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 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.

Contents

1 Introduction **3**

2 Architectures of Object Management Systems **4**

3 Pointer Swizzling Techniques **5**
3.1 The Pointer Swizzling Method 5
3.2 Classification of the Techniques 6
3.3 Exploiting Virtual Memory Facilities 9
3.4 Swizzling in Large Objects 13
3.5 Catalogue of Existing Systems 14

4 Adaptable Pointer Swizzling **14**
4.1 Architecture of Adaptable Pointer Swizzling 15
4.2 Granularities of Pointer Swizzling 17

5 Performance Analysis **20**
5.1 A Cost Model for Application-Specific Swizzling 21
5.2 Cost Models for Type and Context-Specific Swizzling 23
5.3 Storage Overhead 25

6 Performance Experiments **26**
6.1 The Benchmark Environment 26
6.2 Lookups 28
6.3 Traversals 29
6.4 Reverse Traversals 31
6.5 Structural Updates 32
6.6 The Influence of Locality on Pointer Swizzling 33

7 Making the Right Choice **36**
7.1 Monitoring the Variables of the Cost Model 36
7.2 Reconsidering Eager Direct Swizzling 38
7.3 Alternatives 38

8 Conclusions **39**

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 must 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 the 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, KK94]

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

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

Figure 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, Mos90] and O_2 [VBD89].

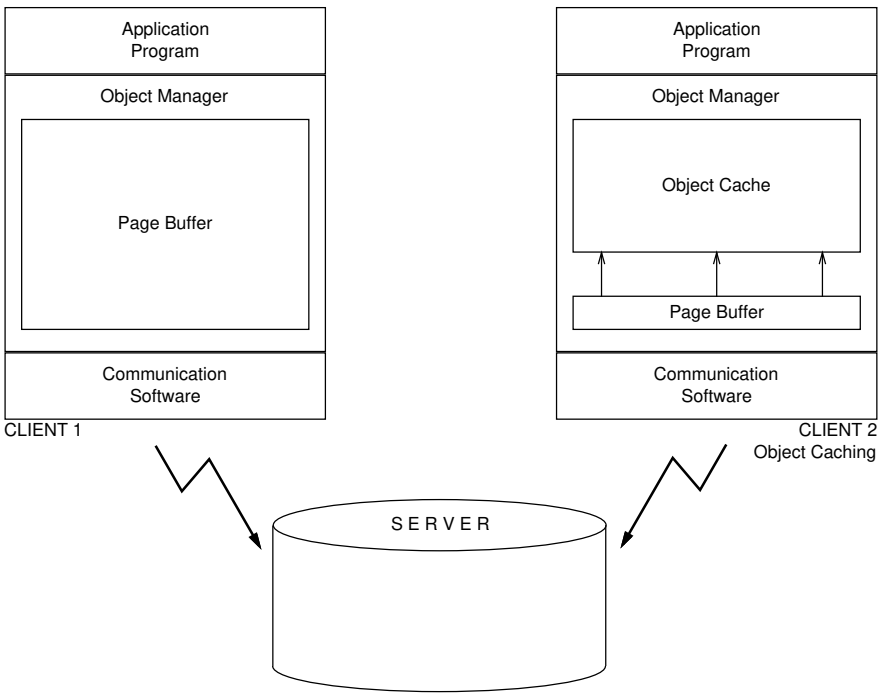


Figure 1: Architecture of an Object Base System

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 described 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 Figure 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, conceptually, allows classifying any technique known to the authors.

3.1 The Pointer Swizzling Method

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.

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.

Address translation can also be carried out on a per page basis. Rather than maintaining an ROT, a page table records the main-memory address of every resident page. A page table is usually much smaller than an ROT since several (small) objects that are accessed in the client are located in the same page. But, page tables can only be used efficiently for physical OIDs that contain the page number and the offset of the object; for logical OIDs, it pays off to maintain an ROT to avoid accessing a persistent object table to determine the page a resident object is located in before in-memory address translation can be carried out. It is even possible to have a logical segment, a collection of pages, as the unit of address mapping if physical OIDs contain the segment number and the offset of the object within the segment. However, the effect of pointer swizzling is basically the same in any architecture. On the one hand, by the means of pointer swizzling, resident objects can be accessed more efficiently since address translation is avoided. 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 the classification scheme). Contrary to this, this work is focussed 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 guarantees that all references are swizzled *before* they are read or dereferenced by an application. Eager swizzling, therefore, is always carried out at fault time.

In GOM, the object is the unit of address mapping. An object fault occurs when an object is accessed that is not registered in the ROT; it should be noted that the object, however, can be main-memory resident since the page in which the object is located can be resident. When an object fault occurs, eager swizzling "scans through" the object and swizzles all the references it finds immediately.

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

Eager swizzling can also be carried out at the granularity of pages, segments and even the whole object base. Accordingly, pagewise eager swizzling scans through a page and swizzles all the references at page-fault time. Segmentwise eager swizzling scans through a segment and swizzles all the references after the segment has been loaded. Some persistent stores that are tailored for specific applications, e.g., a CAD workbench, and that do not offer the full functionality of a database system, load the whole object base and swizzle all pointers at the beginning before any other operation is executed.

Hosking and Moss argue that eagerness at the granularity of pages and segments pays off if spatial locality is high because intra-page and intra-segment references can be swizzled particularly efficiently in their schemes [HM93]. Their schemes, however, preclude replacement in the buffer pool. In addition, such effects cannot be exploited in an architecture that supports object caching. Thus, eager swizzling is carried out at the granularity of objects in this study to avoid swizzling references located in pages and objects that are never accessed.

Even at the fine granularity of objects, an eager approach swizzles references unnecessarily. To minimize unnecessary swizzling of references, *lazy* swizzling only swizzles on demand. No reference is swizzled that is not read and, therefore, cannot be dereferenced by the application. In fact, lazy swizzling carries out pointer swizzling at the finest granularity conceivable: the reference. On the negative side, however, lazy swizzling must handle two different representations of references at running time: swizzled and non-swizzled references. Eager swizzling guarantees that all the references in the resident objects that are registered in the ROT are swizzled. For lazy swizzling, software checks that determine the state of a reference (swizzled or not swizzled) must 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*; for example, 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 several (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

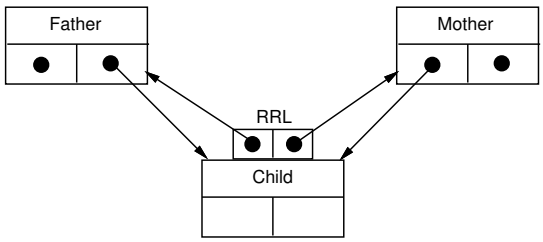


Figure 2: Direct Swizzling and an RRL

are often copied into local variables before they are dereferenced. A great deal of potential is lost, therefore, when using lazy swizzling upon dereference.

3.2.2 Direct, Indirect, and Optimistic Swizzling

Depending on whether it is possible to swizzle a reference which refers to an object that is not main-memory resident, a difference is made between *direct* and *indirect* swizzling. 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)*. Figure 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, these references (i.e., their "home objects") must be displaced, too. This can result in a "snowball" effect. 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. In Section 3.3, a method to implement eager direct swizzling that avoids this snowball effect is discussed: rather than preloading or displacing objects, the corresponding page frames are access-protected. However, the snowball effect can also be controlled by adaptable pointer swizzling as described in Section 4. In any case, preloading can be a desired effect to overlap computation overhead for swizzling with I/O as observed by Moss [Mos92].

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 refer to that object. Assuming that an attribute of an object is assigned a new value, first of all, the RRL of the object that was referenced by the old value of the attribute must be updated. Subsequently, the attribute must 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.

