Physical Pointer Swizzling*

Silvia Nittel†

Computer Science Department
University of California, Los Angeles
Los Angeles, CA 90024

September 30, 1997

Abstract

Most of todays object oriented database systems (OODBS) employ a two-level architecture consisting of an object level and a storage level containing database objects in different formats. Before objects can be used by an application program, they have to be converted from their storage format to an in-memory format. An important aspect in this process is the conversion of references between objects to in-memory pointers so that object access is accelerated. These conversion mechanisms for references are so-called pointer swizzling strategies. Existing pointer swizzling strategies consist of many processing steps, and thus, are cumbersome and time-consuming. We introduce the *physical pointer swizzling strategy* which is based on a different kind of persistent references representation in the storage format, and results in a reduction of the number of conversion steps. By using the OO7 benchmark, we show that the physical pointer swizzling strategy achieves a twofold speedup in the conversion complex objects in oodbs.

1 Motivation

A crucial bottleneck while processing complex object structures in object oriented database systems (OODBS) is the time-consuming conversion of complex objects. The conversion of complex objects results from the architecture used in most of todays OODBS. This architecture consists of two layers: the object management layer, and the storage management layer. In both of these subsystems, objects are processed in a different format: a persistent storage format in the storage layer, and an in-memory processing format in the object management layer. Each of the subsystems contains its own buffer. Due to the two different formats, the database architecture is called a *two-level architecture* (see Figure 1).

*OOPSLA '95 Workshop "Object Database Behavior, Benchmarks, and Performance", Austin, Texas, October 1995.

†This work was partially performed during author's stay at the Database Technology Research Group, ComputerScience Department, University of Zurich, Switzerland.

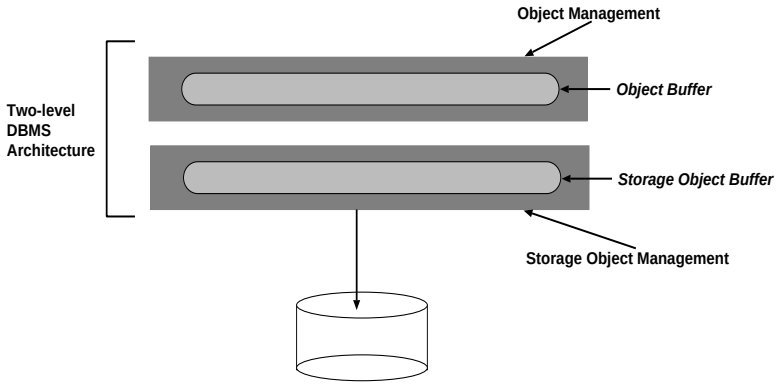


Figure 1: Two-level DBMS Architecture

Before an object can be used by an application it has to be loaded from secondary storage, and must be converted to an in-memory format. This task is accomplished in the following manner: the object manager analyzes which storage objects are needed and requests them from the storage manager. The storage manager loads the storage objects from secondary storage and keeps them in its local buffer. The object manager reconstructs the object structure in its own buffer and reads the objects' values from the storage objects. At the end of a processing, i.e. mostly a transaction, the objects' values are written back to the corresponding storage objects which are saved to secondary storage.

A new aspect is introduced by the conversion of complex objects. Complex objects have (m:n) relationships to other objects, and thus, form a network of objects referring to each other. Relationships between objects are represented by *references*. To access a referenced object, the object manager analyzes where the corresponding object is located in the object buffer (if the object is present), and provides access to the object's values. If the referenced object is not present, the object manager requests the object's storage object from the storage manager and reconstructs the object in its own buffer to provide access to the object's values. Accesses to objects via references are a very common operation in OODBS.

To accelerate access in main memory, the following optimization can be used: object references in complex objects are replaced by the memory address of the corresponding objects in the object buffer, thereby providing fast and direct access to related objects. Naturally, the replacement process is time consuming but provides improved performance if a complex object is accessed over a longer time period. The process of pointer replacement is called *pointer swizzling* [6]. Note, that pointer swizzling is also performed in OODBS that employ a single-level storage architecture [5, 7]. In these OODBS, only one buffer exist, and the object format corresponds to the storage format. When an object is brought into main memory, the pointers *within* the object copy are replaced by memory pointers and thus, have to be overwritten with main memory independent references at the end of processing. In the remainder of this paper, we focus on two-level OODBS architectures.

