Adaptable Pointer Swizzling Strategies in Object Bases

Alfons Kemper Donald Kossmann

Lehrstuhl für Informatik III
RWTH Aachen
Ahornstraße 55 W-5100 Aachen, Germany
[*kemper* | *kossmann*]*@informatik.rwth-aachen.de*

Abstract

In this paper we will classify and evaluate different approaches to optimizing the access to main memory resident persistent objects—techniques which are commonly referred to as "*pointer swizzling*". To speed up the access along inter-object references, the persistent pointers are transformed (swizzled) into main memory pointers (addresses). We will evaluate four different pointer swizzling techniques allowing the displacement of objects from the buffer before the end of an application and contrast them with the performance of an object manager using no pointer swizzling. Our quantitative evaluation proved that there is no *one* superior strategy for *all* application profiles. Consequently, we will suggest an adaptable system that uses the full range of pointer swizzling strategies.

1 Introduction

Object-oriented database systems are emerging as the next generation of database technology—especially for advanced, so-called "non-standard" applications, e.g., CAD/CAM. Despite their superior expressive power in comparison to the established relational systems, the acceptance of the object-oriented systems will ultimately depend on their performance. This is all the more true for the technical application areas, 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.

We investigated a class of techniques 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 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 localize the object in 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 to localize the object each time it is accessed. Thus, pointer swizzling converts database objects from an external (persistent) form containing OIDs into an internal (main memory) form replacing the OIDs by the main memory addresses of the referenced objects. This is, of course, particularly important in *computation-intensive* applications, as experienced, e.g., in CAD/CAM applications.

Various forms of pointer swizzling have by now been implemented in several persistent object managers. Among the first that employed certain forms of pointer swizzling was LOOM [KK83]. Many recently developed object-oriented database systems also incorporate pointer swizzling. In Orion [KBC+88], an architecture is proposed that divides the system buffer into two parts: a page buffer and an object buffer. Main memory resident persistent objects are copied from the page buffer into the object buffer and, in the process, the inter-object references are converted (swizzled) into main memory pointers with one indirection; i.e., the pointer refers to a so-called resident object descriptor (ROD) which, in turn, points to the referenced in-memory object. This method is referred to as indirect swizzling in our terminology. EXODUS supports two forms of pointer swizzling: in version EPVM 1.0 [SCD90], swizzling is restricted to local variables only. In the most recent version, EPVM 2.0 [WD92], a more elaborate swizzling technique is incorporated which is similar to Orion with respect to copying swizzled objects into a separate object buffer. However, the indirection via the ROD is omitted in EXODUS. Thereby, however, eliminating the possibility of replacing a swizzled object from the buffer before the end of the application. In ObjectStore [LLOW91], all references are persistently stored in their swizzled form using a memory mapping technique. Thus, page

faults are detected and handled at the operating system level.

Strong dependencies exist between pointer swizzling and other system components. This severely restricts the design choices in existing systems. Moss [Mos91] was the first who abstracted from system and/or object model characteristics and undertook a systematic, though incomplete, classification of pointer swizzling techniques. The work reported here can be seen as an augmentation along a dimension that was not taken into account by Moss: the possibility of replacing swizzled objects 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 remainder of the paper is organized as follows. Section 2 shows four alternative pointer swizzling strategies that take "precautions" (i.e., maintain control structures) for replacement of swizzled objects. Section 3 describes the benchmark experiments that are presented in Section 4. Section 5 suggests three alternative approaches for an adaptable object base run-time system. Section 6 contains our conclusions.

2 Classification of the Techniques

Moss [Mos91] carried out a thorough analysis of strategies that preclude object replacement. We investigated pointer swizzling strategies that provide the necessary control structures for object replacement. These strategies were classified into two dimensions depending on the following two criteria: (1) When is a pointer swizzled? (2) Can a swizzled pointer refer to an object that is not main memory resident?

