VLDB Journal, 4, 519-567 (1995), Malcolm Atkinson, Editor

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

Alfons Kemper and Donald Kossmann

Received June, 1993; revised version received, April, 1994; accepted March, 1995.

Abstract. In this article, different techniques for *"pointer swizzling"* are classified and evaluated for optimizing the access to main-memory resident persistent objects. 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 divided into two classes: (1) those that allow replacement of swizzled objects from the buffer before the end of an application program, and (2) those that rule out the displacement of swizzled objects. 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 are investigated and compared 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 article—demonstrate that there is no *one* superior pointer swizzling strategy for *all* application profiles. Therefore, an adaptable object base run-time system is devised that employs the full range of pointer swizzling strategies, depending on the application profile characteristics that are determined by, for example, monitoring in combination with sampling, user specifications, and/or program analysis.

Key Words. Pointer swizzling, object-oriented database systems, performance evaluation.

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 su-

Alfons Kemper, Ph.D., is a Professor of Computer Science at the University of Passau, D-94030 Passau, Germany, *kemper@db.fmi.uni-passau.de*. Dr. Donald Kossmann, is Research Associate, Department of Computer Science, University of Maryland, College Park, MD 20742, USA, *kossmann@cs.UMD.EDU*.

perior 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 article, 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 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, for example, 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; Schuh et al., 1991). Moss (1992) was the first to abstract from system and/or object model characteristics, and to undertake 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 (1992) 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 article, four alternative pointer swizzling strategies are 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 article—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 the design of an object manager with adaptable pointer swizzling techniques for our experimental object base system GOM (Generic Object Model; for details see Kemper and Moerkotte, 1994). 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 article 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 article is concluded in Section 8.

2. Architectures of Object Management Systems

In this section, the setting of the investigation is described. Besides pointer swizzling, three crucial architectural decisions must be made when designing an object base system (Winslett, 1993):

1. Whether to buffer only pages or to copy objects into an object cache (Kim et al., 1988; Kemper and Kossmann, 1994);
2. Whether to use logical or physical object identifiers (Khoshafian and Copeland, 1986);
3. Whether to use an object, page, or file server (DeWitt et al., 1990).

Figure 1 depicts the client-server architecture (Roussopoulos and Delis, 1991) 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 (Carey et al., 1986), GemStone (Maier and Stein, 1987), ORION (Kim et al., 1988), Mneme (Moss and Sinofsky, 1988; Moss, 1990), and O_2 (Velez et al., 1989).

The benefits of pointer swizzling take effect when a resident object is accessed (i.e., an object is read or modified 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. The techniques can be used equally well with logical and 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 (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

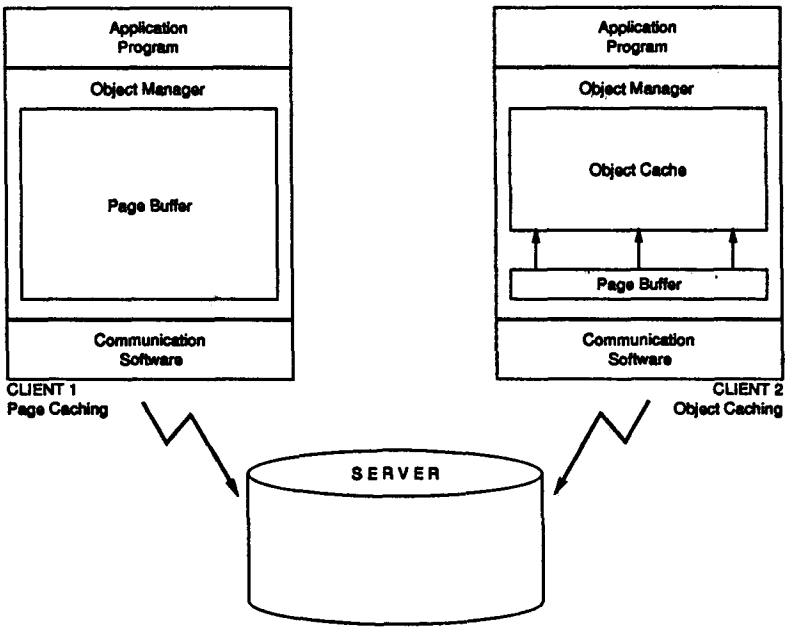
In this section, the approach to pointer swizzling is described. A variety of techniques from the literature and existing systems are discussed and a classification scheme is given that, conceptually, allows the classification of any technique known to the authors.

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 (OIDs; as discussed by Khoshafian and Copeland, 1986). To support large object bases, OIDs are usually at least 64 bits long. In addition, an OID must uniquely identify an object in a client/server system with several machines and several disks so that "flat" virtual memory pointers cannot be used in most systems. On the basis of the OID, the object base system can locate the object in secondary storage and load it into the main-memory buffer allocated to an application.

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.

Figure 1. Architecture of object base system



Each time the object manager accesses a resident object on the basis of its OID, the ROT must be consulted. This is where pointer swizzling takes effect. Pointer swizzling avoids consulting the ROT by materializing the main-memory address of the resident objects and converting (*swizzling*) references into main-memory pointers.

In addition to bypassing the ROT, pointer swizzling can be exploited to *narrow* references (Suzuki et al., 1995) large OIDs are converted into smaller main-memory pointers. As a consequence, the buffer utilization is improved in some systems (e.g., as reported for the LOOM system by Kaehler and Krasner, 1983) and main-memory pointers can be processed in one CPU cycle (Cockshott and Foulk, 1990).

In some systems, address translation is carried out on a per page basis. Rather than maintaining a ROT, a page table records the main-memory address of every resident page. A page table is usually much smaller than a 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 a ROT to avoid accessing a persistent object table to determine in which page a resident object is located 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 means of pointer swizzling, resident objects can be accessed more efficiently, since address translation is avoided. On

Table 1. Classification scheme

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

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

Pointer swizzling has been studied since the end of the Seventies. It was pioneered in the implementation of PS-algol (Atkinson et al., 1983) and in the LOOM project (Kaehler and Krasner, 1983). A large variety of techniques have already been incorporated in existing systems since then. Moss (1992) was the first to undertake a systematic classification and thorough analysis of pointer swizzling techniques. However, Moss only investigated techniques that preclude object replacement (referred to as *optimistic* swizzling in the classification scheme). On the contrary, this work is focused 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 object or page fault time, depending on the unit of address mapping.

In our object base system GOM, which was used as a platform for this study, the object is the unit of address mapping (unless an object is larger than a page, in which case every page of the object is mapped individually; Section 3.4). 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.

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 ones 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.

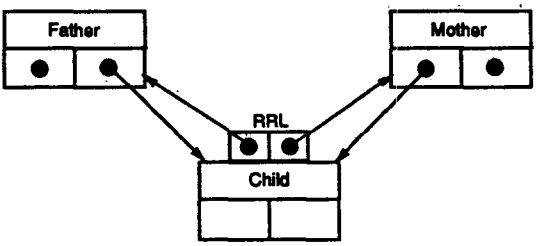
Hosking and Moss (1993) argue that eagerness at the granularity of pages and segments pays off if spatial locality is high, because intra-page and intra-segment references can be swizzled particularly efficiently in their schemes. Their schemes, however, preclude replacement in the buffer pool. In addition, such effects cannot be exploited in an architecture that supports object caching. If the objects are small (i.e., smaller than a page), 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. Swizzling in the context of large objects is discussed in a separate section.

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. References are only swizzled when they are first read by an 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 run 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 or the memory protection hardware must fault all OIDs.

White and DeWitt (1992) distinguished between lazy swizzling *upon discovery* and *upon dereference*. 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 are often copied into local variables before they are dereferenced. A great deal of potential is lost, therefore, when using lazy swizzling upon dereference.

Figure 2. Direct swizzling and an RRL



3.2.2 Direct, Indirect, and Optimistic Swizzling. Depending on whether it is possible to swizzle a reference that 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 also can 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 (Moss, 1992).

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 (Kemper et al., 1994). 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.

Using a garbage collector to improve main-memory buffer utilization also may allow direct swizzling without RRLs: if no more main memory is available, a routine that works like a compacting garbage collector inspects all the main-memory buffers and reclaims the main memory space that is occupied by unreachable objects. In addition, objects that have not been used for a long time could be displaced, and the swizzled references referring to these objects are found while inspecting the buffers with very little additional cost.