Another way to implement direct swizzling without setting up RRLs was recently devised [MS94]. A hash table, the so-called *swizzle table*, is maintained with a maximum number of entries. In every entry, a single directly swizzled reference is recorded. Thus,

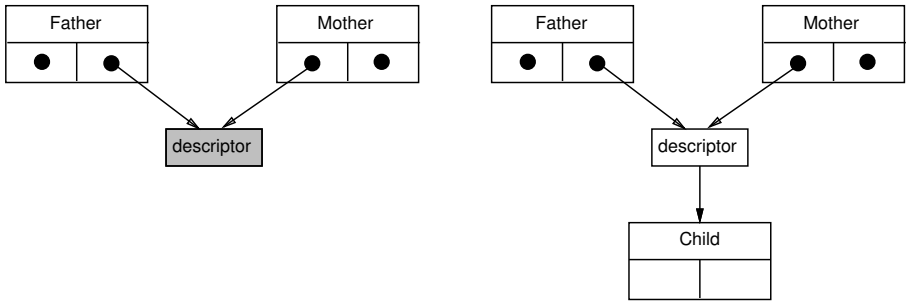


Figure 3: The Scenario of Indirect Swizzling

the total number of references that can be directly swizzled at the same time is restricted by the maximum number of entries in the swizzle table. If an object is evicted from the buffer pool, the swizzle table is inspected, and all the directly swizzled references referring to the object are unswizzled. In this approach, however, it is not clear how the maximum number of entries in the swizzle table can be determined. In any case, simulation results indicate that this way of implementing direct swizzling is not very attractive even given an optimum choice for the size of the swizzle table [MS94].

Indirect swizzling totally avoids the overhead of recording reverse references by permitting the swizzling of references that refer to 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. Figure 3 illustrates this scenario; the shading marks the left-hand descriptor as invalid.

To reclaim unused descriptors, 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, it is guaranteed that every object referred to by a swizzled reference is resident.

In some applications, it is sufficient to consider only *load-work-save* or *create-work-save* sessions. Such applications access few objects that do not exhaust main memory and swap space, and, thus, object replacement can be precluded. If object replacement is precluded, it is possible to swizzle references directly without maintaining RRLs. We refer to such a technique as *optimistic* swizzling. Optimistic swizzling can be found in PS-algol [ACCM83, CAC+84], Moss' experimental platform [Mos92], and the run-time system of E (EPVM 2.0) [WD92].

3.3 Exploiting Virtual Memory Facilities

As stated above, a possible way to implement a persistent store is to exploit the virtual memory facilities supported by hardware. In this section, this "hardware" approach is discussed, and explained why the implementation of pointer swizzling using "software"

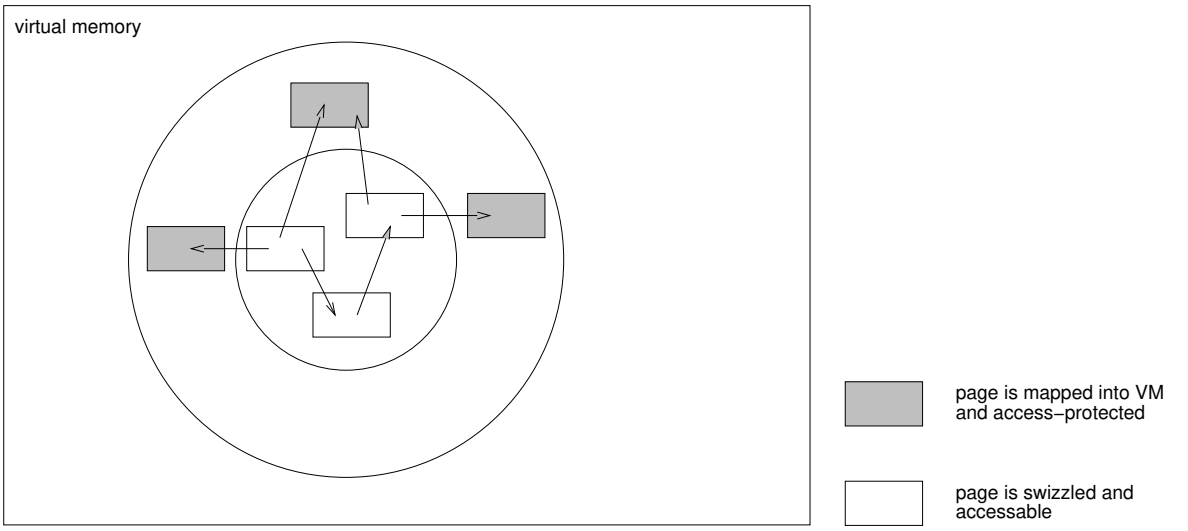


Figure 4: Wave Front of Swizzled and Mapped Pages in the Texas Persistent Store

only was chosen.

In the hardware approach, all the references in memory are implemented as virtual addresses. Thus, pointers to persistent objects are dereferenced like pointers to transient data; i.e., no interpretation overhead is necessary for residency checks or to determine the state of a reference. Like object descriptors for indirect swizzling, virtual memory allows keeping direct pointers, swizzled to virtual memory addresses, to objects located in pages that are not resident in physical main memory. Virtual memory permits getting access protections to pages which can be exploited in the following way: when a reference referring to a non-resident object is dereferenced, an exception is signaled, and the persistent object system reads the corresponding page into the main-memory buffer pool.

Wilson and Kakkad describe the swizzling of pointers and the mapping of pages in virtual memory as a "wave front" [Wil91, WK92]. At the beginning, all the pages that are referred to by an entry pointer are mapped into virtual memory and access-protected. These pages are not loaded nor is any physical main memory allocated. Only when a page is accessed for the first time, it is loaded and the access protection is released. This is achieved by a trap to the object base system which is rather expensive. In their Texas persistent store [SKW92], at the same time (at page-fault time), all pointers located in the page are swizzled. In Texas, pointers are stored as physical OIDs in persistent memory, i.e., they contain the page number and the offset of the object they refer to. A persistent pointer referring to an object located in a page that is already located in virtual memory is translated into a virtual address by consulting Texas' page table that records the mapping of pages to their virtual memory addresses. If the page is not registered in the page table (i.e., no pointers referring to objects located in that page have been encountered before), the page is mapped into virtual memory and access-protected first. Figure 4 illustrates this principle. When a page is accessed for the first time, it is swizzled (i.e., all the pointers in the page are swizzled) and its access protection is released thereby moving the *inner* wave front ahead. At the same time, when pointers are swizzled, the *outer* wave front is moved ahead to map new pages into virtual memory.

ObjectStore, a commercial object-oriented database system, apparently uses a virtual memory mapping approach in a similar way [LLOW91]. Deviating from Texas, the unit of address mapping is the segment, a collection of pages, rather than an individual page. A persistent pointer contains the segment number and the offset of the object within the segment. When a segment is accessed for the first time, the whole segment, i.e., all the pages of the segment, is mapped into virtual memory. ObjectStore thus reduces some of the computational overhead involved in mapping pages into virtual memory since mapping a whole segment once is cheaper than mapping every page individually. On the other side, more virtual memory is reserved by segments or parts of segments that are never accessed. Pages, however, are also loaded and swizzled incremently by ObjectStore in a client/page-server architecture.