Pointer swizzling strategies for two-level OODBS architectures have been investigated in recent years as a primary technique to speed up object access in OODBS. All known pointer swizzling strategies [3, 4, 6, 8] rely on the same basic principle: references to objects are represented by the object identifiers (OID) of the corresponding objects. OIDs are typically used to identify objects

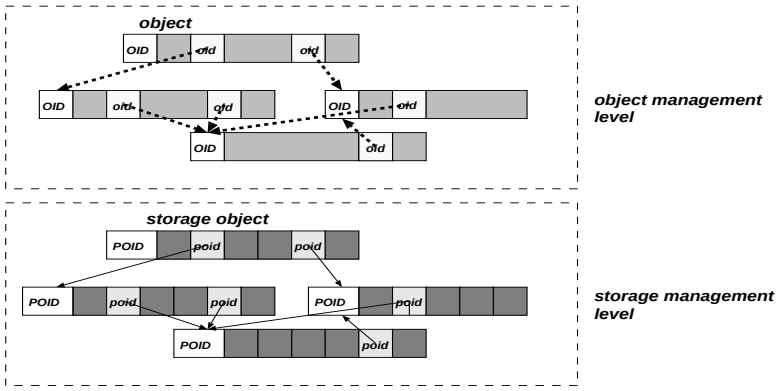


Figure 2: Mapping of Complex Object Structure to the Storage Level

at the object management level. These OIDs are stored as values in the storage objects, and are interpreted by the object management layer to locate the corresponding objects or their storage objects if the objects are not currently present in main memory. While pointer swizzling speeds up object access in main memory, the pointer swizzling process itself contains many processing steps imposing an additional computational burden on object conversion. A reduction of these processing steps improves overall OODBS performance by accelerating object conversion and providing fast in-memory access to referenced objects.

We introduce a new kind of pointer swizzling strategy which represents references between objects by the *storage object identifier* of the corresponding objects. We claim that by using storage object identifiers the number of conversion steps can be reduced significantly, and thus, leads to a major performance improvement. Since a storage object identifier is used to identify a so-called "physical object" in a DBMS (as opposed to the "logical objects" on the object management level), we call the pointer swizzling process *physical pointer swizzling*.

The remainder of this paper is organized as follows. Section 2 presents the physical pointer swizzling strategy in detail. In section 3, we present the organization of the OO7 benchmark for performance tests on the physical pointer swizzling strategy, and the results of the benchmark. Section 4 contains our conclusions.

2 Physical Pointer Swizzling

The physical pointer swizzling strategy relies on the assumption that the object structure of a complex object is also mapped onto the storage level, i.e. each logical object at the object level is represented by a storage object, and each reference to an object is represented by the storage object identifier of the corresponding storage object. This situation is depicted in Figure 2. The storage object identifiers are used for the physical pointer swizzling strategy.

The following steps are executed to load a complex object and to perform the physical pointer swizzling strategy (see Figure 3). The object manager first reconstructs the root of the complex object in its in-memory format. For each reference, the object manager reads the storage object identifier in the storage object and checks to see if the corresponding object already exists in the