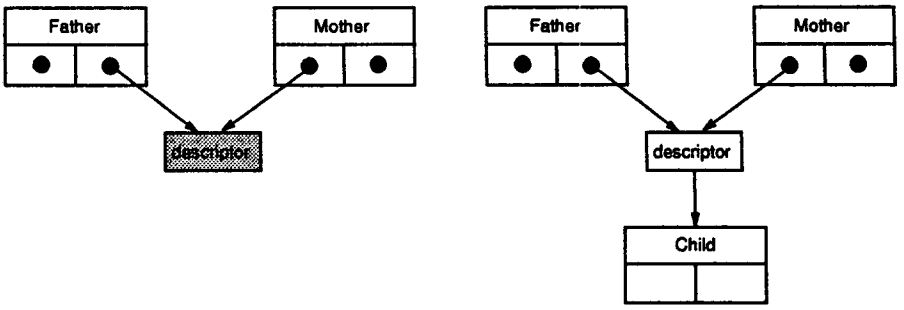
Another way to implement direct swizzling without setting up RRLs was recently devised by McAuliffe and Solomon (1995). A hash table, the so-called *swizzle table*, is maintained with a maximum number of entries. In every entry, a single directly swizzled reference is recorded. Thus, the total number of references that can be swizzled directly 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 (McAuliffe and Solomon, 1995).

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

Figure 3. Scenario of indirect swizzling



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[1] (Atkinson et al., 1983; Cockshott et al., 1984), Moss' experimental platform (Moss, 1992), and the run-time system of E (EPVM 2.0) (White and DeWitt, 1992).

3.3 Exploiting Virtual Memory Facilities

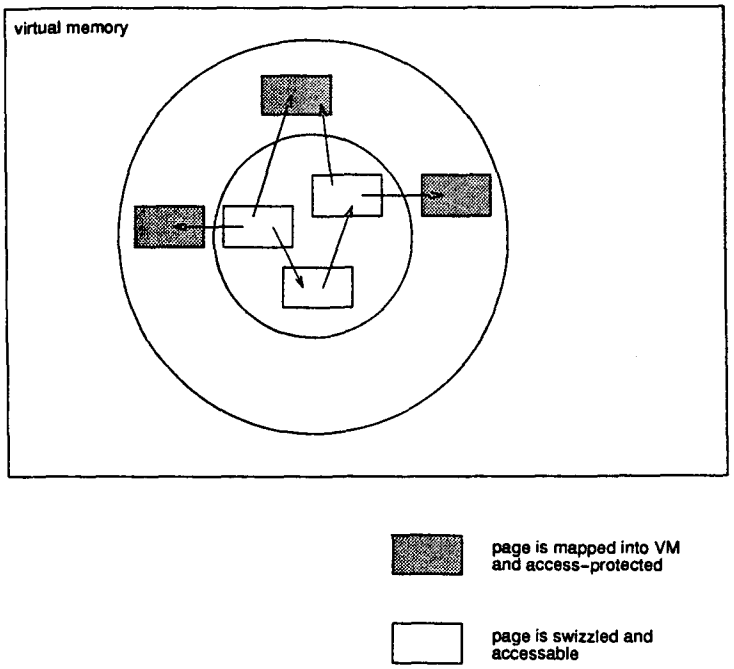
As stated in Section 3.2.2, a possible way to implement a persistent store is to exploit the virtual memory facilities supported by hardware. In this section, this "hardware" approach is discussed, and we explain why the implementation of pointer swizzling using "software" only was chosen.

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 direct pointers to be kept swizzled, and virtual memory addresses referring to objects to be located in pages that are not resident in physical main memory. Virtual memory provides access protection for pages that 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.

The swizzling of pointers and the mapping of pages in virtual memory have been described as a "wave front" (Wilson, 1991; Wilson and Kakkad, 1992). At the

1. Actually, different versions of PS-algol employed different kinds of pointer swizzling; among others the lazy optimistic swizzling method was used.

Figure 4. Wave front of swizzled and mapped pages in Texas persistent store



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, and no physical main memory is allocated. Only when a page is accessed for the first time is it loaded and the access protection is released. This is achieved by a trap to the object base system, which is rather expensive. In the Texas persistent store (Singhal et al., 1993), all pointers located in the page are swizzled simultaneously at page-fault time. In Texas, pointers are stored as physical OIDs in persistent memory (i.e., they contain the page number and the offset of the object to which they refer). 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, which 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 (Lamb et al., 1991). 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 at once is cheaper than mapping every page individually. On the other hand, more virtual memory is reserved by segments or parts of segments that are never accessed. Pages, however, are also loaded and swizzled incrementally 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 (White and DeWitt, 1994), 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 because 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 (Shekita and Zwilling, 1991) guarantees that pages always are mapped to the same address in virtual memory. To this end, Cricket restricts the size of an object base to the size of the virtual address space, a few gigabytes using today's conventional hardware. This is not acceptable. The current trend towards 64-bit architectures, however, favors the design of Cricket.

All the systems discussed in this section can be termed *eager* since they ascertain that a pointer be swizzled before it is dereferenced. Vaughan and Dearle (1993) proposed a hybrid scheme of eager and lazy swizzling; again, exploiting the virtual memory's access protection facility to avoid software checks. 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 the

node marking scheme of Hosking and Moss (1991, 1993).

Vaughan and Dearle's (1993) approach is motivated by virtual address space economy. Virtual address space is used only 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 (1992) devised a mass invalidation algorithm to re-use 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 (1993) 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 often can 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 for in any case. The use of a "software-only" approach, however, is preferred for the following reasons:

1. None of the systems that currently use memory mapping fully support object identity (White and DeWitt, 1994). 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 re-organized, 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, and mapping individual objects is not supported by today's conventional hardware. (The DAIS processor is an initial approach to the design of customized hardware that supports object faulting; Russel et al., 1995.)
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 (Kemper and Kossmann, 1994) 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 often is high in a memory-mapped system (White and DeWitt, 1994). Exception handling is very expensive in many operating systems (e.g., Unix), and software checks and indirections usually are the better alternative (Hosking and Moss, 1993). 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 in which the object is located.

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. The "hardware" approach might become more attractive with special-purpose hardware (e.g., as proposed for the MONADS system; Koch and Rosenberg, 1990) and with more flexible operating systems; for example, the Mach operating system allows the definition of an "external" (user-defined) paging algorithm, as exploited in Eos (Gruber et al., 1992). Nevertheless, memory mapping probably never will provide the same flexibility in the design of a system that can be achieved by a software-only approach to implement pointer swizzling.

3.4 Swizzling in Large Objects

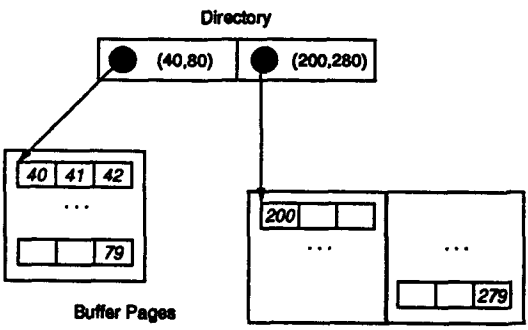
Large objects are encountered in many applications. A large object is defined as an object whose size exceeds the size of a page. However, there are various types of large objects (e.g., a satellite picture, represented as a matrix, the set of all the citizens of Cairo [an index structure], or a very 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 also is desirable. In this section, some of the principal difficulties concerning pointer swizzling in the context of large objects are addressed. An *open* architecture, which integrates the implementations of various type constructors (e.g., vectors, matrices, and sets), and which can easily be extended, has been described (Bruns et al., 1992).

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 $1{,}066^{th}$ elements are resident but are not stored in a contiguous chunk?
2. How are indexes on pointer-valued fields dealt with when references are swizzled?

Another thing that might have to be taken into account in accessing a large object is that a large object can be stored on secondary memory in a compressed format, and must be transformed before parts of it can be accessed in main memory.

Figure 5. Directory of a list



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 often is implemented as a derivative form of a B-tree (Biliris, 1992). In main memory, however, a different representation is appropriate that allows programs to reference a part of the large object given the OID and the offset more easily (Carey et al., 1986).

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 (Kotulla, 1992). 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. Conversely, 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 can be mapped at once into virtual memory to ascertain that the virtual memory frames associated with the large object are contiguous. Alternatively, the pages of a large object can be mapped incrementally using an additional page table as in the MONADS architecture (Rosenberg et al., 1990). Either way, address arithmetic easily can be carried out given the base address of the large object and the offset of the chunk to be accessed. Nevertheless, precautions must be taken to implement large objects. In QuickStore, for example, a special descriptor is maintained for a large object to record which pages of the large object have been accessed.