In the current implementation of Texas, all pointers are translated at page-fault time from their persistent representation to virtual addresses. ObjectStore as well as QuickStore, a persistent store recently devised by White and DeWitt [WD94], try to avoid this overhead by storing swizzled pointers persistently in secondary storage. These systems try to map a page (for ObjectStore, a whole segment) every time to the same virtual address and thus pointers that refer to objects located in such pages need not be translated. For large object bases, however, the probability can become quite high that a page cannot be mapped to its predestined virtual memory address since pointers that were previously encountered by different applications and refer to different pages were swizzled to the same address. In this case, these pointers must be translated by ObjectStore and QuickStore, too. In any case, the validity of every pointer located in a page must be checked when the page is accessed for the first time. Only the Cricket database storage system [SZ90] devised by Shekita and Zwilling guarantees that pages are always mapped to the same address in virtual memory. To this end, Cricket restricts the size of an object base to a few gigabytes, the size of 32-bit virtual address space, which usually is not acceptable. The current trend towards 64-bit architectures, however, favors their design.

All the systems discussed above can be termed *eager* since they ascertain that a pointer be swizzled before it is dereferenced. Vaughan and Dearle propagate a hybrid scheme of eager and lazy swizzling; again, exploiting the virtual memory's access protection facility to avoid software checks [VD92]. In the first phase, at page-fault time, all pointers are translated as in Texas. However, pointers to pages that have not been mapped into virtual memory are only *partially* swizzled: rather than mapping the page into virtual memory, a partially swizzled pointer is directed to an entry in a page table that is access-protected. In the second phase, when a partially swizzled pointer is dereferenced, an exception is signaled for referring to the access-protected page table, the page is loaded and mapped into virtual memory, and the pointer (together with others referring to the same page and registered by a chaining mechanism) is fully swizzled, i.e., redirected to the virtual address of the page. This approach to redirect pointers in a second phase is very similar to Hosking and Moss's node marking scheme [HM90, HM93].

Vaughan and Dearle's approach is motivated by virtual address space economy. Virtual address space is only used for pages that are really accessed. Indeed, in a 32-bit architecture, the exhaustion of virtual memory can become a problem for the virtual memory mapping schemes. In Texas, ObjectStore and QuickStore, an application must abort if no more pages can be mapped into virtual memory. Wilson and Kakkad [WK92] devised a mass invalidation algorithm to reuse virtual address space. When virtual address space is exhausted, the mapping of all pages to virtual addresses is given up and

thus, the whole virtual memory is reclaimed. The mapping then incrementally builds up again. Vaughan and Dearle refined this mass invalidation scheme and proposed an incremental invalidation approach in which fully swizzled pointers are converted back to partially swizzled. However, as far as is known none of these algorithms have been implemented and evaluated yet.

In some fields, the "hardware" approach using virtual memory mapping is attractive. Persistent objects are accessed in the same way as transient objects and thus, code initially implemented for transient applications can often be easily reused for persistent applications. Furthermore, pointers are dereferenced as efficiently as directly swizzled pointers in a "software-only" approach without the need to maintain RRLs. The indirection induced by virtual memory addresses must be paid in any case. The use of a "software-only" approach, however, was preferred for the following reasons:

1. As stated in [WD94], none of the systems that currently use memory mapping fully support object identity or "checked references" since the overhead would be prohibitive. Relying on virtual memory pointers, buggy applications can overwrite meta-data and cause severe inconsistencies in the database; for example, schema information mapped into virtual memory and page layouts are possibly overwritten in an uncontrolled way if an application modifies elements of an array out of range. GOM, however, supports "true" object identity using logical OIDs. Logical OIDs are independent of the object's storage position which is very important when the object base is reorganized or when objects migrate from one site to another in a distributed environment. Using logical OIDs, the object is the natural unit of address mapping. Mapping individual objects is not supported by hardware.

2. Using conventional operating systems such as Unix, explicit main-memory buffer management is difficult in memory-mapped systems. It is possible to let the operating system decide what pages to replace when main-memory buffers are scarce; however, this is not always acceptable for database systems. In particular, object caching that can improve buffer utilization dramatically [KK94] is not possible in a memory-mapped system. Furthermore, the exhaustion of virtual memory need not be considered at all in a "software" approach.

3. The cost per page fault is high in a memory-mapped system [WD94]. Exception handling is very expensive, and software checks and indirections are usually the better alternative [HM93]. In addition, (pagewise) eager swizzling does not always pay off: costs are induced to locate all the pointers, and swizzling logical OIDs often requires I/O to access a persistent object table that determines the page the object is located in.

4. Tables that register the mappings of pages to virtual addresses must be maintained in main memory. These tables can become very large since they also record mappings of pages that are not main-memory resident.

In summary, the memory-mapping approach gives up control of important features such as explicit buffer management, location independence and "true" object identity. On the other hand, the "software" approach to implement swizzling "holds on" to the explicit control by the DBMS and thus, gives more flexibility that pays off in a better performance.

3.4 Swizzling in Large Objects

In many applications, large objects are encountered. A large object is defined as an object whose size exceeds the size of a page. However, there are various types of large objects: e.g., a satellite picture (represented as a matrix), the set of all the citizens of Cairo (an index structure) or a very large tuple-structured object. One of the strong points of object-oriented database systems is that they allow well-tuned implementations of any data structure. Of course, in these implementations making use of 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., vectors, 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 vector 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?

3.4.1 Large Object Descriptors

Some objects have the semantics of a large contiguous chunk, e.g., long vectors. In secondary storage, such an object is often implemented as a derivative form of a B-tree [Bil92]. In main memory, however, a different representation is appropriate that allows to reference a part of the large object given the OID and the offset more easily [CDRS86].

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 in a "software" implementation of pointer swizzling. Thus, a descriptor for the large object, called its *directory*, must be maintained. Figure 5 depicts the in-memory implementation of a large list whose size may vary dynamically in GOM [Kot92]. Each time an element of a list is accessed, the directory of the list is consulted. This is where swizzling takes effect. Using indirect swizzling the address of the directory is materialized. On the other hand, under no-swizzling, the ROT must 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.

In a "hardware" implementation (cf. Section 3.3), all pages of a large object are mapped at once into virtual memory to ascertain that the virtual memory frames associated to the large object are contiguous. Thus, address arithmetic can easily be carried out in virtual memory given the base address of the large object and the offset of the chunk to access. Nevertheless, precautions must be taken to implement large objects. In QuickStore, a special descriptor (conceptually, akin to a directory) is maintained for a large object to record which pages of the large object have been accessed. This descriptor, however, need not be consulted if a resident page is re-accessed.

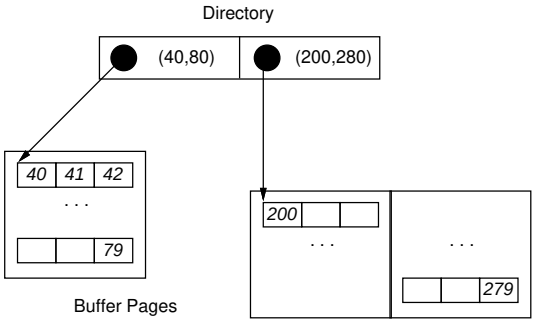


Figure 5: The Directory of a List

3.4.2 Swizzling and Indexes

A reference can 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 sense to swizzle these references since 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.

In Texas, a smart-pointer technique is used to implement indexes. The internal pointers of an index are declared as smart pointers: they are not swizzled, but given the semantics of swizzled references (virtual addresses) by overloading their operators. Edelson gives a good survey on the general use of smart pointers [Ede92].

To summarize, 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.

3.5 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 an implementation of direct swizzling that maintains RRLs which was described in this work, every other swizzling technique can be found there.

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; for example, direct swizzling will be favorable in applications with very high temporal locality, whereas no-swizzling should be preferred in applications with very low locality.