The first dimension was borrowed from Moss and consists of two alternative techniques referred to as *eager* and *lazy* swizzling. The second dimension refers to *direct* versus *indirect* swizzling techniques. The following table summarizes the classification scheme:

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

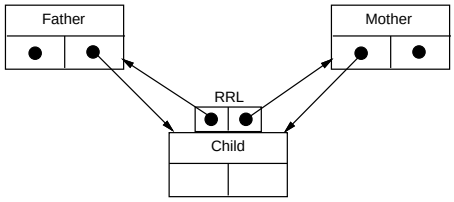
2.1 Eager and Lazy Swizzling

Eager swizzling guarantees that all the pointers in the main memory are swizzled. An object is an aggregation of pointers and other bytes representing atomic values. When an object is loaded from disk the object is "scanned through" and all the pointers in the object are immediately swizzled.

In contrast to eager swizzling, *lazy* swizzling only swizzles pointers on demand; i.e., a pointer is not swizzled until the object it refers to is accessed via this particular pointer. The advantage of lazy swizzling is that no pointers are swizzled unnecessarily. On the negative side, lazy swizzling must handle two different kinds of pointers at run-time: swizzled and non-swizzled pointers. Software checks that determine the state of a pointer (swizzled or not swizzled) need to be included every time an object is accessed.

2.2 Direct and Indirect Swizzling

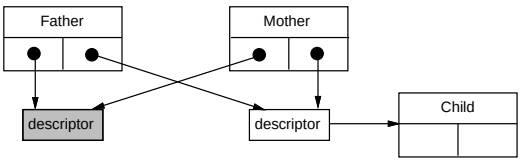
For pointer swizzling with object replacement, a new dimension was introduced. Depending on whether it is possible to swizzle a pointer that refers to an object that is not main memory resident, we distinguish between *direct* and *indirect* swizzling. Direct swizzling requires that the referenced object be resident. A directly swizzled pointer contains the main memory address of the object it references. The problem with direct swizzling is that if an object is displaced from the system buffer, i.e., is no longer resident, all the directly swizzled pointers that reference the displaced object need to be unswizzled. To unswizzle these pointers, they are registered in a list called *reverse reference list (RRL)*. The following figure illustrates this scenario:



In the case of eager direct swizzling simply unswizzling the pointers cannot be done because eager swizzling guarantees that all pointers in the buffer are swizzled. Instead those pointers (i.e., their "home objects") must be displaced, too. This may result in a "snowball" effect.

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

Indirect swizzling avoids the overhead of maintaining an RRL for every resident object by permitting the swizzling of pointers that reference non-resident objects. To attain indirect swizzling, a swizzled pointer 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; otherwise, the descriptor is marked invalid. If an object is displaced, the swizzled pointers that reference the object need not be unswizzled; only the descriptor is marked invalid. The following figure illustrates this scenario.



To be able to collect garbage every descriptor keeps a counter that counts the number of indirectly swizzled pointers referring to the descriptor. Maintaining a counter is much cheaper than maintaining an RRL when executing updates or when swizzling and unswizzling pointers. On the other hand, indirect swizzling induces an additional overhead in comparison to direct swizzling when it comes to simple object lookups. Indirect swizzling leads to an additional indirection, due to the descriptor, and to a residency check, a check of whether the descriptor is valid. For direct swizzling the information that an object is resident is coded in the swizzled pointer.

3 The Benchmark Environment

In this section the benchmark and its environment are described, as used in our experiments presented in Section 4.

3.1 Software and Hardware Used

All the experiments were carried out on a Sun SPARC station 1 plus with 40 MB memory under SunOS 4.1.1. In all the experiments the size of the system buffer was restricted to 4 MB (1000 pages at 4096 bytes). Access to persistent data was effected by calls of the GOM object manager. The GOM object manager is based on Version 1.3 of the EXODUS storage manager [CDRS86]. To determine the trade-off between the swizzling strategies (including no-swizzling), five different configurations of the GOM object manager were measured: one for no-swizzling and one for each of the four swizzling strategies. Another three configurations of the GOM object manager that are presented in Section 5 were measured to investigate the efficiency of adaptable pointer swizzling. In any configuration, a pointer to the object must be passed to the object manager to read or manipulate the state of the object. For no-swizzling, the pointers passed to the object manager are *logical* OIDs consisting of 8 bytes. Directly and indirectly swizzled pointers are also 8 bytes long.