3.4.2 Swizzling and Indexes. A reference can be a key of an index, for example, of an indexed path expression in the *Access Support Relations* (Kemper and Moerkotte,

1992) 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, for example, a B-tree (Bayer and McCreight, 1972) or a hash table (Fagin et al., 1979). 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 are given the semantics of swizzled references (virtual addresses) by overloading their operators. Edelson (1992) gave a good survey on the general use of smart pointers.

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 (Section 3.2.2) every other swizzling technique can be found there.

4. Adaptable Pointer Swizzling

The discussion of the pros and cons of the individual swizzling techniques (Section 3) demonstrates 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.
2. Large objects sometimes 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 run-time speed-up to storage overhead ratio.

These observations have led to the design of an *adaptable* object manager for the object base system GOM. The adaptable object manager provides the flexibility to

Table 2. Classification of swizzling techniques in existing systems

Technique Used	*System*	*Reference*
(objectwise) eager indirect	LOOM	Kaehler and Krasner (1983)
lazy optimistic	PS-algol	Atkinson et al. (1983), Cockshott et al. (1984)
	EPVM 2.0	White and DeWitt (1992)
(objectwise) eager indirect	ORION	Kim et al. (1988)
no-swizzling	O_2	Bancilhon et al. (1988)
memory mapping, i.e., (page-/segment-wise) eager direct in VM	ObjectStore	Lamb et al. (1991)
	Texas	Wilson and Kakkad (1992), Singhal et al. (1993)
	QuickStore	White and DeWitt (1994)
	Eos	Gruber et al. (1992)
	FLASK	Munro et al. (1994)
adaptable swizzling	GOM	Kemper and Kossmann (1993)

choose the most effective swizzling techniques for every individual application. Specific performance experiments that justify adaptable pointer swizzling were reported in Kemper and Kossmann (1993). 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 usually must 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

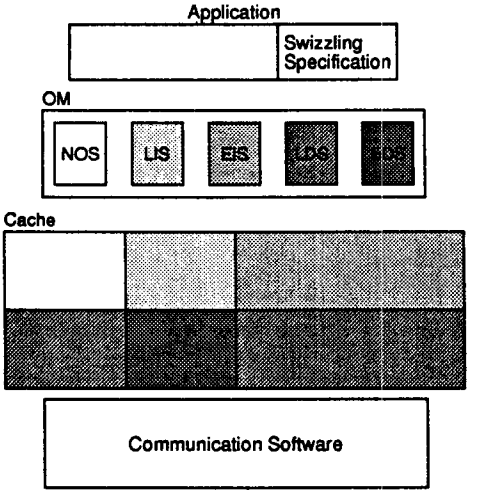
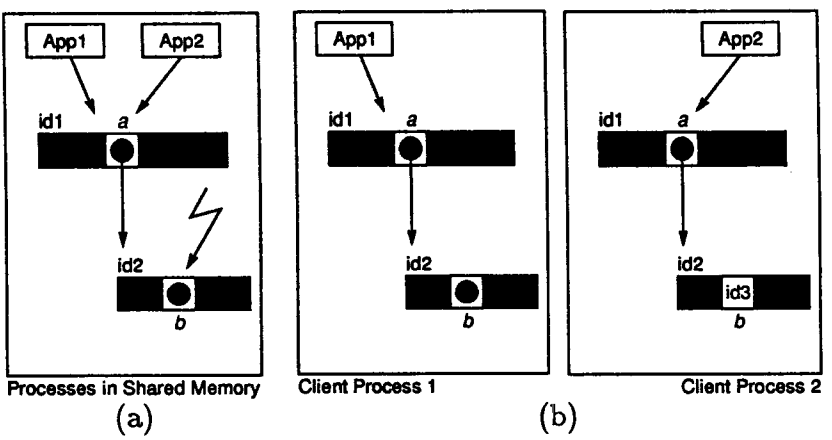


Figure 7. Conflicts due to adaptable pointer swizzling



4.1.1 Conflicting Applications. If two applications conflict with each other (i.e., they 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 *Appl* 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, *Appl* 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 also must keep its own copy of object *id1* because this object contains a pointer to object *id2*.

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

To overcome these difficulties, we propose a *lazy reswizzling* approach (i.e., references are reswizzled as soon as their "home objects" are accessed for the first time by application *App2*). To realize lazy reswizzling, the objects are protected as long as they have not been accessed. When application *App1* 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. When 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 is 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 representations 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 coherent with the eager loading effect of eager direct swizzling observed in Section 3.2.

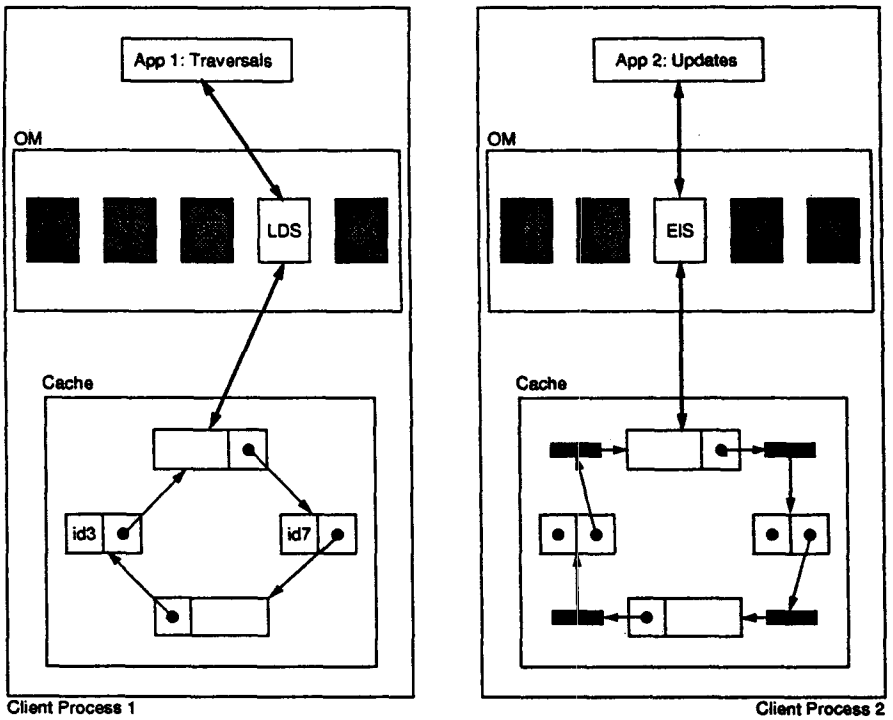
4.2 Granularities of Pointer Swizzling

4.2.1 Application-Specific Swizzling. The "coarsest" way to realize adaptable pointer swizzling is to swizzle all the pointers uniformly 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 facilitates lazy direct swizzling for applications that traverse the object base with high locality (left-hand side of Figure 8) and eager indirect swizzling for different applications that carry out structural updates in addition (right-hand side of 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, because 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 (Cattell and Skeen, 1992). All the references that refer to *Parts* (i.e., those in the *Connections*) are swizzled

Figure 8. Scenario of application-specific swizzling



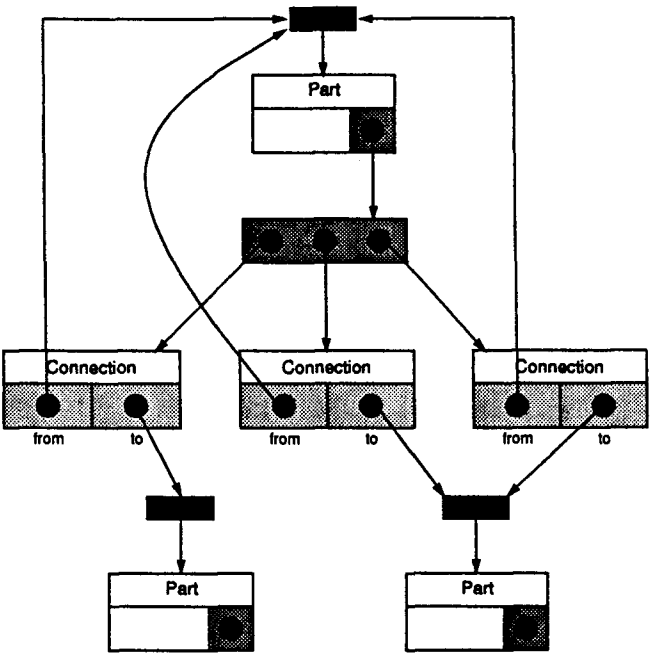
eagerly and indirectly, whereas all the other references are swizzled eagerly and directly. This example does not emphasize that, within the same object, references can be swizzled differently; for type-specific swizzling, the type of the *referenced* object (and not the type of the "home object") determines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

This example illustrates how type-specific swizzling allows the use of eager direct (and optimistic) swizzling, the most efficient technique for object lookups. In application-specific mode, this technique usually must 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., Kemper et al., 1991). Only in strongly typed languages can the compiler 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

Figure 9. Type-specific swizzling in the 001 benchmark



to the definition of *type descriptors* in Texas (Singhal et al., 1993) 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.