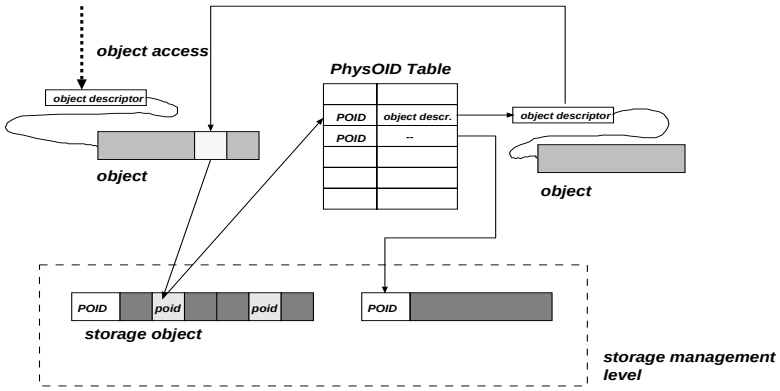


Figure 3: Two-level DBMS Architecture

object buffer. If this is the case, the memory address of the object is assigned as the reference value to the root object in the object buffer. Note, that the reference in the storage object is not replaced with an in-memory pointer. If the referenced object does not exist in the object buffer, the object manager *directly* uses the storage object identifier to access the storage object in the storage object buffer, and reconstructs the object in its buffer. The memory address of the newly rebuilt object is then assigned as the reference value to the original object. This terminates the conversion cycle for the reference.

The same conversion cycle is applied to the other references of the root object, and the references of its subobjects. At the end of the complex object processing (i.e. at the end of the transaction), the complex object is written back to disk. It has to be checked which references changed their value during the object processing, and if new references were inserted or existing ones deleted. The storage object identifiers of these objects are written to the storage objects. All other references still exist in their persistent format in the storage object and thus, the storage objects can be directly be written back to disk.

The strategy above shows that by using storage object identifiers the number of processing steps of pointer swizzling can be reduced significantly since storage objects can be loaded directly from secondary storage and table look-up operations are avoided. The reduction of a look-up operation per referenced object may appear as a small improvement, but as a complex object can consist of several hundred subobjects the conversion process for the complex object is reduced by several hundred operations.

For the basic concept of the physical pointer swizzling strategy, we assumed that the complex object is loaded and converted in its entirety. However, loading a complete complex object is not necessary for most operations. Therefore, we introduce some additional optimizations to the basic pointer swizzling strategy.

Partial exchange

For the execution of operations like navigation through complex objects, the processing is sped-up if only the parts of a complex object that are actually necessary are converted. This avoids expensive conversion and loading operations. There are two ways to support partial exchange. First, only the object parts and references satisfying a query are converted.

Second, references are converted when they are first accessed (lazy pointer swizzling [6]). In both cases, the object manager has to be aware that two kinds of references exist, i.e. swizzled and non-swizzled ones. Before every object access, the pointer's type must be checked.

Indirect exchange

When replacing persistent references with main-memory pointers to the corresponding objects, it must be guaranteed that all references contain valid memory locations. This solution has some disadvantages. For example, if an object is relocated in main memory all in-memory references to the object must be updated. This disadvantage can be avoided if a persistent reference is replaced by a *object descriptor* which has a fixed main memory address and contains the actual memory address when the object exists in main memory. Otherwise, it contains a flag indicating that the object is not present. Object descriptors simplify the management of references since the object manager only needs to deal with one type of pointer. Objects can be relocated in main memory without affecting references to them. Additionally, there is no need to deswizzle all references to an object, if an object is saved to secondary storage. Exchanging persistent references with object descriptors is called *indirect pointer swizzling* [6]. Due to the indirection, this kind of pointer swizzling is slightly slower than using direct in-memory pointers, but combined with partial loading, partial conversion, and saving back objects during a long transaction, it can used to significantly optimize the pointer swizzling process.

3 Performance Analysis

In this section, we present a subset of the benchmark we performed to analyze the influence of the physical pointer strategy compared to the logical pointer swizzling strategy on processing complex objects in OODBS.

3.1 Measurement Approach

For modelling complex objects, we used the OO7 benchmark [2]. We constructed the small and medium database according to the benchmark parameters shown in Table 1. These parameters correspond, with the exception to the number of atomic parts per composite part in the medium sized database and the number of base assemblies used, to the original benchmark parameters.

The OO7 operations

We used a subset of the OO7 benchmark operations, i.e. the traversal operations T1 and T2. Both operations traverse a complex object hierarchy and visit every subobject of a complex object. In T1, a *DoNothing()* operation is performed on every subobject while in T2 simple operations on subobjects are executed.

In both test operations, a complex object is loaded from disk, and the pointer swizzling strategies are performed. T1 is run in a cold and a warm mode, i.e. the buffer cache was flushed between runs in the cold mode while it was not flushed in warm mode. T2 is just run cold.