System	*Technique Used*	*Reference*
LOOM	(objectwise) eager indirect	[KK83]
PS-algol	lazy optimistic	[ACCM83, CAC+84]
ORION	(objectwise) eager indirect	[KBC+88]
O_2	no-swizzling	[BBB+88]
EPVM 2.0	lazy optimistic	[WD92]
ObjectStore	(pagewise) eager direct in VM	[LLOW91]
Texas	(pagewise) eager direct in VM	[WK92, SKW92]
GOM	adaptable swizzling	[KK93]

Table 2: Classification of the Swizzling Techniques in Existing Systems

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 to the design of an *adaptable* object manager for the object base system GOM. The adaptable object manager provides the flexibility to choose the most effective swizzling techniques for every individual application. 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 the 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. Figure 6 depicts this scenario schematically.

It is very important that the mapping of the references to the granules be static for each 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*.

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

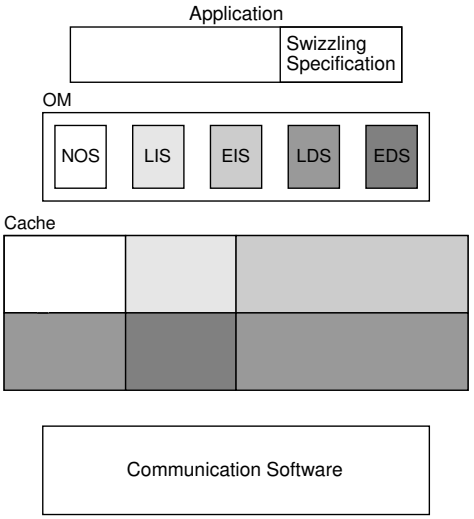


Figure 6: Granularities of Adaptable Pointer Swizzling

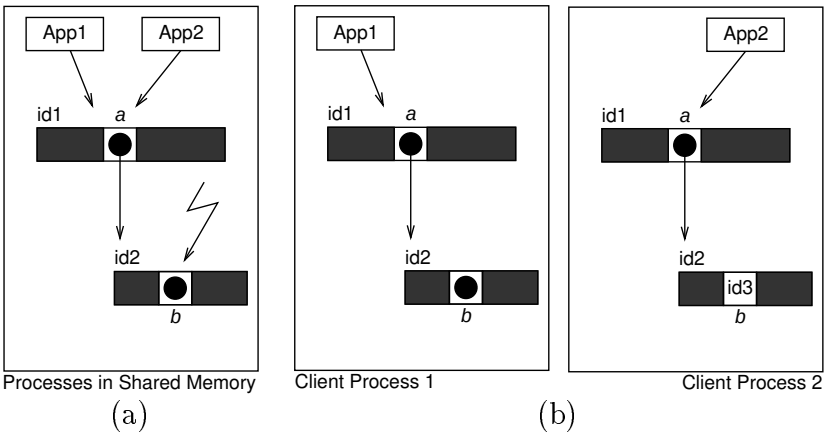


Figure 7: Conflicts due to Adaptable Pointer Swizzling

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. Figure 7 illustrates why, in general, it is impossible for two conflicting applications that run in parallel to 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 possibly have not 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 must be *reswizzled* before they can be dereferenced by application *App2*.

To overcome these difficulties, *lazy reswizzling* approach was proposed, 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* is committed, a flag is set in the entries of the ROT and, 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 *traps* the object and fixes its representation according to the specification of application *App2*. Again, an approach with software checks was preferred to access protections since exception delivery is poor in many operating systems (e.g., SunOS).

Eager direct swizzling can again induce a snowball effect. As soon as an object is accessed, the representation of all the objects that are referred to by this object need to be investigated because the object manager cannot trap 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 (Figure 8).

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.

Figure 9 shows a typical situation from the OO1 benchmark [CS92]. All the references 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

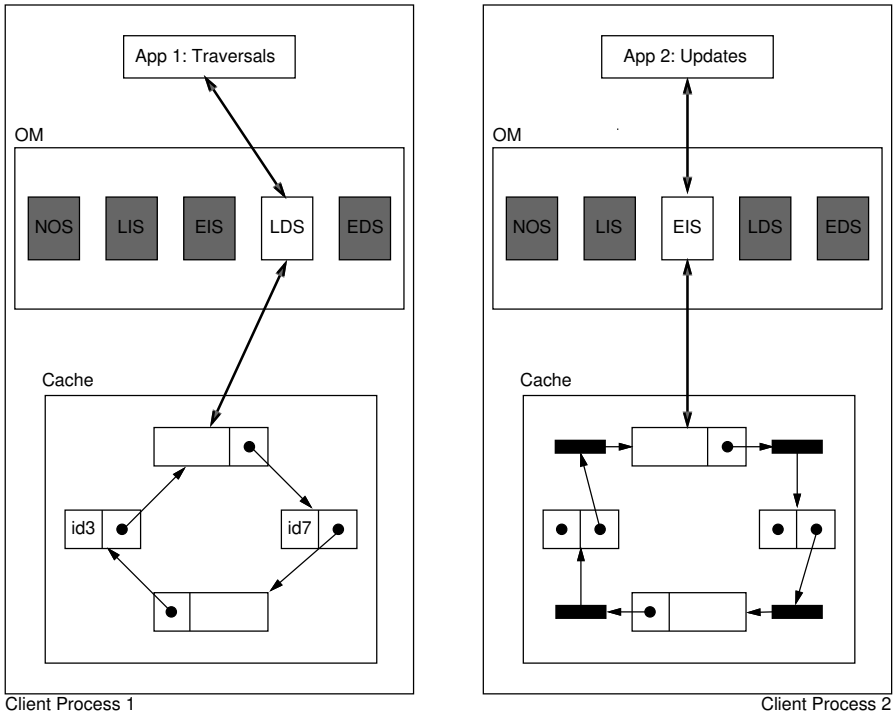


Figure 8: The Scenario of Application-Specific Swizzling

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 must usually 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 procedure for every type. Each time an object is faulted, the corresponding procedure is called using *late binding*, and swizzling is carried out accordingly. This approach is equivalent to the definition of *type descriptors* in Texas [SKW92] to locate pointers for eager swizzling. In the following, the code of the procedures for the example shown in Figure 9 is given.

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

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

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.

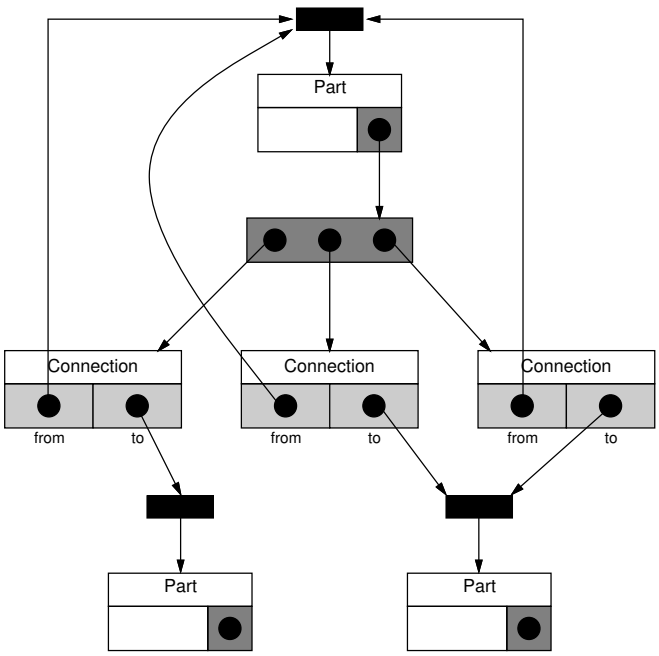


Figure 9: Type-Specific Swizzling in the OO1 Benchmark

In conclusion, type-specific swizzling exploits better the merits of the swizzling techniques than application-specific swizzling. Calling a procedure using late binding every time an object is faulted, 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, Figure 10 shows 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 procedure is generated for every type that is called when an object is faulted and induces the same overhead as for type-specific swizzling. Subsequently, the code is given for the *Parts* and the *Connections* of the example shown in Figure 10. In the procedure *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.