In conclusion, type-specific swizzling better exploits 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* in which it is stored. 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, from);
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;
```

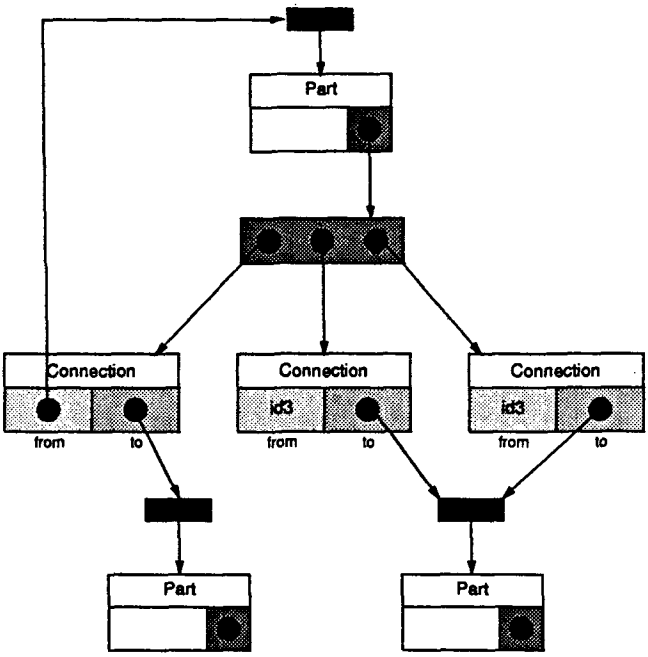
If the *to* fields are swizzled directly and the *from* fields are swizzled indirectly, the indirectly swizzled reference of *myConn.from* must be "translated" before it is copied into *myConn.to*. Translations also must be carried out to evaluate Boolean expressions such as *myConn.from* = *yourConn.to*.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

So far, the focus has been on inter-object references. A specific class of context is defined by variables. In our scheme, the identifier of each variable defines its own context. Often a special treatment of variables is followed, for example, in the first version of EXODUS (Schuh et al., 1991) or in Kossmann (1991). 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 (e.g., the repetitive computation of a path expression in a loop), can reference-specific swizzling be profitable. However, in these cases, compile-time techniques (e.g., code motion; Morel and Renvoise, 1979) probably do a much better job. Consequently, reference-specific swizzling was not investigated any further.

Figure 10. Context-specific swizzling in the 001 benchmark



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 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 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, we 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 (1992). Table 3 describes the session variables and Table 4 describes the *SW, US, LO,* and *UP* functions.

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

Table 3. Session variables

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.
$m(st)$	Depends on whether eager or lazy swizzling is used.

Table 4. Cost functions

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.

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 the exact fan-in of an object. Only an average fan-in $\overline{fi}$ can be determined, as a rule. Obviously, considering only the average fan-in is a source of inaccuracy. Nevertheless, the experiments reported in Kemper and Kossmann (1993) showed that this approximation works very well in almost every case.

5.1.1 Determining the Functions of the Cost Model. Table 5 lists the costs to read a field of an object containing an **int** (4 bytes) or a reference (8 bytes).[2] It could be concluded that eager-direct swizzling has the potential of being 6.5 times more efficient than no-swizzling. One investigator found lazy swizzling to be up to 10% inferior to eager swizzling for pure lookups (Moss, 1992).

The tremendous gap we found between transient C (called *TC*) and eager direct swizzling (also observed for the pointer swizzling techniques in White and DeWitt, 1992) is due to two factors: in the first place, the object manager is based on the EXODUS storage manager, which provides user-descriptors that cause an additional indirection; in the second place, each time an object is accessed, the object manager sets a flag to carry out an *LRU* replacement policy. This procedure is omitted in *TC*. The gap is even wider for lookups on fields that contain references. In *TC*, pointers that are 4 bytes long are copied, whereas, in a persistent object manager,

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

Table 5. Object lookups in μs

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

See Table 1 for definitions.

Table 6. Swizzling and unswizzling a reference 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

See Table 4 for cost functions.

swizzled and non-swizzled references that are 8 bytes long are copied.

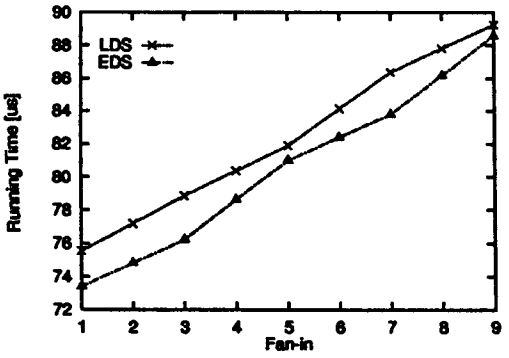
Table 6 lists the costs of swizzling and unswizzling a reference. For direct swizzling, these costs depend on the fan-in, because the RRL of the referenced object must be maintained. For $fi = 0$, the costs are particularly high because, in this case, an RRL (or a descriptor) is allocated when the reference is swizzled, and destroyed when the reference is unswizzled again. With increasing fan-in, the cost to swizzle and unswizzle a reference directly grows proportionally, whereas the cost to swizzle and unswizzle a reference indirectly is 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 11*a*), while the cost for indirect swizzling is constant. The time to modify a field that contains an **int** is shown in Figure 11*b*. 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 *TC* 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 *TC*.

5.1.2 Best and Worst Case Analyses. 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 outperforms no-swizzling 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 is assumed that $\overline{fi} = 25$.

Figure 11. Object updates in μs



(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)

(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.

Table 7. Swizzling and unswizzling a reference in μs

best/ worst	*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

See Table 1 for definitions.

The cost of eager swizzling can become arbitrarily high (∞) when almost every reference is swizzled unnecessarily (i.e., without being dereferenced). In our comparison of eager-direct and eager-indirect swizzling, the fact that eager direct swizzling can cause additional I/O activity is disregarded.

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

5.2 Cost Models for Type and Context-Specific Swizzling

For type-specific swizzling, Equation (1) is adapted in two ways. One, the costs induced by every granule are summed up. Accordingly, Equation (1) is applied to every type t, considering the type-specific parameters $m_t(st_t)$, l_t, and u_t, and

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

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	–	–

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.

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

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 C) must be considered.