3.2 Benchmark Specification

The benchmark used was derived from the OO1 Benchmark [CS90]. In particular, we borrowed the data structures which are defined as follows:

type *Part* **is**	**type** *Connection* **is**
[*part-id*: **int**;	[*from*: *Part*;
type: **string[10]**;	*to*: *Part*;
x, y: **int**;	*type*: **string[10]**;
built: **int**]	*length*: **int**]
end type *Part*	**end type** *Connection*

For every *Part*, r = *number_conns*/*number_parts* *Connections* exist whose *from* field reference that *Part*. The *to* fields of the *Connections* reference randomly selected *Parts*.

The following three different database configurations, called A, B and C, were measured.

configuration	*number_parts*	*number_conns*
A	6000	3000
B	3000	3000
C	3000	12000

In configuration A, only half the *Parts* are referenced by a *from* field of a *Connection* ($r = 0.5$). Furthermore, all three databases fit into the system buffer. This is intended because we did not want to measure I/O or the actual object replacements, even though our pointer swizzling techniques provide for object replacements. On the contrary, we wanted to concentrate on the analysis of the costs induced by the control structures that allow object replacement; i.e., maintaining an RRL or a descriptor.

The benchmark measures applications composed of the following two operations:

Traversal Reading the *from* field of a *Connection*; reading the *part-id* of the *Part* referenced by the *from* field; then, reading the *to* field and the corresponding *part-id*. This *Traversal* operation carries out four object lookups.

Update Overwriting the *to* field of a *Connection* with a pointer that refers to a randomly selected *Part*.

Session variables	
l	Number of lookup operations performed.
u	Number of update operations performed.
$\overline{fi}$	Average fan-in of an object.
p	Probability that a pointer is traversed.
n	Total number of pointers.
m	Number of pointers that are swizzled.

Cost parameters	
SW	Time to swizzle a pointer.
US	Time to unswizzle a pointer.
LO	Time to perform a lookup.
UP	Time to perfrom an update.

Table 1: Session variables and cost parameters.

This *Update* operation is unusual for database applications, as it changes relationships between objects. More commonly, atomic-valued attributes are updated, e.g., the coordinates of a *Vertex* in a CAD application. However, this operation was measured to investigate the costs caused by maintaining the RRL for direct swizzling.

3.3 Cost Model of Swizzling

We measured the CPU time required for pinning all the objects, for swizzling m pointers, for carrying out t *Traversals* ($l = 4 * t$ lookup operations), u *Updates*, and for unswizzling m pointers. Only the worst case that all m swizzled pointers need to be unswizzled was considered. In fact, pointers on pages that have not been modified and, therefore, do not need to be written back to disk, need not be unswizzled. For eager swizzling, $m = n$ applies, where n is the total number of pointers brought into the buffer. For lazy swizzling, only pointers that are really traversed are swizzled. If p is defined as the probability that a pointer is followed, then $m = p * n$ for lazy swizzling. The specific costs of swizzling depending on the strategy used—denoted *strat*—can be summarized by function (1). This function is an adaptation of the cost function used by Moss [Mos91]. Table 1 describes the session variables and the cost parameters of interest for the analysis.

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

Aside from the swizzling technique used *SW*, *US* and *UP* depend on the fan-in of an object. Unfortunately, it is not possible to determine in general the exact fan-in of an object. Usually, only an average fan-in $\overline{fi}$ can be determined. Obviously, considering an average fan-in only is a source of inaccuracy. Nevertheless, the experiments showed that this approximation works quite well.

4 The Quantitative Analysis

In this section the results of our performance experiments are presented. First, the values of the parameters of the cost model are listed. The cost model is verified and the break-even points at which one strategy outperforms the other determined.

4.1 Measuring the Cost Model Parameters

Table 2 lists the costs for object lookups. Each *Traversal* consists of 2 *pointer* lookups (the *from* and *to* field of a *Connection*) and 2 **int** lookups (the *part-id* fields of the two connected *Parts*). We concluded that eager direct swizzling has the potential of being 5 times more efficient than no-swizzling. We confirmed the results in [Mos91] that lazy swizzling is 10 % inferior to eager swizzling for pure lookups. For the *Traversal* operation, indirect swizzling is approximately 15 % less efficient than direct swizzling.