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

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

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

```
myConn.to := myConn.from;
```

If the *to* fields are swizzled directly and the *from* fields are swizzled indirectly, the indirectly swizzled reference of *myConn.from* must be "translated" before it is copied into

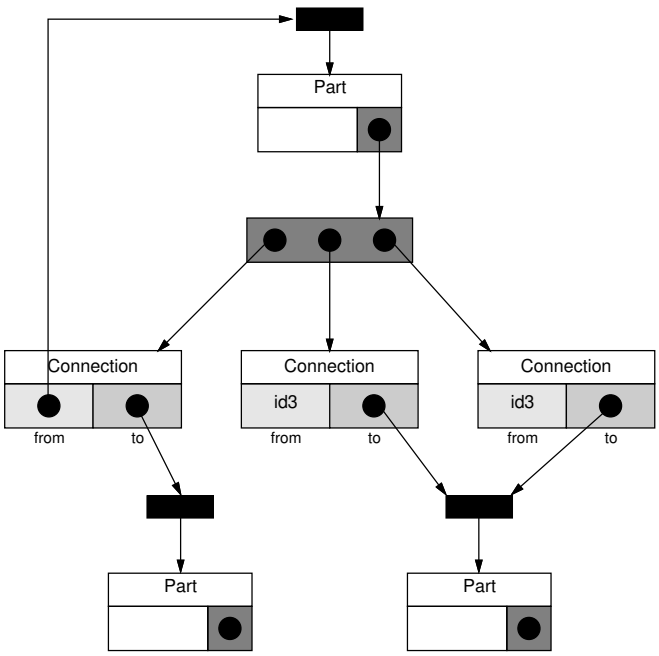


Figure 10: Context-Specific Swizzling in the OO1 Benchmark

myConn.to. Translations must also be carried out to evaluate Boolean expressions such as *myConn.from* = *yourConn.to*. Following application as well as type-specific swizzling, translations are never necessary.

So far, the focus was on inter-object references. A 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* can 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 must be resolved at run-time at high cost.

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

5 Performance Analysis

In this section, a cost model is described for application, type and context-specific swizzling. With the help of this cost model, the following two decisions can be made, provided that

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

the profile of an application and the characteristics of the object base are known exactly: (1) in what granularity to swizzle and (2) which techniques to use. 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.

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

In addition to 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 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 every case.

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

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).[1] It could be 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.

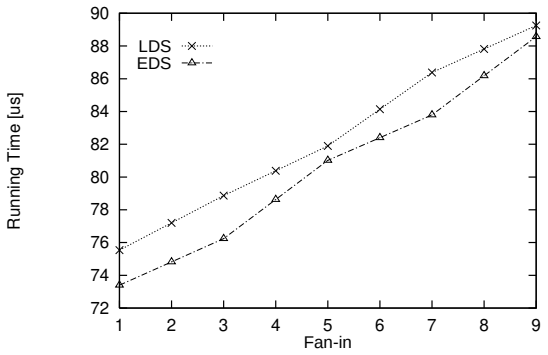
Transient C (called *TC*) was also measured. The tremendous gap between transient C and eager direct swizzling (which is also observed for the pointer swizzling techniques in [WD92]) is due to two factors: in the first place, the object manager was based on the EXODUS storage manager which provides user-descriptors that cause an additional indirection; then, each time an object was accessed, the object manager set a flag to carry out an *LRU* replacement policy. 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 (Figure 11a). On the other hand, the cost for indirect swizzling is constant. The time to modify a field that contains an **int** was also measured (Figure 11b). In this case, direct swizzling outperforms 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 must mark the updated objects. This overhead, which is relatively expensive, is not induced under transient C.

[1] 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 11: 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.

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

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 is not 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 (∞) when almost every reference is swizzled unnecessarily, i.e., without being dereferenced. When comparing eager direct and eager indirect swizzling, the fact that eager direct swizzling can 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 con-

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

sidering the type-specific parameters, $m_t(st_t)$, l_t and u_t, and the type-specific technique, st_t. Two, the cost to call the type-specific procedure 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 procedure using late binding. Any swizzling that is carried out within the type-specific procedure 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 can induce an additional overhead caused by 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; for example, 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.

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

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.

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

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 must be accounted for as well.

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

The amount of additional code is not significant since software checks are not inlined, but coded in the object manager. In addition, the overhead for the *fetch* functions for type and context-specific swizzling is negligible. On the other hand, every RRL entry consumes $SR = 12$ bytes and a descriptor is $SD = 24$ bytes long in the GOM object manager [Kot92, Kos91].

Considering the data structures of the OO1 benchmark [CS92], 43% of the main memory must 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.

If spatial locality is high and page-based buffering is used, the space overhead of RRLs can be reduced significantly by maintaining reverse references pagewise: rather than maintaining precise reverse references (from every object to every swizzled reference), page-based reverse references can be maintained. For example, page B is registered in the RRL of page A if page B contains directly swizzled references referring to objects located in page A; inter-object references within page A need not be recorded at all. If page A is replaced in the main-memory buffer pool, the object manager scans through page B to detect all these references. In addition, the object manager checks the run-time stack to unswizzle all local variables that refer to objects located in A. Thus, the space overhead is reduced at the price of higher computation overhead to locate the swizzled references.

Furthermore, RRLs may be swapped out to improve the main-memory buffer utilization since they are rarely used. On the other hand, descriptors are hot spots because a descriptor is involved every time an indirectly swizzled reference is dereferenced. In addition, no compact representation of descriptors is possible.

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 were 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 referred to 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 the *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 the *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* must 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*. Thus, modifications are carried out, but the state of the object base does not change ultimately.

The creation of new objects was not measured. Although usually the implementation of this operation depends on the swizzling technique used, there is no swizzling specific cost

Lookups[10000]	*Running Time [s]*				
	NOS	*LIS*	*EIS*	*LDS*	*EDS*
1	11.54	9.07	13.37	9.22	33.59
5	14.32	9.97	12.44	10.70	35.65
10	16.56	11.75	11.70	11.20	38.19

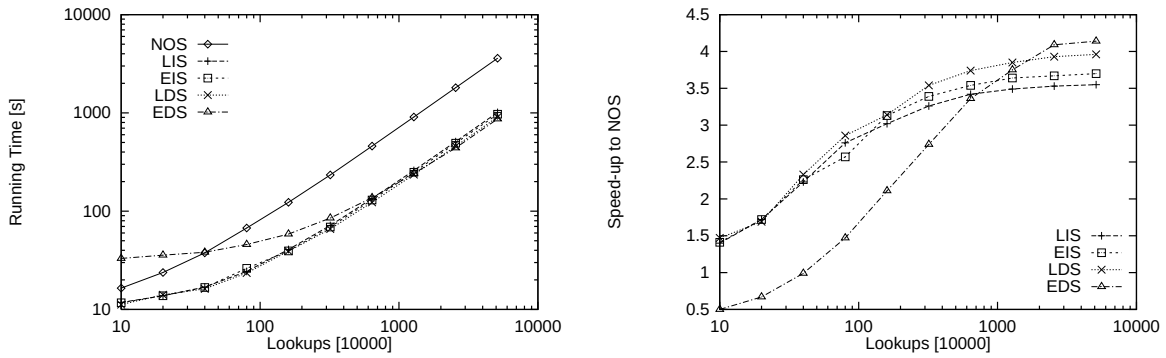


Figure 12: Measuring the *Lookup* Operation

in creating an object. However, newly created objects are usually immediately initialized. This initialization can be seen as a sequence of operations that have the same profile as the *Lookup* and *Update* operations.