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

5.2.1 Determining the Additional Cost Functions. In the benchmark environment (Section 6.1), the overhead for late binding when calling the type-specific function, referred to as 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 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 Analyses. It is always possible to specify context-specific swizzling so that no translations are necessary. Consequently, the best and worst

case analyses 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 each 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.

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

The best case of type and context-specific swizzling is an application that dereferences some references with very high locality, and others only once. For such an application, type and context-specific swizzling allow eager direct swizzling where it is profitable and no-swizzling where it is not. On the other hand, if eager swizzling is not profitable, application-specific swizzling faces a dilemma between no-swizzling and lazy indirect swizzling, 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 where 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.

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

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 always are 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 performed in-line, 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 (Kossmann, 1991; Kotulla, 1992).

Considering the data structures of the OO1 benchmark (Cattell and Skeen, 1992), 43% of the main memory must be invested for each descriptor or RRL 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 the references that refer to objects located in page A. 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 rarely are 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 Section 5 are confirmed with the help of the OO1 benchmark (Cattell and Skeen, 1992). Diverse application profiles, as well as various object base configurations and the influence of object caching and clustering, are the focus of the present article. In a previous article (Kemper and Kossmann,

1993), we presented experiments that investigated specifically the break-even points between the swizzling techniques and the influence of the fan-in.

6.1 The Benchmark Environment

6.1.1 Software and Hardware Used. 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 1,000 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 (Carey et al., 1986). As stated in Section 4, the GOM object manager permits applications to use no-swizzling or one 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 (Cattell and Skeen, 1992). 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 20,000 *Parts* and 60,000 *Connections* were measured. In GOM, which maps logical OIDs to physical addresses using a linear hash table with separators (Larson, 1988), such a database consumed 8.9 MB.

The *Parts* and *Connections* were described as follows: *part-ids* were numbered from 1 through 20,000 and, for every *Part*, three *Connections* existed whose *from* field referred to that *Part* (Cattell and Skeen, 1992). To support *Traversals*, references to these three *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 the implementation of this operation usually depends on the swizzling technique used, there is no swizzling-specific cost in creating an object. However, newly created objects usually are initialized immediately. This initialization can be seen as a sequence of operations that have the same profile as the *Lookup* and *Update* operations.

6.2 Lookups

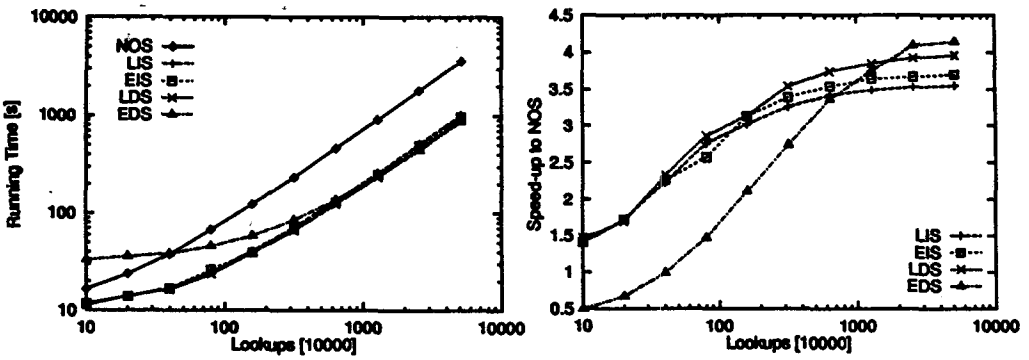
The *Lookup* operation was measured on a 10,000 *Parts* and 30,000 *Connections* object base (i.e., an object base in which all *Parts* and *Connections* fitted in the main-memory buffers so that eager direct swizzling was a reasonable approach). With increasing computation intensity, swizzling becomes more and more attractive (i.e., the speed-up of swizzling against no-swizzling increases; Figure 12).

The maximum speed-up that can be achieved is about 4.5. In Section 5, the maximum speed-up was calculated to be 6.5. However, here and in most realistic applications, the results are diluted (e.g., by calls of the "random" function 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.

Figure 12. Measuring the *Lookup* operation



6.3 Traversals

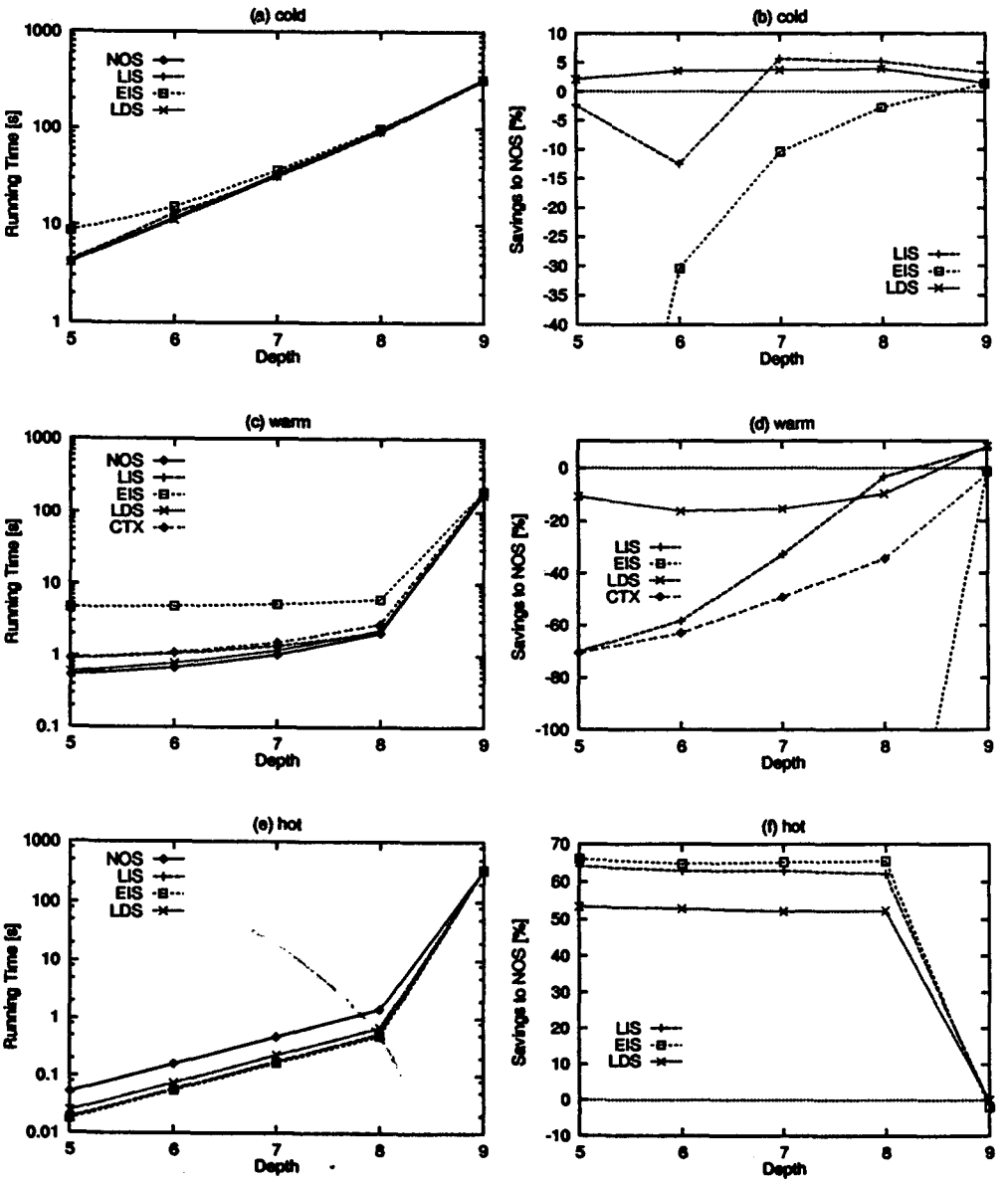
The *Traversals* were carried out *cold, warm,* and *hot* on a 20,000 *Parts,* and 60,000 *Connections* object base. The cold *Traversals* are I/O bound. In this case, it did not make much difference whether references were swizzled or not, because the running time of the application was dominated by I/O activity rather than in-memory object lookups. Figure 13*a* shows the running time of every technique,[3] and Figure 13*b* shows the savings obtained by the swizzling techniques as compared to no-swizzling.[4] Only eager indirect swizzling dropped behind due to the unnecessary swizzling of references; it caught up with increasing depth, since with increasing depth, the effect of cycles in the object graph became more apparent.

The warm *Traversals* were carried out in the following way: first, a *Traversal* was carried out using no-swizzling and, then, the running time was measured to carry out the same *Traversal,* using the alternative swizzling techniques. This experiment investigated the behavior of the system when pages were buffered in the client after commit of an application (Section 4.1); many objects were main-memory resident, but they were not in the right representation. Again, the running time (Figure 13*c*) and the comparison (Figure 13*d*) 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, because the low I/O activity diluted the results. Here, also, context-specific swizzling (*CTX*) was investigated to

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

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

Figure 13. Measuring *Traversals*