We also measured transient C (called *TC*). The tremendous gap between transient C and eager direct swizzling (which is also observed for the pointer swizzling techniques in [WD92]) is due to two factors: in the first place, the user-descriptors provided by the EXODUS storage manager cause an additional indirection; then, each time an object is accessed, the object manager sets a flag to carry out an *LRU* replacement strategy for the buffer. This procedure is omitted in *TC*. The performance gap is even larger for pointer lookups (e.g., reading the *from* field of a *Connection*). In transient C, direct memory pointers that are 4 bytes long are copied, whereas in the persistent object manager, swizzled and non-swizzled database pointers that are 8 bytes long are copied

Strategy	*Traversal*	*pointer Lookup*	**int** *Lookup*
TC	3.8	0.9	1.0
EDS	20.6	6.7	3.6
LDS	22.2	7.1	4.0
EIS	23.4	7.4	4.3
LIS	25.0	7.8	4.7
NOS	99.6	26.4	23.4

Table 2: Object lookups in μs.

Table 3 lists the costs of swizzling and unswizzling a pointer. For direct swizzling, these costs depend on the fan-in, as the RRL of the referenced object must be maintained. For $fi = 0$, the costs are particularly high, as in this case an RRL (or a descriptor) is created when the pointer is swizzled and destroyed when the pointer is unswizzled again. With increasing fan-in, the costs to directly swizzle and unswizzle a pointer grow proportionally, whereas the costs to swizzle and unswizzle a pointer indirectly are constant.

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

Table 3: Swizzle and unswizzle a pointer in μs.

Similarly, the costs of direct swizzling grow linearly with increasing fan-in for the *Update* operation, whereas the costs for indirect swizzling are constant. In Figure 1, the costs for the *Update* operation without a call of the Unix function *random*[1] were analyzed. The diagram in Figure 1 illustrates the linear behavior of lazy and eager direct swizzling under increasing fan-in values (plotted against the x-axis).

We also measured the time to update a field that contains an **int** (e.g., the *length* field of a *Connection*). In this case, direct swizzling outperforms indirect swizzling by approximately the same difference as for lookups, as no RRL needs to be maintained.

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

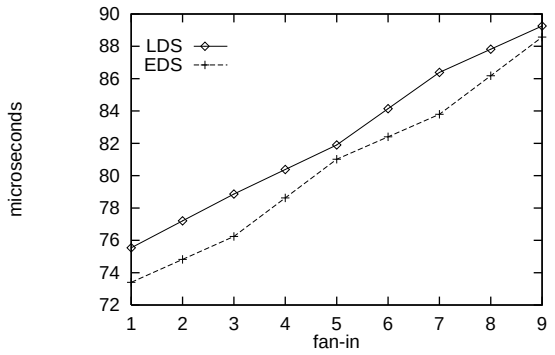
4.2 No-swizzling vs. Swizzling

The major parameter determining the virtues of swizzling over no-swizzling is the number of accesses via a particular swizzled pointer. $v = l/m$ is defined as the number of object visits per pointer. The higher this ratio is, the more favorable swizzling becomes.

In Figure 2, we ran our benchmark for the 3000 *Parts* and 12000 *Connections* database ($\overline{fi} = 8$) and $p = 0.8$. As expected, with an increasing number of *Traversals* no-swizzling (NOS) becomes less attractive and, finally, is outperformed by all swizzling strategies.

Table 4 lists the break-even points for the four swizzling strategies and two different fan-ins obtained by applying the cost model.

[1] As stated in [WD92] this function dilutes the results.



Update	TC	EDS	LDS	EIS	LIS	NOS
pointer	1.3	-	-	32.1	33.3	48.7
int	1.3	29.4	29.7	30.1	30.4	46.6

Figure 1: Object updates in μs.

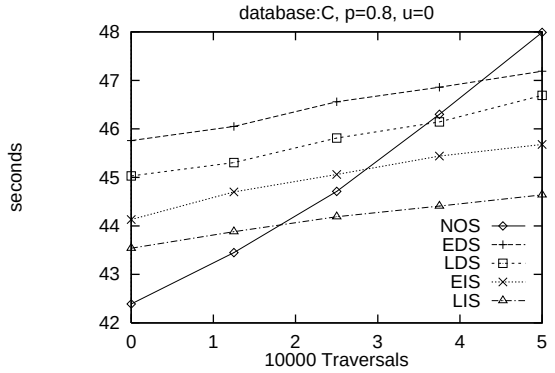


Figure 2: No-swizzling vs. pointer swizzling strategies.