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* fitted into the main-memory buffers 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 (Figure 12). We have no explanation why in this experiment eager indirect swizzling performed abnormally when less than 100,000 *Lookups* were carried out (a decreasing response time with an increasing number of *Lookups*).

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 did not make much difference whether references were swizzled or not because the running time of the application was dominated by I/O activity rather than in-memory object lookups. Figure 13a shows the running time of every technique[2] and Figure 13b the savings obtained by the swizzling techniques as compared to no-swizzling.[3] 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 the alternative swizzling techniques. This experiment investigated the behavior of the system when pages were buffered in the client after commit of an application (see Section 4.1); many objects were main-memory resident but they were not in the right representation. Again, the running time (Figure 13c) and the comparison (Figure 13d) between the swizzling techniques and no-swizzling were reported. Any pointer swizzling technique was outperformed by no-swizzling since the objects were not referenced often enough to make swizzling profitable. The losses of the swizzling techniques were rather high as only little I/O activity diluted the results. Here, also context-specific swizzling (referred to as *CTX*) was investigated to demonstrate how large the losses can become due to calls of the *fetch_part* and *fetch_connection* procedures 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 13e and 13f summarize the experimental results for hot *Traversals*. Here, a *Traversal* was executed once using a specific swizzling technique and then, the running time was measured to carry out the same *Traversal* again using the same swizzling technique. Consequently, many objects were main-memory resident and, in addition, in the desired 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 application-specific 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

[2] *EDS* had to be precluded because the size of the object base exceeded the size of the main-memory buffers.

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

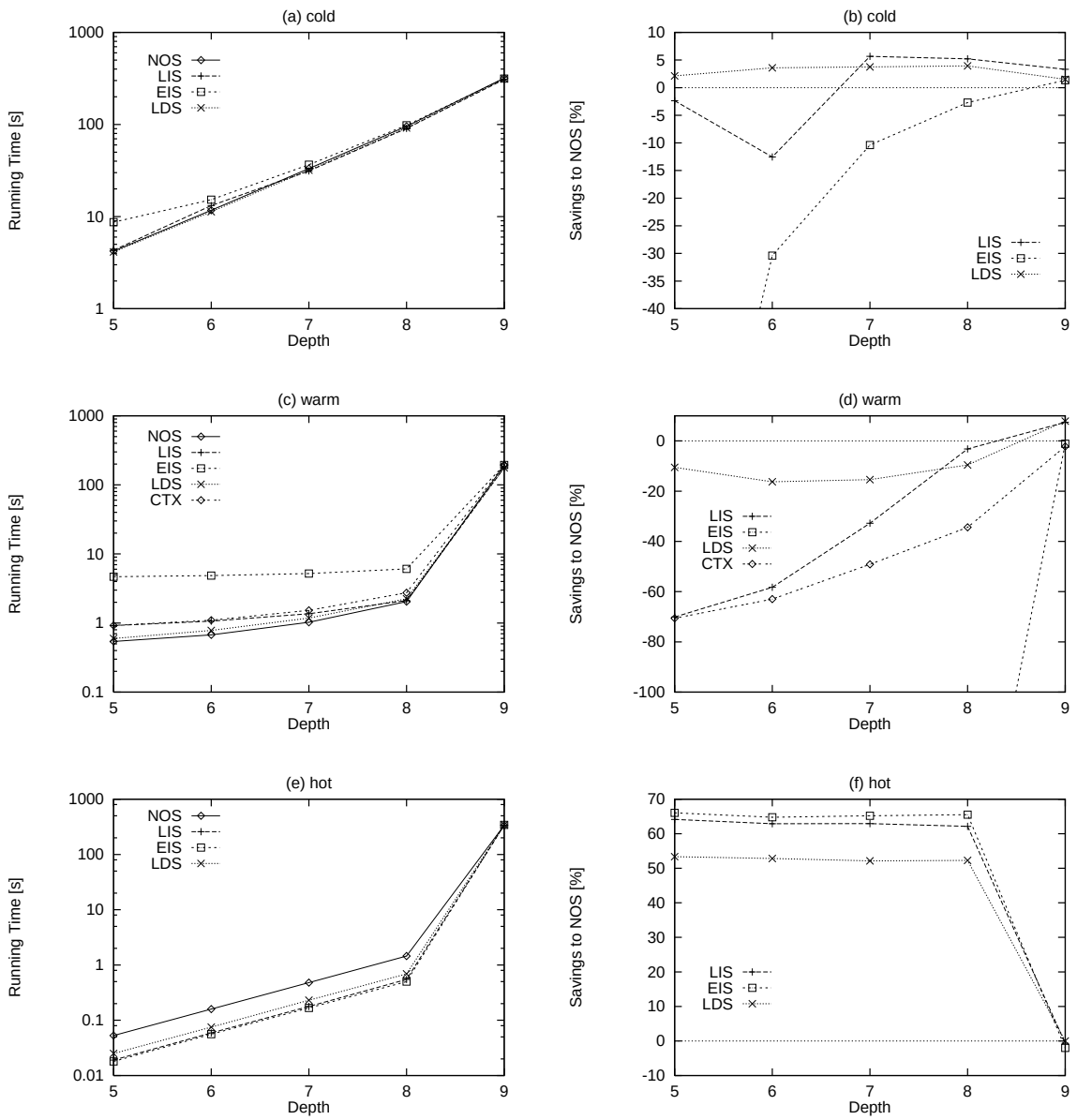


Figure 13: Measuring *Traversals*

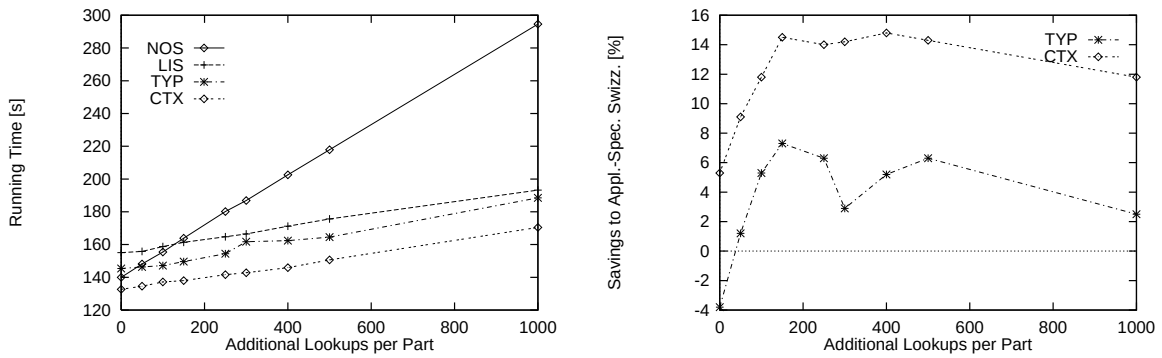


Figure 14: *Traversals* in Combination with *Lookups* (warm)

was "warmed up" by carrying out a *Traversal* using no-swizzling beforehand.

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 and outperformed application-specific swizzling by savings of up to 16% (Figure 14).