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 13*e* and Figure 13*f* 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 was "warmed up" by carrying out a *Traversal* using no-swizzling beforehand.

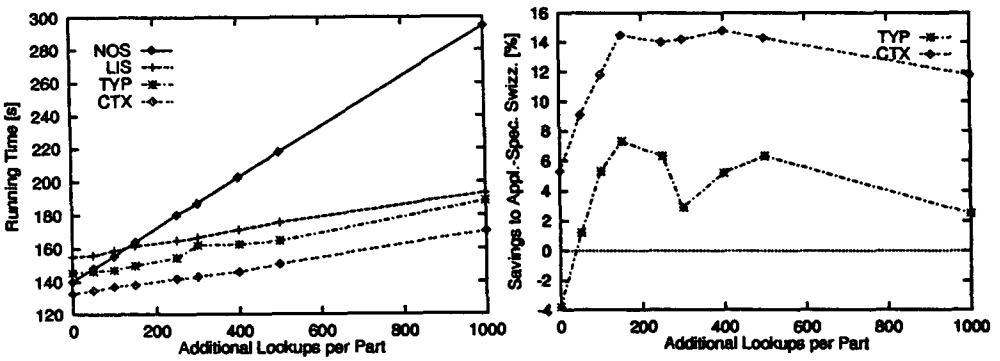
In this experiment, application-specific swizzling faced a dilemma. On the one hand, no-swizzling should have been used to carry out the warm *Traversal* efficiently; on the other hand, the additional *Lookups* provided enough locality to justify direct swizzling of references referring to *Parts*. Type 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 in the main-memory buffers. To improve locality, the set of all *Connections* was partitioned

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



into several disjoint and equally-sized subsets so that every subset fitted in 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. The approach 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.

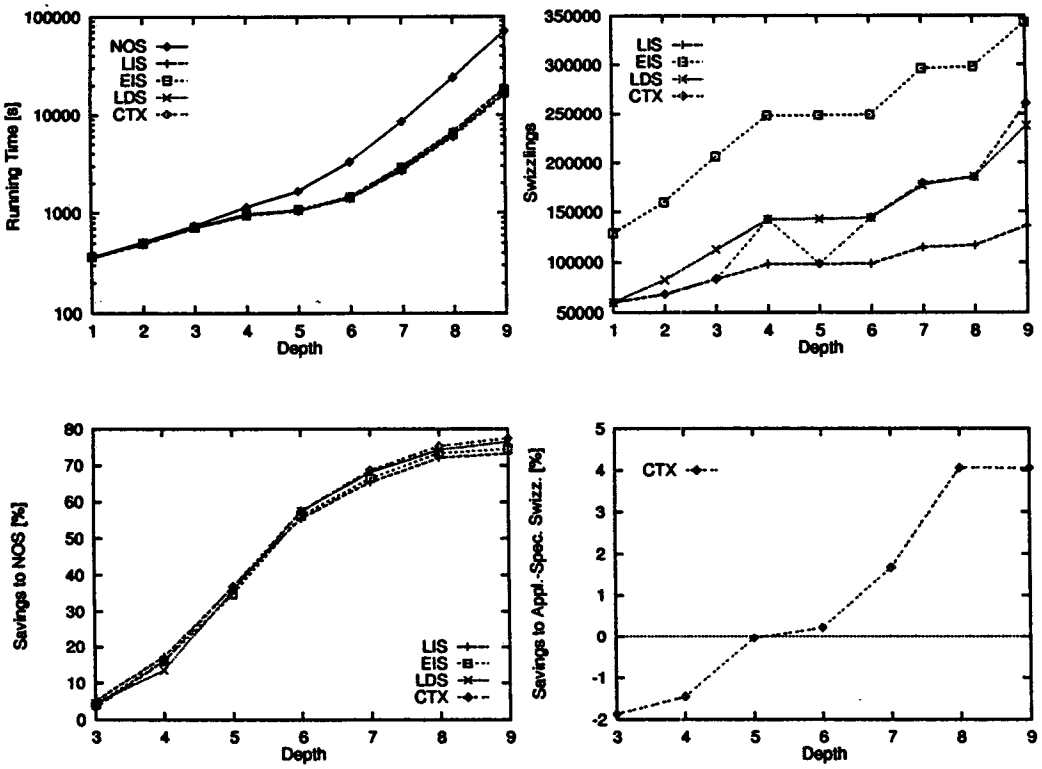
To reduce the running time of the benchmark, the object base, as well as the buffer pool were scaled down. Figure 15 summarizes the experimental results for a 10,000 *Parts* and 30,000 *Connections* object, using a buffer pool of 500 pages. With increasing depth, the running time increased exponentially. 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 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

Figure 15. Measuring *Reverse Traversals*



been executed using the same swizzling technique). As expected, direct swizzling causes tremendous overhead due to maintaining the RRLs, and indirect swizzling outperforms no-swizzling by savings of more than 50%, since it avoids consulting the ROT. The best performance, however, can be achieved by type and context-specific swizzling, since these strategies allow the references referring to *Connections* to be swizzled 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.

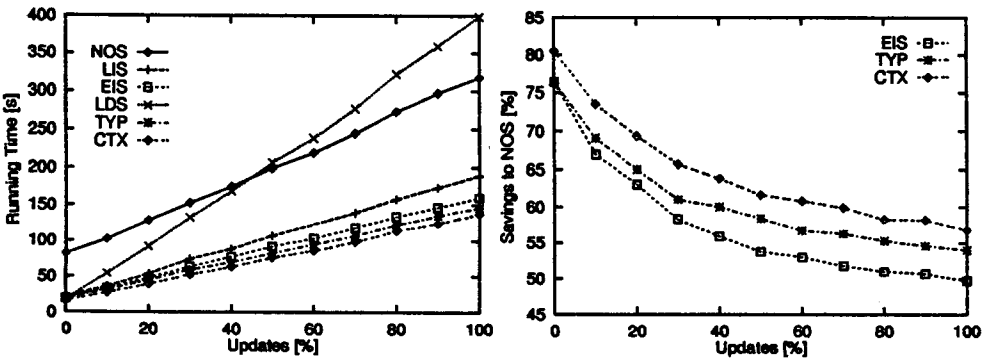
Figure 16 summarizes the experiments for *Updates* in combination with *Lookups*. With an increasing number of *Updates*, the savings of swizzling compared to no-swizzling decreased because *Updates* are relatively more expensive than *Lookups*. On the other hand, type-specific swizzling became more 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.

Table 9. Running time in μs and savings in % of pointer swizzling for the *Update* operation (hot)

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

See Table 1 for definitions.

Figure 16. Operation mix: Updates and lookups (hot)



6.6 The Influence of Locality on Pointer Swizzling

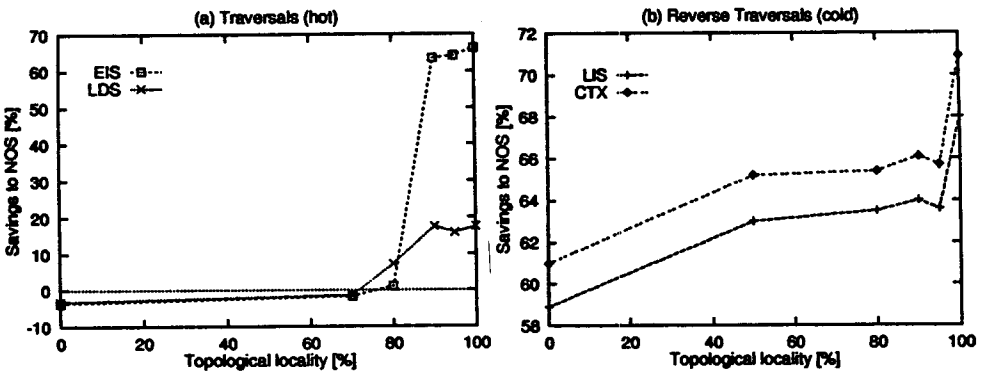
The most decisive parameter for any swizzling technique is the application's (temporal and spatial) *locality* (i.e., the probability that an application accesses an object that is already main-memory resident). The higher the locality, the higher the ratio l/m in Equation (1), and the more profitable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

Locality is determined not only 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 also were investigated—in which, for example, all (100%) or no (0%) *Connections* satisfied this demand. This parameter is referred to as the *topological locality* of the OO1 object base.

In 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

Figure 17. Varying topological locality



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 (Kim et al., 1988). 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 never are 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	20,000	60,000	100	8.9 MB
B	100,000	300,000	100	36.3 MB
C	20,000	60,000	9	42.5 MB

Figure 18. Measuring hot *Traversals* in an object cache (OC) and in a page buffer (PB)

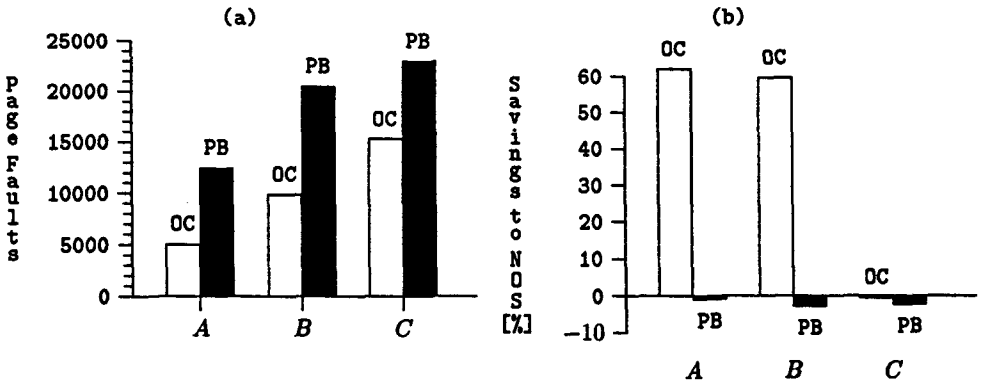


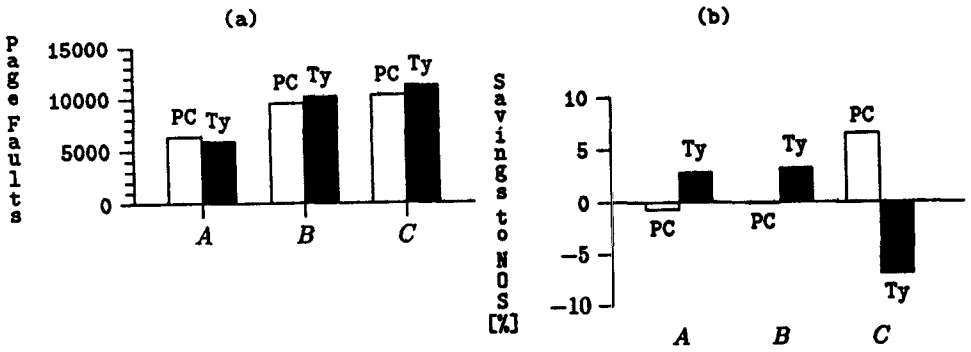
Figure 18*a* illustrates how, due to object caching, the number of page faults can be decreased significantly. For example, for the object base configuration *A*, fewer than half the faults occurred under the *OC* architecture. Figure 18*b* illustrates how this affected the performance of pointer swizzling; 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 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 and swizzling.

7. Making the Right Choice

In Section 5, a cost model for determining the most profitable swizzling strategy is described. This section details how, in practice, the variables of the cost model can be derived so that the most profitable swizzling strategy may be determined. Some of the ideas were borrowed from *clustering* in object bases (Gerlhof et al.,

Figure 19. Measuring cold *Traversals* in a *Part-to-Connection* (PC) and in a *Type-based* (Ty) clustered object base



1993), where 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 profile of an application can be determined by monitoring. The application is executed in the *training mode,* using no-swizzling.[5] Thus, variables *o, l, u,* and *m* (*st*) are computed for every context (i.e., the smallest granularity) by adding up the contribution of every object and reference.

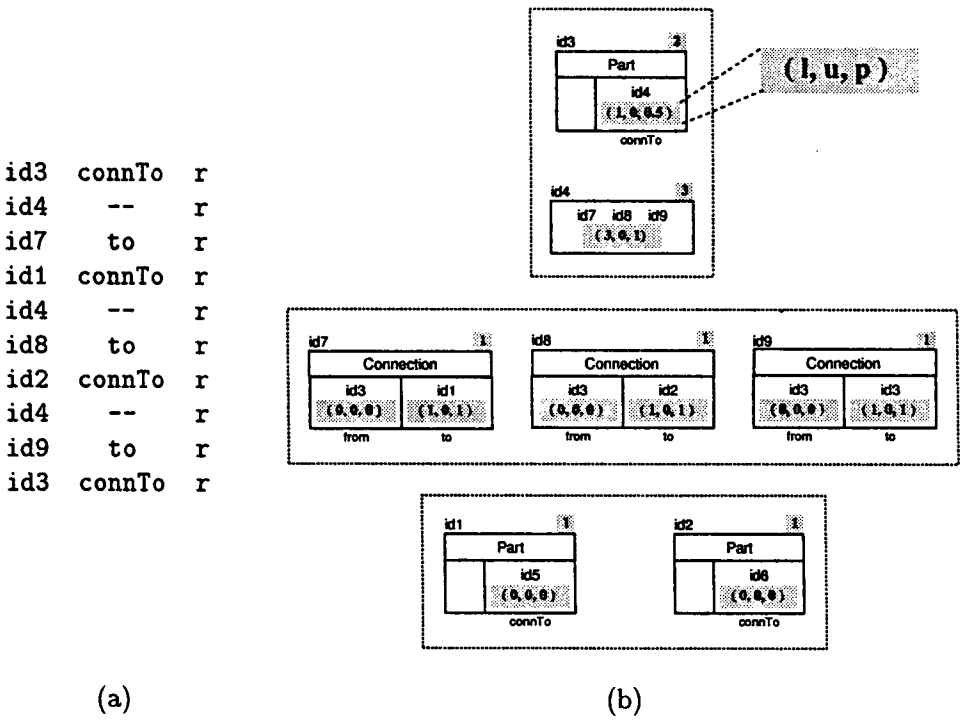
Figure 20 illustrates this approach with the help of a *swizzling graph* (Figure 20*b*) and the trace of a *Traversal* with depth 1 (Figure 20*a*). The trace keeps a record of every call of the object manager; for every access to an object, it records the OID of the object, registers the attribute if 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* (Tsangaris and Naughton, 1991). 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 20*b*, 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.

---

5. It is also possible to carry out a trace-driven simulation (Dan et al., 1993).

Figure 20. Generating a swizzling graph (b) from an object trace (a)



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, a count is kept of the number of times 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 20*a*.

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	
to	3	0	1	3	3	
connTo	1	0	1/4	1	4	*CTX*
{*Conns*}	3	0	1	3	3	
Parts	3	0	1/2	3	6	
Conns	3	0	1	3	3	*TYPE*