4.3 Eager vs. Lazy Swizzling

Eager swizzling performs fast object accesses (lookups and updates) but carries the danger of swizzling pointers unnecessarily. Thus, eager swizzling will be superior to lazy swizzling if many lookups and updates (high l and u) are present. On the other hand, lazy swizzling will outperform eager swizzling with a low p. The average fan-in of an object for lazy swizzling can be lower than for eager swizzling. For direct swizzling techniques, this may be another argument in favor of lazy swizzling.

strat	*EDS*		*LDS*		*EIS*		*LIS*	
fi	1	8	1	8	1	8	1	8
v_{be}	4.4	3.7	4.3	3.6	3.3	2.0	3.2	1.9

Table 4: Break-even points: NOS vs. swizzling.

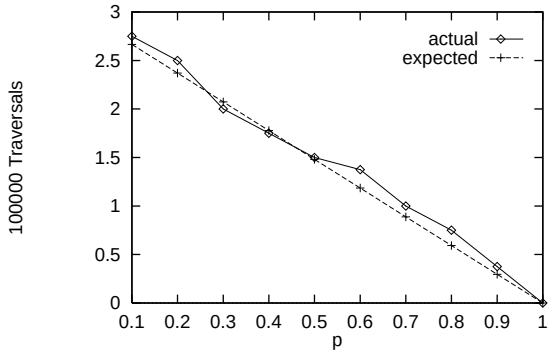


Figure 3: Break-even curve EIS vs. LIS.

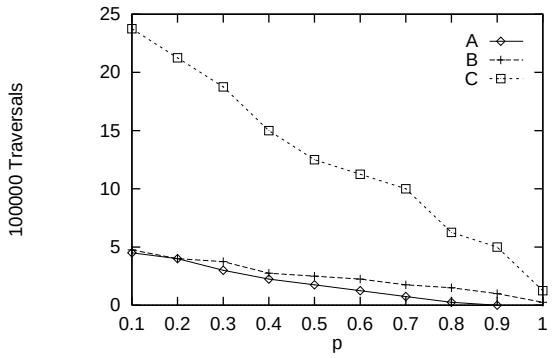


Figure 4: Break-even curves EDS vs. LDS.

In Figure 3, we determined experimentally the break-even points between eager indirect and lazy indirect swizzling for the 3000 *Parts* and 3000 *Connections* database. For a given p, we ran the benchmark with varying numbers of *Traversals* and hence determined the break-even point. The diagram compares the resulting *actual* break-even curve with the *expected* curve derived from the cost model.

Analogously, Figure 4 illustrates the break-even curves gained experimentally for eager vs. lazy direct swizzling and the three different database configurations. Again the results are very close to what we would expect when applying the cost model (not shown in Figure 4). The higher the fan-in the more expensive swizzling and unswizzling is and the less attractive eager swizzling is when $p < 1$.

4.4 Direct vs. Indirect Swizzling

Section 2 showed that direct swizzling promises more efficient object lookups. On the other hand, indirect swizzling is more efficient in swizzling and unswizzling a pointer and in updating a field, that contains a pointer.

Figure 5 illustrates the break-even curves determined running the benchmark for $p = 1$. The letters A, B and C denote the three database configurations.

The decision of whether to swizzle directly or indirectly can be made independently from the decision of whether to swizzle eagerly or lazily, if we disregard the special interaction between eager and direct swizzling, and the fact that eager swizzling may contribute to a higher fan-in. Eager (or lazy) swizzling has the same effect on lookups and updates for direct swizzling as for indirect swizzling; in both cases eager swizzling omitts the software check of whether a pointer is swizzled. Consequently, the break-even curves for *EDS* vs. *EIS* and *LDS* vs. *LIS* are almost identical.

The experiments also confirmed the expectation that the higher the fan-in becomes, the less attractive direct swizzling is. The break-even curve for the 3000 *Parts* and 3000 *Connections* database ($\overline{fi} = 2$) is clearly above the break-even curve for the 6000 *Parts* and 3000 *Connections* database ($\overline{fi} = 1$). Furthermore, the break-even curve for the 3000 *Parts* and 12000 *Connections* database ($\overline{fi} = 8$)—configuration C—has the highest gradient.