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* fit 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. Figure 15 summarizes the experimental results for a 10000 *Parts* and 30000 *Connections* object using a buffer pool of 500 pages. With increasing 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

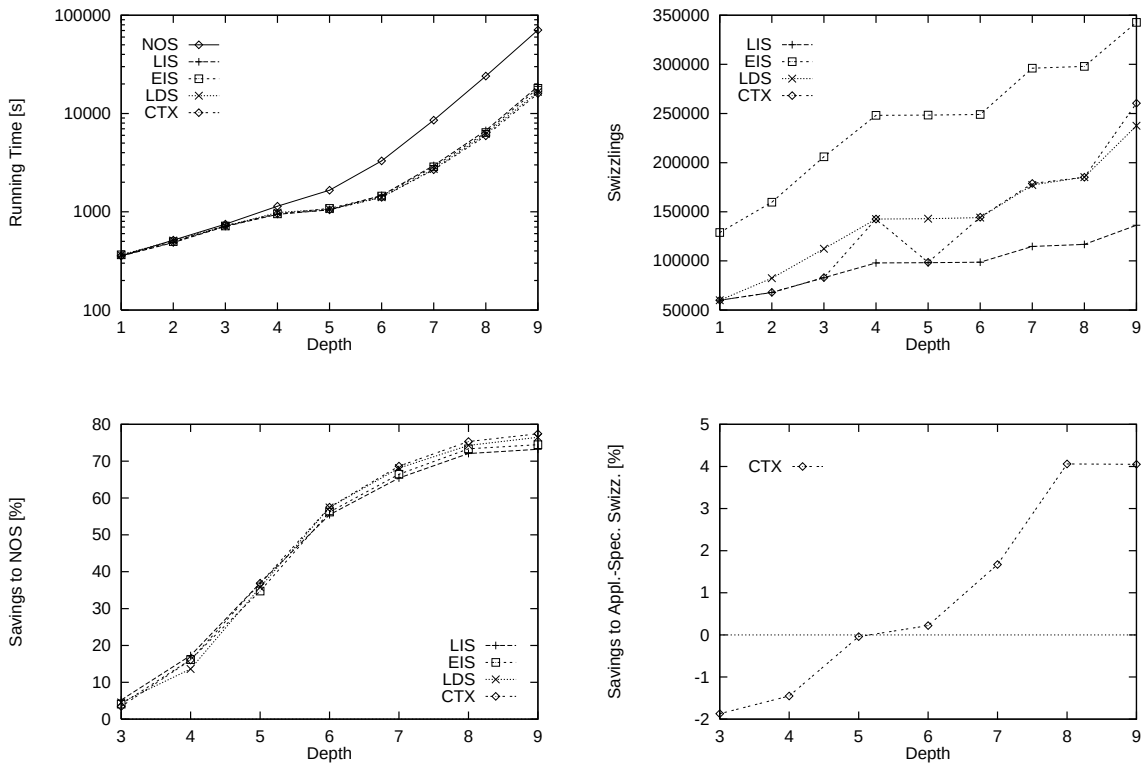


Figure 15: Measuring *Reverse Traversals*

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. Eager indirect swizzling translates many references unnecessarily. However, both techniques can compensate for this by their more efficient object lookup mechanism, and consequently, all swizzling techniques perform equally well.

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

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)

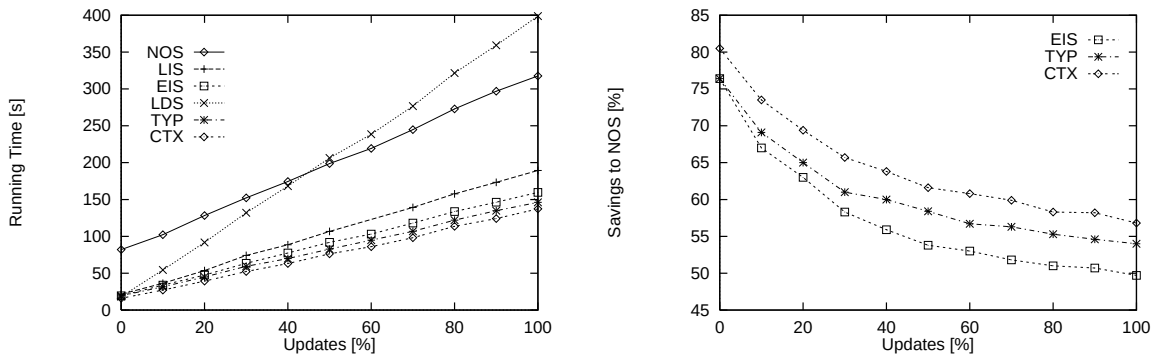


Figure 16: Operation Mix: Updates and Lookups (hot)

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). The running time is reduced by 23% for this operation when using type or context-specific swizzling compared to eager indirect swizzling which is the most efficient technique for application-specific swizzling.

Figure 16 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 (temporal and spatial) *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 21 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%) *Connec-*

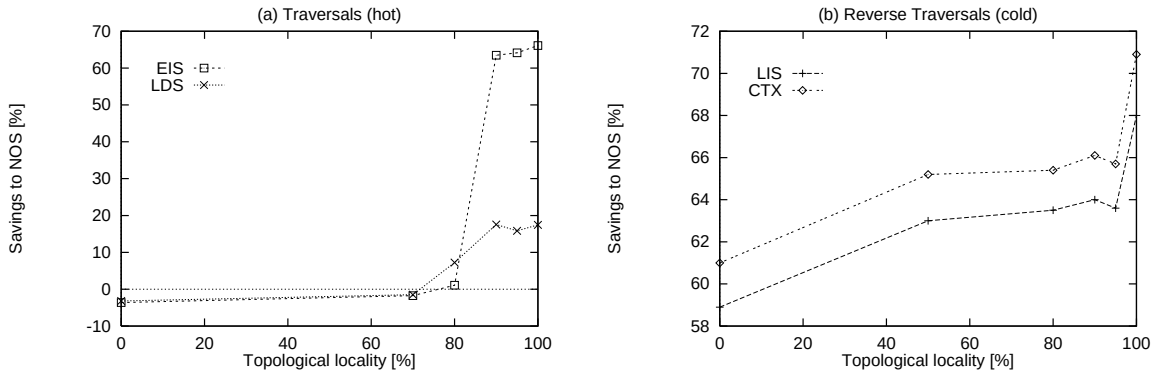


Figure 17: Varying Topological Locality

tions satisfied this demand. This parameter is referred to as the *topological locality* of the OO1 object base.

In Figure 17, 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 architecture that buffers 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

Figure 18a 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

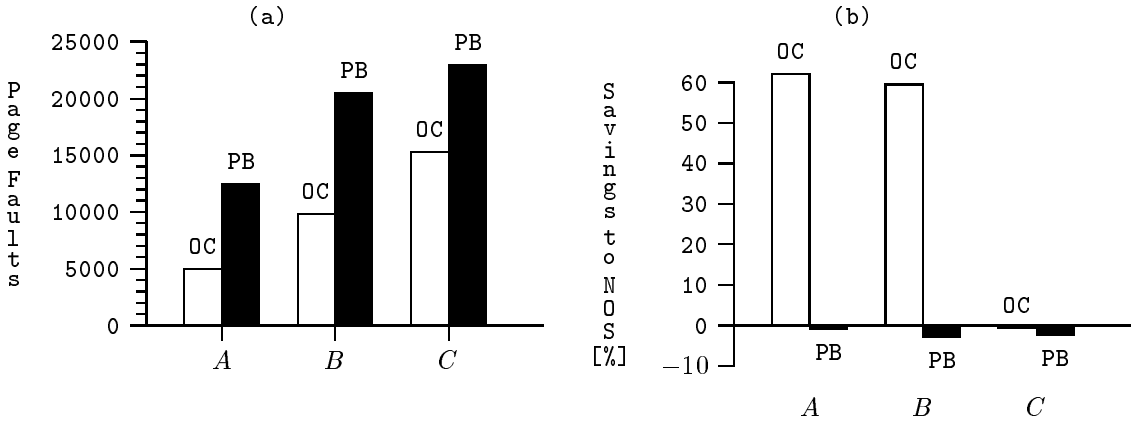


Figure 18: Measuring hot *Traversals* in an Object Cache (OC) and in a Page Buffer (PB)