{*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 Section 7.1 (incremental loading was assumed in any case) and, therefore, for some granules, the decision to swizzle eagerly and directly should be reconsidered.

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

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

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

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

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). 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 (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 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 become 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 (Kossmann et al., 1993), can be applied to making the right swizzling decision. Decapsulation was developed in previous work on application profile dependent optimization (Gerlhof et al., 1992; Kemper et al., 1992, 1994). 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 one second).

8. Conclusion

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 supported by the German Research Council (DFG), and it was carried out while Donald Kossmann was a fellow in the "Graduiertenkolleg Informatik und Technik" at the Technical University (RWTH) of Aachen. 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, two anonymous referees and especially Malcolm Atkinson for their substantial suggestions for the revision of the paper. We are grateful to Judith Kossmann for improving the presentation significantly.

References

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

Bancilhon, F., Barbedette, G., Benzaken, V., Delobel, C., Gamerman, S., Lécluse, C., Pfeffer, P., Richard, P., and Velez, F. The design and implementation of O_2, an object-oriented database system. In: Dittrich, K.R., ed., *Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science No.334,* New York: Springer-Verlag, 1988, pp. 1-22.

Bayer, R. and McCreight, E.M. Organization and maintenance of large ordered indices. *Acta Informatica*, 1(3):173-189, 1972.

Biliris, A. The performance of three database storage structures for managing large objects. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, San Diego, CA, 1992.

Bruns, K., Kilger, C., Kossmann, D., Moerkotte, G., Walter, H.-D., and Zachmann, A. Objekte in multiplen Repräsentationen. *Workshop on OODBMS of the German "Gesellschaft für Informatik*, Frankfurt, Germany, 1992.

Carey, M., DeWitt, D., Richardson, J., and Shekita, E. Object and file management in the EXODUS extensible database system. *Proceedings of the Conference on Very Large Data Bases*, Kyoto, Japan, 1986.

Cattell, R. and Skeen, J. Object operations benchmark. *ACM Transactions on Database Systems*, 17:1-31, 1992.

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J., and Morrison, R. Persistent object management system. *Software—Practice and Experience*, 14:49-71, 1984.

Cockshott, W.P. and Foulk, P.W. Implementing 128 bit persistent addresses on 80×86 processors. In: Rosenberg, J. and Keedy, J.L., eds. *Security and Persistence, Workshops in Computing*, New York: Springer-Verlag, 1990, pp. 123-136.

Dan, A., Yu, P., and Chung, J.-Y. Database access characterization for buffer hit prediction. *Proceedings of the IEEE Conference on Data Engineering*, Vienna, Austria, 1993.

DeWitt, D.J., Futtersack, P., Maier, D., and Velez, F. A study of three alternative workstation server architectures for object-oriented database systems. *Proceedings of the Conference on Very Large Data Bases*, Brisbane, Australia, 1990.

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

Fagin, R., Nievergelt, J., Pippenger, J., and Strong, H. Extendible hashing—A fast access method for dynamic files. *ACM Transactions on Database Systems*, 4(3):315–344, 1979.

Gerlhof, C., Kemper, A., Kilger, C., and Moerkotte, G. Clustering in object bases. Technical Report 6/92, Fakultät für Informatik, Universität Karlsruhe, D-76050 Karlsruhe, 1992.

Gerlhof, C., Kemper, A., Kilger, C., and Moerkotte, G. Partition-based clustering in object bases: From theory to practice. *Proceedings of the International Conference on the Foundations of Data Organization and Algorithms (FODO)*, Chicago, IL, 1993.

Gruber, O., Amsaleg, L., Daynès, L., and Valduriez, P. Eos, an environment for object-based systems. *Proceedings of the Hawaii International Conference on System Sciences*, Hawaii, 1992.

Hosking, A.L. and Moss, J.E.B. Towards compile-time optimizations for persistence. In: Dearle, A., Shaw, G.M., and Zdonik, S.B., eds. *Implementing Persistent Object Bases*, San Mateo, CA: Morgan-Kaufmann, 1991.

Hosking, A.L. and Moss, J.E.B. Object fault handling for persistent programming languages: A performance evaluation. *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, Washington, DC, 1993.

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

Kemper, A., Kilger, C., and Moerkotte, G. Function materialization in object bases: Design, implementation and assessment. *IEEE Transactions on Knowledge and Data Engineering*, 6(4):587–608, 1994.

Kemper, A. and Kossmann, D. Adaptable pointer swizzling strategies in object bases. *Proceedings of the IEEE Conference on Data Engineering*, Vienna, 1993.

Kemper, A. and Kossmann, D. Dual-buffering strategies in object bases. *Proceedings of the Conference on Very Large Data Bases*, Santiago, Chile, 1994.

Kemper, A. and Moerkotte, G. Access support relations: An indexing method for object bases. *Information Systems*, 17(2):117–146, 1992.

Kemper, A. and Moerkotte, G. *Object-Oriented Database Management: Applications in Engineering and Computer Science*. Englewood Cliffs, NJ: Prentice Hall, 1994.

Kemper, A., Moerkotte, G., and Steinbrunn, M. Optimizing Boolean expressions in object bases. *Proceedings of the Conference on Very Large Data Bases*, Vancouver, Canada, 1992.

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

Khoshafian, S.N. and Copeland, G.P. Object identity. *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, Portland, OR, 1986.

Kim, W., Ballou, N., Chou, H.T., Garza, J.F., Woelk, D., and Banerjee, J. Integrating an object-oriented programming system with a database system. *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, San Diego, CA, 1988.