Parameter	Small	Medium
NumAtomicPerComp	20	100
NumConnPerAtmoic	3/6/9	3/6/9
NumCompPerAssm	3	3
NumAssmPerAssm	3	3
NumAssmLevels	2	2
NumModules	0	0

Table 1: OO7 Benchmark Parameters

Testing Pointer Swizzling Strategies with the OO7 Benchmark

To test the influence of the pointer swizzling strategy on the object conversion process, we used the EOS storage manager [1] as the basic storage system. On top of EOS, we simulated the higher-level DBMS components that perform the pointer swizzling strategy. Therefore, we implemented a rudimentary object manager that managed object identifiers, data model objects and activated objects. The object manager kept the information about logical objects, their OIDs and related storage objects in a hash table. Activated objects were managed in a list. While this approach would not be used in a real-world system, it did not influences the benchmark results since it imposed the same load on both the physical and logical pointer swizzling strategy. Beside the object loading and reconstruction strategies, the object manager was implemented identically for both the physical and logical pointer swizzling strategy.

3.2 Results

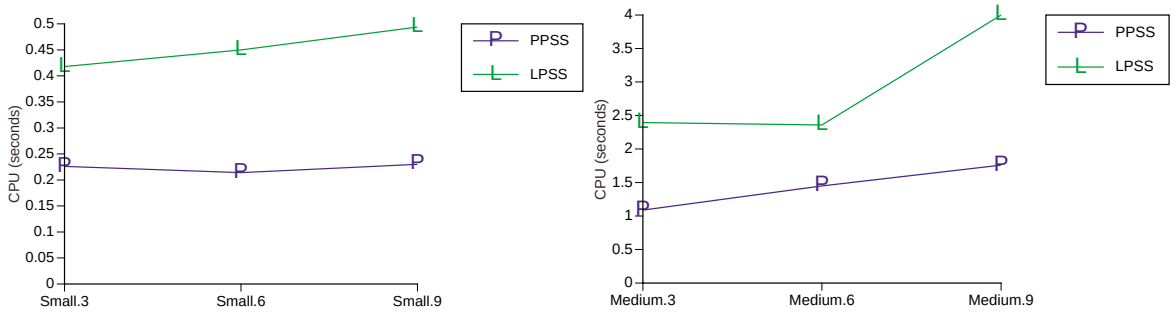
The testbed configuration consisted of a Sun Sparc 20 Model 71 running Solaris 2.4 and a Sparc 10 Model 30 running SunOS 4.1.3. The Sparc 20 was used as the server machine and was configured with 128 MByte main memory, an internal 1.05 GByte (128 MByte swap) disk drive and an external 10 GByte (203 MByte swap) disk drive. On the larger disk, a directory was used as a raw disk device to store the databases. The smaller disk contained the system software, the swap space and the log information of the recovery management.

The Sun Sparc 10 was used as the client machine and was equipped with 32 MByte of main memory, an internal 424 MByte (128 MByte swap) disk drive and two 2.1 GByte external disk drives. The internal disk drive contained the system software.

All tests were run in single user mode. The CPU time taken to perform a transaction served as performance measure. All times are in seconds.

3.2.1 *Traversal T1*

The *T1 traversal* was used to measure the effect of the *physical pointer swizzling strategy* (PPSS) on the conversion of complex objects from secondary storage format to main-memory format and in



(a) (b)

Figure 4: Benchmark results for T1, cold: (a) small database; (b) medium database

the opposite direction in a two-level OODBS with two buffers. Using the PPSS, references between related objects were represented by the physical object identifiers of the corresponding storage objects. These were directly used for loading relevant storage objects from disk. In the logical pointer swizzling strategy (LPSS), references were represented by the logical OIDs of the related objects. These logical OIDs were used to look-up, load, and access the corresponding storage objects. The results of T1 run in cold mode are shown in Figures 4(a-b).

