience*, 13:259–271, 1983.

[BBB+88] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. The design and implementation of O_2, an object-oriented database system. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems*, pages 1–22, Berlin, Heidelberg, New York, London, Paris, Tokyo, Sep 1988. Lecture Notes in Computer Science No.334, Springer-Verlag. Proc. of the Second Intl. Workshop on Object-Oriented Database Systems.

[BGW92] P. Buhr, A. Goel, and A. Wai. μdatabase: A toolkit for constructing memory mapped databases. In *Proc. Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[BKK+92] K. Bruns, C. Kilger, D. Kossmann, G. Moerkotte, H.-D. Walter, and A. Zachmann. Objekte in multiplen Repräsentationen. Workshop on OODBMS of the German "Gesellschaft für Informatik" (GI), Aug 92.

[CAC+84] W. P. Cockshot, M. P. Atkinson, K. J. Chisholm, P. J. Bailey, and R. Morrison. Persistent object management system. *Software—Practice and Experience*, 14:49–71, 1984.

[CDRS86] M. Carey, D. DeWitt, J. Richardson, and E. Shekita. Object and file management in the EXODUS extensible database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 91–100, Kyoto, Japan, Aug 86.

[Com79] D. Comer. The ubiquitous B-tree. *ACM Computing Surveys*, 11(2):121–137, 1979.

[CS92] R. Cattell and J. Skeen. Object operations benchmark. *ACM Trans. on Database Systems*, 17:1–31, March 1992.

[DFMV90] D. J. DeWitt, P. Futtersack, D. Maier, and F. Velez. A study of three alternative workstation server architectures for object-oriented database systems. In D. McLeod, R. Sacks-Davis, and H. Schek, editors, *16th International Conference on Very Large Data Bases*, pages 107–121, Aug 1990.

[DYC93] A. Dan, P. Yu, and J.-Y. Chung. Database access characterization for buffer hit prediction. In *Proc. IEEE Conf. on Data Engineering*, pages 134–143, Vienna, April 1993.

[FNPS79] R. Fagin, J. Nievergelt, J. Pippenger, and H. Strong. Extensible hashing—a fast access method for dynamic files. *ACM Trans. on Database Systems*, 4(3):315–344, 1979.

[GKKM92a] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Clustering in object bases. Technical Report 6/92, Fakultät für Informatik, Universität Karlsruhe, D-7500 Karlsruhe, Jun 1992.

[GKKM92b] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. Technical Report 92-34, RWTH Aachen, D-5100 Aachen, Dec 1992.

[KBC+88] W. Kim, N. Ballou, H. T. Chou, J. F. Garza, D. Woelk, and J. Banerjee. Integrating an object-oriented programming system with a database system. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 142–152, San Diego, Ca., Sep. 1988.

[KC86] S. N. Khoshafian and G. P. Copeland. Object identity. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 408–416, Nov 1986.

[KK83] T. Kaehler and G. Krasner. LOOM—large object-oriented memory for Smalltalk-80 systems. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*. Addison Wesley, Reading, MA, 1983.

[KK93] A. Kemper and D. Kossmann. Adaptable pointer swizzling strategies in object bases. In *Proc. IEEE Conf. on Data Engineering*, pages 155–162, Vienna, April 1993.

[KKM+90] A. Kemper, C. Kilger, G. Moerkotte, H.-D. Walter, and A. Zachmann. Objektorientierte Programmierung und Datenmodellierung in GOM—Das GOM-Handbuch. Interner Bericht, Nov 1990.

[KKM91] A. Kemper, C. Kilger, and G. Moerkotte. Materialization of functions in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 258–268, Denver, USA, May 91.

[KM92] A. Kemper and G. Moerkotte. Access Support Relations: an indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

[KMS92] A. Kemper, G. Moerkotte, and M. Steinbrunn. Optimizing Boolean expressions in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 79–90, Vancouver, B.C., Canada, Aug 1992.

[KMWZ91] A. Kemper, G. Moerkotte, H.-D. Walter, and A. Zachmann. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. der GI-Fachtagung Datenbanken in Büro, Technik und Wissenschaft (BTW)*, pages 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik-Fachberichte Nr. 270.

[Kos91] D. Kossmann. Entwurf und Implementierung von Laufzeitoptimierungsmaßnahmen im GOM-Prototyp. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, D-7500 Karlsruhe, Okt 1991.

[Kot92] A. Kotulla. Neuentwurf und Leistungsbewertung des Laufzeitsystems im objektorientierten Datenbanksystem GOM. Master's thesis, RWTH Aachen, Lehrstuhl für Informatik III, W-5100 Aachen, Dec 1992.

[KSK93] D. Kossmann, M. Steinbrunn, and A. Kemper. Decapsulation: Estimating execution profiles in object bases. Technical report, RWTH Aachen, D-5100 Aachen, 1993. (forthcoming).

[Lar88] P.-Å Larson. Linear hashing with separators—a dynamic hashing scheme achieving one-access retrieval. *ACM Trans. on Database Systems*, 13(3):366–388, September 1988.

[LLOW91] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The ObjectStore database system. *Communications of the ACM*, 34(10):50–63, 1991.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, August 1992.

[MR79] E. Morel and C. Renvoise. Global optimization by suppression of partial redundancies. *Communications of the ACM*, 22(2):96–103, February 1979.

[MS87] D. Maier and J. Stein. Development and implementation of an object-oriented DBMS. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 355–392, Cambridge, MA, 1987. MIT Press.

[MS88] J. E. B. Moss and S. Sinofsky. Managing persistent data with Mneme: Designing a reliable shared object interface. In K. R. Dittrich, editor, *Advances in Object-Oriented Database Systems: 2nd Int. Workshop on Object-Oriented Database Systems*, pages 298–316, Bad Münster, September 1988. Lecture Notes in Computer Science 334, Springer-Verlag.

[NSS89] E. Nemeth, G. Snyder, and S. Seebass. *Unix System Administration Handbook*. Prentice Hall Software Series. Prentice Hall, Englewood Cliffs, NJ 07632, 1989.

[RD91] N. Roussopoulos and A. Delis. Modern client-server DMBS architectures. *ACM SIGMOD Record*, 20(3):52–61, 1991.

[SCD90] D. Schuh, M. Carey, and D. DeWitt. Persistence in E revisited—implementation experiences. In *Proc. of the Fourth International Workshop on Persistent Object Systems*, pages 345–359. Morgan Kaufmann, September 1990.

[TN91] M. M. Tsangaris and J. F. Naughton. A stochastic approach for clustering in object bases. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 12–21, Denver, CO, May 1991.

[VBD89] F. Velez, G. Bernard, and V. Darnis. The O_2 object manager: an overview. In *Proceedings of the Fifteenth International Conference on Very Large Data Bases*, pages 357–366, Aug 1989.

[WD92] S. J. White and D. J. DeWitt. A performance study of alternative object faulting and pointer swizzling strategies. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 419–431, 1992.

[Wil90] P. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. Technical Report UIC-EECS-90-6, University of Illinois at Chicago, Chicago, Illinois 60680, Dec 1990.

[Win93] M. Winslett. Architecture and performance for object-oriented DBMSes. Tutorial handouts for the Data Engineering Conference, 1993.