Koch, D.M. and Rosenberg, J. A secure RISC-based architecture supporting data persistence. In: Rosenberg, J. and Keedy, J.L., eds. *Security and Persistence, Workshops in Computing,* New York: Springer-Verlag, 1990. Also in: *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, Germany, 1990.

Kossmann, D. Entwurf und Implementierung von Laufzeitoptimierungsmaβnahmen im GOM-Prototyp. Master's thesis, Universität Karlsruhe, Fakultät für Informatik, D-76050 Karlsruhe, 1991.

Kossmann, D., Steinbrunn, M., and Kemper, A. Decapsulation: Estimating execution profiles in object bases. Unpublished Manuscript, 1993.

Kotulla, A. Neuentwurf und Leistungsbewertung des Laufzeitsystems im objekt-orientierten Datenbanksystem GOM. Master's thesis, RWTH Aachen, Lehrstuhl für Informatik III, D-52056 Aachen, 1992.

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

Larson, P.-Å. Linear hashing with separators—A dynamic hashing scheme achieving one-access retrieval. *ACM Transactions on Database Systems*, 13(3):366-388, 1988.

Maier, D. and Stein, J. Development and implementation of an object-oriented DBMS. In: Shriver, B. and Wegner, P., eds. *Research Directions in Object-Oriented Programming*, Cambridge, MA: MIT Press, 1987, pp. 355-392.

McAuliffe, M.L. and Solomon, M.H. A trace-based simulation of pointer swizzling techniques. *Proceedings of the IEEE Conference on Data Engineering*, Taipei, Taiwan, 1995.

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

Moss, J.E.B. Design of the Mneme persistent object store. *ACM Transactions on Office Information Systems*, 8(2):103-139, 1990.

Moss, J.E.B. Working with persistent objects: To swizzle or not to swizzle. *IEEE Transactions on Software Engineering*, 18(8):657-673, 1992.

Moss, J.E.B. and Sinofsky, S. Managing persistent data with Mneme: Designing a reliable shared object interface. In: Dittrich, K.R., ed. *Advances in Object-Oriented Database Systems, Lecture Notes in Computer Science No.334,* New York: Springer-Verlag, 1988, pp. 298-316. (*Proceedings of the Second International Workshop on Object-Oriented Database Systems,* Bad Münster, Germany).

Munro, D.S., Connor, R.C.H., Morrison, R., Scheuerl, S., and Stemple, D.W. Concurrent shadow paging in the FLASK architecture. In: Atkinson, M., Maier, D., and Benzaken, V., eds. *Persistent Object Systems, Workshops in Computing,* New York: Springer-Verlag, 1994, pp. 16-42. (*Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994).

Rosenberg, J., Henskens, F.A., Brown, A.L., Morrison, R., and Munro, D. Stability in a persistent store based on a large virtual memory. In: Rosenberg, J. and Keedy, J.L., eds. *Security and Persistence, Workshops in Computing,* New York: Springer-Verlag, 1990, pp. 229-245. *Proceedings of the International Workshop on Computer Architectures to Support Security and Persistence of Information,* Bremen, Germany, 1990).

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

Russel, G., Shaw, P., and Cockshott, P. DAIS: An object-addressed processor cache. In: Atkinson, M., Maier, D., and Benzaken, V., eds. *Persistent Object Systems, Workshops in Computing,* New York: Springer-Verlag, 1995, pp. 374-386. *Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994).

Schuh, D., Carey, M., and DeWitt, D. Persistence in E revisited—implementation experiences. In: Dearle, A., Shaw, G.M., and Zdonik, S.B., eds., *Implementing Persistent Object Bases*, San Mateo, CA: Morgan-Kaufmann, 1991, pp. 345-359. (*Proceedings of the Fourth International Workshop on Persistent Object Systems*), Martha's Vineyard, MA, 1990).

Shekita, E. and Zwilling, M. Cricket: A mapped, persistent object store. In: Dearle, A., Shaw, G.M., and Zdonik, S.B., eds. *Implementing Persistent Object Bases*, San Mateo, CA: Morgan-Kaufmann, 1991, 89-102. (*Proceedings of the Fourth International Workshop on Persistent Object Systems,* Martha's Vineyard, MA, 1990).

Singhal, V., Kakkad, S., and Wilson, P. Texas: An efficient, portable persistent store. In: Albono, A. and Morrison, R., eds. *Persistent Object Systems, Workshops in Computing,* New York: Springer-Verlag, 1993, 11-33. (*Proceedings of the Fifth International Workshop on Persistent Object Systems,* San Miniato (Pisa), Italy, 1992).

Suzuki, S., Kitsuregawa, M., and Takagi, M. An efficient pointer swizzling method for navigation intensive applications. In: Atkinson, M., Maier, D., and Benzaken, V., eds., *Persistent Object Systems, Workshops in Computing,* New York: Springer-Verlag, 1995, pp. 79-95. (*Proceedings of the Sixth International Workshop on Persistent Object Systems,* Tarascon, France, 1994).

Tsangaris, M.M. and Naughton, J.F. A stochastic approach for clustering in object bases. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, Denver, CO, 1991.

Vaughan, F. and Dearle, A. Supporting large persistent stores using conventional hardware. In: Albono, A. and Morrison, R., eds. *Persistent Object Systems, Workshops in Computing,* New York: Springer-Verlag, 34-53, 1993. (*Proceedings of the Fifth International Workshop on Persistent Object Systems,* San Miniato (Pisa), Italy, 1992).

Velez, F., Bernard, G., and Darnis, V. The O_2 object manager: An overview. *Proceedings of the Conference on Very Large Data Bases*, Amsterdam, 1989.

White, S.J. and DeWitt, D. QuickStore: A high performance mapped object store. *Proceedings of the ACM SIGMOD Conference on the Management of Data*, Minneapolis, MN, 1994.

White, S.J. and DeWitt, D.J. A performance study of alternative object faulting and pointer swizzling strategies. *Proceedings of the Conference on Very Large Data Bases*, Vancouver, B.C., 1992.

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

Wilson, P. and Kakkad, S. Pointer swizzling at page fault time: Efficiently supporting huge address spaces on standard hardware. *Proceedings of the International Workshop on Object Orientation in Operating Systems*, Paris, 1992.

Winslett, M. Architecture and performance for object-oriented DBMSes. Tutorial handouts for the Data Engineering Conference, Vienna, Austria, 1993.