In the T1 traversal, the use of the PPSS is clearly superior to the LPSS in loading, converting and saving complex objects. In the small database, the PPSS is 2 times faster than the LPSS (PPSS 0.226 - 0.23 sec, LPSS 0.418 - 0.494 sec). The performance gains are due to the fact that look-up operations to analyze which storage objects have to be loaded are avoided. Even if the look-up function was organized as a hash table as in our test implementation, the time consumed for the look-up amounts to a significant factor if a complex object contains a large number of subobjects to be accessed. In the small database, around 130 look-up operations are performed each of which took around 0.0008 seconds on average. Thus, for the pure hash-table look-up 0.1 seconds were consumed, i.e. a fourth of the overall processing time for the LPSS test runs. In the medium sized database, the advantage of the PPSS is even more evident. In this case, the PPSS is about 2.5 times faster than the conventional approach (PPSS 1.088 - 1.76 sec, LPSS 2.394 - 4.00 sec). This results is due to the fact that the hash table size increases with the database size, and thus, look-up operations take longer, and more look-up operations have to be performed since the complex object is larger.

For the small database, T1 was also run in warm mode, i.e. the buffer were not flushed between the runs. Here, the PPSS is about 2.6 times faster than the LPSS (PPSS 0.071 - 0.106 sec, LPSS 0.183 - 0.242 sec) which is a higher factor than in the cold run. The results are shown in Figure 5.

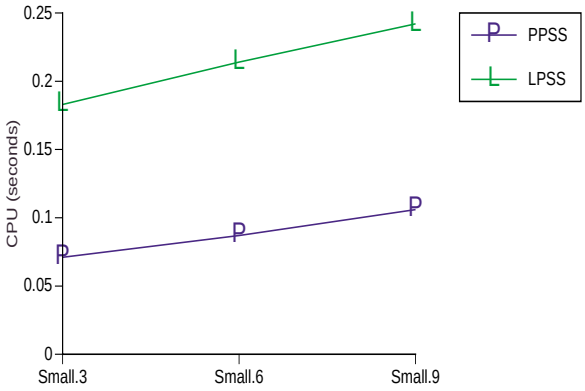
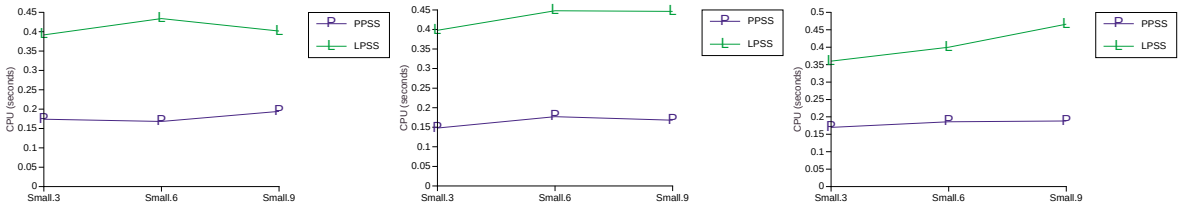


Figure 5: Benchmark results for T1, small db, warm



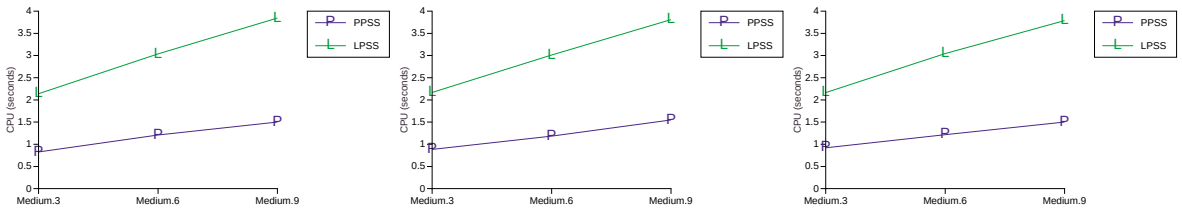
(a) (b) (c)

Figure 6: Benchmark results for T2abc, small database

3.2.2 *Traversal T2*

In the T2 traversal, we measured the effect of the PPSS on object loading, converting, updating and saving operations. The execution steps for the loading-processing-saving cycle are performed as described in T1. Additionally, update operations are performed on the objects before they are written back to secondary store. The results of T2 a, b, and c for the small database are presented in Figure 6.