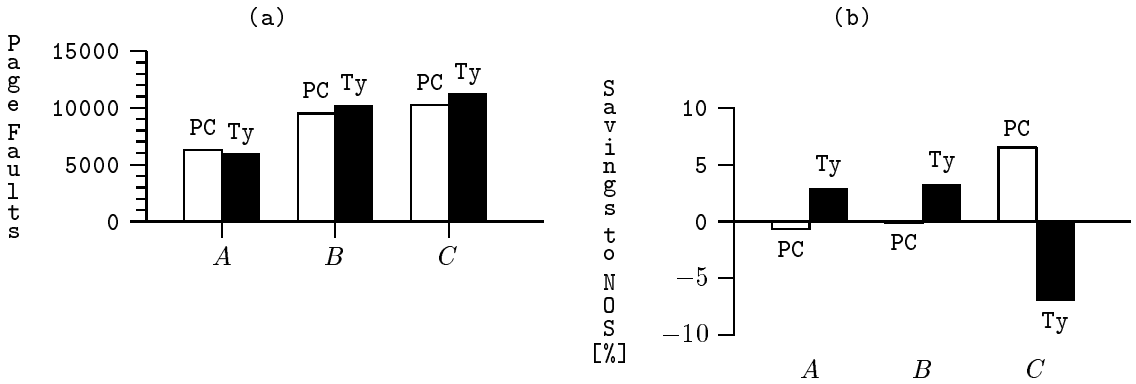


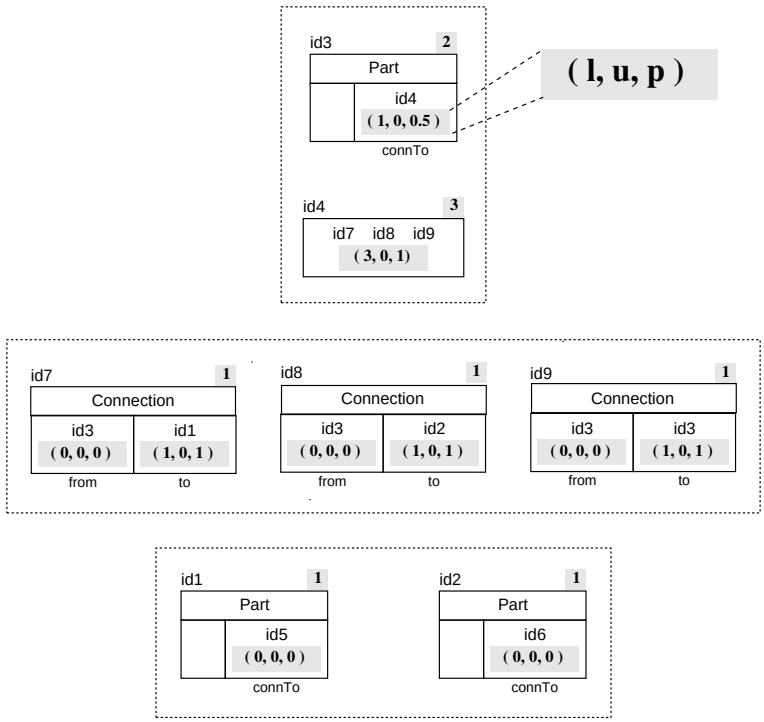
Figure 19: Measuring Cold *Traversals* in a *Part-to-Connection* (PC) and in a *Type-based* (Ty) Clustered Object Base

the faults occurred under the *OC* architecture. Figure 18b 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. 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 Figure 19, *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

```
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 20: Generating a Swizzling Graph (b) from an Object Trace (a)

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.[4] 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 20 illustrates this approach with the help of a *swizzling graph* (Figure 20b) and the trace of a *Traversal* with depth 1 (Figure 20a). The trace keeps a record of every call of the object manager; for every access to an object, it records the OID of the object,

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

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., an object 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 a weight of 3. In Figure 20b, 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, for example, the *connTo* field of *Part id3* is dereferenced only once although it is read twice in the trace shown in Figure 20a.

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 given in Figure 20, $o = 10$ 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; for instance, executing the application in the training mode 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 preceeding section (incremental loading was assumed in any case) and, therefore, for some granules, the decision to swizzle eagerly and directly should be reconsidered.

To find those granules automatically that may be swizzled eagerly and directly without possibly inducing additional I/O activity, the following greedy algorithm is applied:

1. Sort all granules[5] 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 (Figure 9 on page 19). However, it is restrictive since it precludes eager direct swizzling as soon as the simulation discovers that a snowball effect can 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 must follow the same swizzling strategy.

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; for example, training a long design transaction that will

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

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 can be outdated very soon.

To overcome these difficulties, the subject of current investigation is to find out 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 apply 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 Paul Wilson and Dan Weinreb for giving us details of Texas and ObjectStore, and two anonymous referees for their substantial suggestions for the revision of the paper. We are grateful to Judith Kossmann for improving the presentation significantly.

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, September 1988. Lecture Notes in Computer Science No.334, Springer-Verlag. Proc. of the Second Intl. Workshop on Object-Oriented Database Systems.

[Bil92] A. Biliris. The performance of three database storage structures for managing large objects. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 276–285. acmpress, June 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), August 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, August 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 *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 107–121, Brisbane, Australia, August 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.

[Ede92] D. Edelson. Smart pointers: They're smart, but they're not pointers. Technical Report UCSC-CRL-92-27, University of California, Santa Cruz, CA, June 1992.

[FNPS79] R. Fagin, J. Nievergelt, J. Pippenger, and H. Strong. Extendible 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-76050 Karlsruhe, June 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-52056 Aachen, December 1992.

[HM90] A. L. Hosking and J. E. B. Moss. Towards compile-time optimizations for persistence. In *Proc. of the Fourth International Workshop on Persistent Object Systems*, pages 17–27. Morgan Kaufmann, September 1990.

[HM93] A. L. Hosking and J. E. B. Moss. Object fault handling for persistent programming languages: A performance evaluation. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 288–303, October 1993.

[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., September 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, November 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.

[KK94] A. Kemper and D. Kossmann. Dual-buffering strategies in object bases. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pages 427–438, Santiago, Chile, September 1994.

[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, November 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, August 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, March 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-76050 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-52056 Aachen, December 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.

[Mos90] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Trans. Office Inf. Syst.*, 8(2):103–139, April 1990.

[Mos92] J. Eliot B. Moss. Working with persistent objects: To swizzle or not to swizzle. *IEEE Trans. Software Eng.*, 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.

[MS94] M. L. McAuliffe and M. H. Solomon. A trace-based simulation of pointer swizzling techniques. Submitted for publication, 1994.

[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.

[SKW92] V. Singhal, S. Kakkad, and P. Wilson. Texas: An efficient, portable persistent store. In *Proc. Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[SZ90] E. Shekita and M. Zwilling. Cricket: A mapped, persistent object store. In *Proc. of the Fourth International Workshop on Persistent Object Systems*, pages 89–102. 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, August 1989.

[VD92] F. Vaughan and A. Dearle. Supporting large persistent stores using conventional hardware. In *Proc. Workshop on Persistent Object Systems*, San Miniato, Italy, September 1992.

[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, Vancouver, B.C., Canada, August 1992.

[WD94] S. White and D. DeWitt. QuickStore: A high performance mapped object store. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 395–406, Minneapolis, USA, May 1994.

[Wil91] P. Wilson. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *Computer Architecture News*, 19(4), June 1991.

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

[WK92] P. Wilson and S. Kakkad. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. In *Proc. Int. Workshop on Object Orientation in Operating Systems*, pages 364–377, Paris, September 1992. IEEE Press.