5 Adaptable Swizzling

In the preceding section, it became apparant that there is no *one* best pointer swizzling strategy for *all* situations. This observation led us to develop an *adaptable* object manager for our object base system GOM that provides the flexibility to choose the most favorable swizzling strategy, depending on the particular application profile. How this adaptability can be achieved with as little additional run-time overhead as possible is discussed in this section. Three granularities of pointer swizzling adaptability are introduced: *application-*, *type-*, and *context-specific* swizzling and the trade-off between these granularities is discussed.

In the following we will assume that only one application works on the same main memory buffer at a time. If two or more applications access the same data in the same buffer and run in parallel, the possibilities to exploit adaptable pointer swizzling are restricted [KK93].

5.1 Application-Specific Swizzling

The "coarsest" way to realize adaptable pointer swizzling is to uniformly swizzle all pointers within one application. We called this approach application-specific swizzling. At the beginning of the execution of

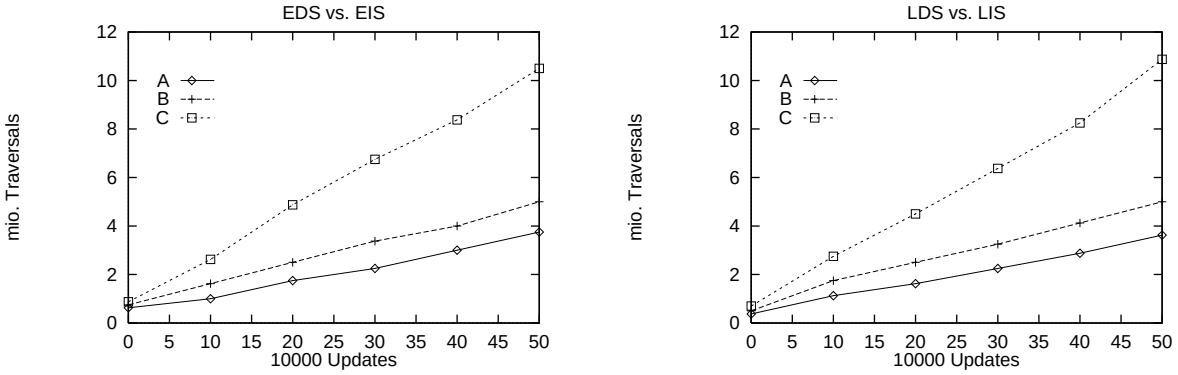


Figure 5: Break-even curves direct vs. indirect swizzling.

an application the swizzling strategy to follow is specified. For example, *application-specific* swizzling allows following indirect swizzling for applications that are update-intensive and direct swizzling for applications that perform relatively few updates.

5.2 Type- and Context-Specific Swizzling

Application-specific swizzling has one decisive disadvantage: it does not provide the flexibility to choose different swizzling techniques for different references within the same application. For example, it is not possible to swizzle the pointers that reference *Connections* directly and the pointers that reference *Parts* indirectly. However, this is exactly what could be required in an application that updates only references referring to *Part* objects. Therefore, *type-specific* swizzling was introduced, where only pointers that reference objects of the same type are uniformly swizzled, rather than uniformly swizzling all pointers application-wide.

For some application profiles, another improvement to type-specific swizzling is conceivable. Sometimes it is desirable to swizzle a pointer depending on where the pointer is used as a reference, e.g., in a local variable or in a particular field of an object. For example, if only the *to* fields of the *Connections* are updated, swizzling the pointers that appear in the *from* fields directly and swizzling only the pointers in the *to* fields indirectly may be of advantage. We called this granularity of specifying how to swizzle pointers *context-specific* swizzling.

Another advantage of type- and context-specific swizzling is their potential of avoiding the snowball effect described in Section 2 in eager direct swizzling. In practice, type- and context-specific swizzling will be the only possibility to support the EDS technique.

To attain type- and context-specific swizzling, the compiler generates, for every type, a function that specifies which swizzling strategies to follow for objects of that particular type. This function has to be invoked by the object manager each time an object of this type is loaded into the system buffer. For example, for context-specific swizzling, the compiler generates a function for *Connections* that eagerly swizzles the *from* field directly and the *to* field indirectly.