T2 for the small database shows that the execution time is almost independent to the number of updated operations performed on converted objects in in-memory format. Almost 100 percent of the execution time is spent on the loading-conversion-saving cycle for objects. Due to this fact, T2 demonstrates again that the PPSS is about 2.5 times faster than the traditional LPSS. The test runs for the medium database confirm this fact. The results of T2 a, b, and c for the medium database are presented in Figure 7.



(a) (b) (c)

Figure 7: Benchmark results for T2abc, medium database

3.2.3 Summary of Results

While performing the benchmark, we made several observations on the effect of the physical pointer swizzling strategy on OODBS:

- In the load-process-save cycle of complex objects, almost 100 percent of the processing time is consumed by loading and writing complex objects from and to secondary storage. The processing times are almost independent of the number of updated objects and the number of update operations. This demonstrates that it is essential to apply optimized loading and conversion strategies in order to improve the performance of processing complex objects.
- We demonstrated by applying the physical pointer swizzling strategy which is based on storage object identifiers for representing references within complex objects on the storage level, the number of processing steps during pointer swizzling is reduced, and look-up operations for identifying which storage objects need to be loaded and/or accessed are avoided. (Note, that look-up operations are independent from object clustering issues.) Measurements showed that even in the case of optimized table look-ups, the time consumed for all look-ups on a complex object takes around one fourth of the overall processing time in the load-process-save cycle. We demonstrated that by avoiding these operations and applying the PPSS, the processing time to convert complex object is reduced by a significant factor of 25 - 50 percent.

4 Conclusion

In this paper, we presented the concept of the *physical pointer swizzling strategy* which is designed to efficiently convert references in complex objects from a main memory independent format to in-memory pointers and vice versa in a two-level OODBS architecture. In the physical pointer swizzling strategy, references are represented by the *storage object identifiers* in the main memory independent format. This reduces the number of conversion steps from the main-memory independent format to in-memory pointers since the storage object identifiers are *directly* used to further access and/or load storage objects that store the data of the referenced objects. Thus, expensive table look-up operations to identify relevant storage objects are avoided. Benchmark results

presented demonstrate that the physical pointer swizzling strategy accelerates the conversion of complex objects in a OODBS by a factor of 2 - 2.5.

Acknowledgements

I want to thank Edmond Mesrobian and the anonymous referees for their helpful comments that significantly improved the presentation of the paper.

References

[1] A. Birilis. An Efficient Database Storage Structure for Large Dynamic Objects. In *Eighth International Conference on Data Engineering, Tempe, Arizona*, pages 301–308. IEEE Computer Society, 1992.

[2] M.J. Carey, D.J. DeWitt, and J.F Naughton. The OO7 Benchmark. Technical Report Technical Report, Computer Science Department, University of Wisconsin-Madison, 1994.

[3] A. Kemper and D. Kossmann. Adaptable Pointer Swizzling Strategies in Object Bases. In *Ninth International Conference on Data Engineering, Vienna, Austria*, pages 155–162. IEEE Computer Society, 1993.

[4] W. Kim, J.F. Garza, N. Ballou, and D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Transactions on Knowledge and Data Engineering*, 2(1):109–123, 1990.

[5] C. Lamb, G. Landis, J. Orenstein, and D. Weinreb. The Objectstore Database System. *Communications of the ACM*, 34(10):50–63, 1991.

[6] J. Eliot B. Moss. Working with Persistent Objects: To Swizzle or Not to Swizzle. *IEEE Transactions on Software Engineering*, 18(8):657–673, 1992.

[7] V. Singhal, S.V. Kakkad, and P.R. Wilson. Texas: An Efficient, Portable Persistent Store. In *Fifth Int'l Workshop on Persistent Object Systems, San Miniato, Italy*, 1992.

[8] S.J. White and D.J. DeWitt. A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies. In *Eighteenth International Conference on Very Large Databases*, pages 419–431. IEEE Computer Society, 1992.