Figure 6 shows experiments carried out for the best configurations of application-, type-, and context-specific swizzling; the corner points show where a switch-over between two techniques was effected. For $u = 0$, application-specific swizzling outperforms type- and context-specific swizzling due to the additional overhead caused by invoking the function that specifies how to swizzle every time an object is loaded. This additional overhead cannot be compensated by the finer granularity, as here all pointers are swizzled eagerly and directly application-wide, in any case.

However, already for an application performing 50 000 *Updates* type- and context-specific swizzling outperform application-specific swizzling. The advantage gained by the finer granularity (swizzling the *to* fields indirectly for context-specific swizzling or swizzling all the pointers that reference *Parts* for type-specific swizzling) exceeded the additional overhead. With an increasing number of *Updates* the gap between application-specific swizzling on the one hand and type- and context-specific swizzling on the other hand grew.

Figure 6 does not emphasize the fact that context-specific swizzling causes an additional overhead over type-specific swizzling and, therefore, it may be outperformed by type-specific swizzling for some application profiles. Due to the possibility that the same object is referenced by an indirectly and by a directly

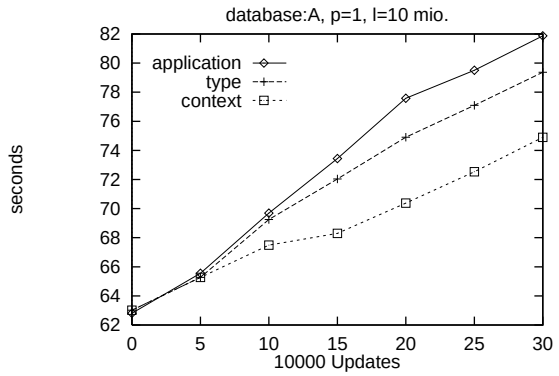


Figure 6: Granularities of pointer swizzling.

swizzled pointer, a descriptor as well as an RRL need to be maintained.

It is crucial to adaptable pointer swizzling to always know at compile-time which strategy has already been followed in swizzling a pointer. This is the reason why we do not suggest the swizzling of individual pointers, the finest granularity conceivable. In our system no additional software checks need to be inserted to determine the way a pointer was swizzled. Type- and context-specific swizzling only make sense in a strongly typed database programming language (e.g., our model GOM [KMWZ91]). In the presence of a type hierarchy how the pointers that reference a type are swizzled must be inherited by all subtypes.

6 Conclusions

In this work we investigated alternative pointer swizzling techniques for a persistent object manager that facilitates object replacement during an application run. Our 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 us to develop an adaptable object manager that allows varying the pointer swizzling technique applied according to the particular profile. We devised three different granularities within which the pointer swizzling techniques can be varied.

Determining the most appropriate swizzling technique and adjustment granularity for a given application remains an open research issue. One way—albeit the least appealing—is to require the user to specify these parameters. Our future work in this area will concentrate on static program analysis in combination with sampling the current state of the object base.

Acknowledgements

This work was carried out within the graduate college "Computer Science and Engineering" at the Technical University (RWTH) Aachen. Axel Kotulla implemented the swizzling strategies and performed the experiments. We thank Judith Kossmann for improving the presentation.

References

[CDRS86] M. Carey et al. Object and file management in the EXODUS extensible database system. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, pp. 91–100, Kyoto, Aug 1986.

[CS90] R. Cattell and J. Skeen. Engineering database benchmark. Technical report, Database Engineering Group, Sun Microsystems, Mountain View, Ca., 1990.

[KBC+88] W. Kim et al. Integrating an object-oriented programming system with a database system. In *Proc. of the ACM Conf. on Object-Oriented Programming Systems and Languages (OOPSLA)*, pp. 142–152, San Diego, Sep 1988.

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

[KK93] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. Technical Report RWTH Aachen, 1993.

[KMWZ91] A. Kemper et al. GOM: a strongly typed, persistent object model with polymorphism. In *Proc. of the Conf. on Databases in Office, Engineering, and Science (BTW)*, pp. 198–217, Kaiserslautern, Mar 1991. Springer-Verlag, Informatik-Fachberichte No. 270.

[LLOW91] C. Lamb et al. The ObjectStore database system. *CACM*, 34(10):50–63, 1991.

[Mos91] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. Technical Report 90-38 University of Massachusetts at Amherst, May 1990.

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

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