Dual-Buffering Strategies in Object Bases

Alfons Kemper[†] Donald Kossmann[‡]

[†]Lehrstuhl für Informatik
Universität Passau
D–94030 Passau, F. R. G.

[‡]Graduiertenkolleg "Informatik und Technik"
RWTH Aachen
D–52056 Aachen, F. R. G.

Tel. (++49-851) 509-321
FAX (++49-851) 509-771
{ *kemper* | *kossmann* }*@db.fmi.uni-passau.de*

Abstract

In this work, we devise and evaluate control strategies for combining two potentially powerful buffer management techniques in object bases: (1) buffer pool segmentation with segment-specific replacement criteria and (2) dual buffering consisting of copying objects from pages into object buffers. We distinguish two dimensions for exerting control on the buffer pool: (1) the *copying* time determines when objects are copied from their memory-resident home page and (2) the *relocation* time determines the occasion on which a (copied) object is transferred back into its home page. Along both dimensions, we distinguish an *eager* and a *lazy* strategy. Our extensive experimental results indicate that a lazy object copying combined with an eager relocation strategy is almost always superior and significantly outperforms page-based buffering in most applications.

1 Introduction

In the Eighties, object-oriented database systems emerged as the potential next-generation database technology. However, now that the initial "hype" has vanished the question is whether or not they can actually conquer a substantial share of the information technology market. In our opinion, one of the key issues will be their performance.

In this work, we address the particularly crucial issue of main-memory buffer management in optimizing object-oriented database systems. Buffer management has been studied intensively by both researchers of operating systems [CD73, Den80] and database management systems [EH84, OOW93]—to list just a few. In the relational context buffer pool segmentation has been studied. Segmenting the buffer pool provides the flexibility to customize the replacement strategies for particular reference patterns [SS86, CD85, NFS91]. This is even more important in object-oriented database applications which tend to be computationally more complex and, therefore, exhibit an even larger variety of different reference patterns. Consequently, in our work we adopted the segmented buffer pool organization as used by the DBMIN algorithm [CD85].

Most of the work on buffer management assumes page-based buffering which is consequently incorporated in most commercial object-oriented database systems, e.g., ObjectStore [LLOW91]. However, in this work we show that in the object-oriented model

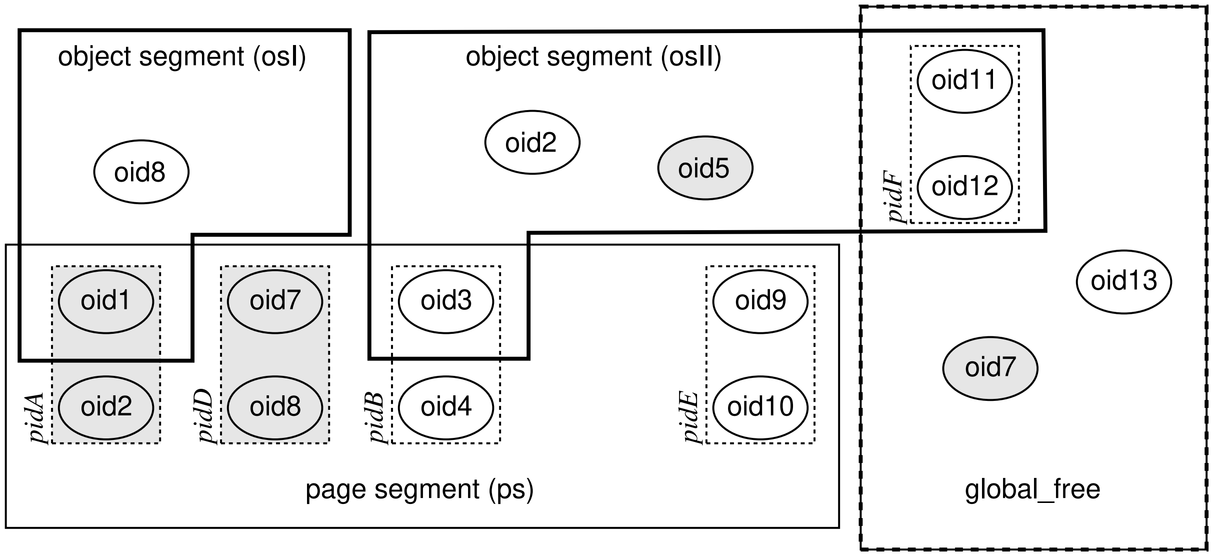

Figure 1: Snapshot of a Dual-Buffer Pool

the flexibility of dual buffering [KBC+88, KGBW90] should be exploited which allows to buffer entire pages as well as isolated objects. This way it is possible to buffer well-clustered pages that inhabit many application-relevant objects and at the same time to extract (isolate) objects from otherwise useless pages.

The contribution of this work consists of devising and evaluating control strategies for combining these two potentially powerful buffer management techniques in object bases: (1) buffer pool segmentation with segment-specific replacement criteria and (2) dual buffering consisting of copying objects from pages into object buffers or leaving well-clustered pages intact. There are two dimensions for exerting control on the buffer pool: (1) the *copying* time determines when objects are copied from their memory-resident home page and (2) the *relocation* time determines the occasion on which a (copied) object is transferred back into its home page. Along both dimensions, we devised an *eager* and a *lazy* strategy. To assess the different control strategies, we built an experimental system consisting of a page server [DFMV90] connected with clients managing a segmented dual-buffer pool. Our extensive experimental results indicate that a lazy object copying combined with an eager relocation strategy is almost always superior and substantially outperforms pure page-based buffer management in most applications.

The rest of the paper is organized as follows. In Section 2 the basic concepts of dual buffering in a segmented buffer pool are sketched. In Section 3 the various control schemes for managing a segmented dual-buffer pool are classified. Implementation issues are discussed in Section 4. A quantitative assessment follows: Section 5 describes our experimentation platform; in Section 6, the results of the benchmarks are visualized and discussed. Section 7 concludes this paper.

2 Dual-Buffer Management

As stated above, work on database buffer management in the relational context indicates that a segmented buffer pool outperforms a global buffer pool. Segmentation of the buffer pool allows to use dedicated replacement strategies for different segments of the pool; thereby, tuning buffer management for specific reference patterns as reported, e.g., in [CD85]. In an object-oriented database system the (segmented) buffer could be organized to hold either pages or objects—generalized to *granules* in the following.

A pure *page-based* buffer organization only allows the maintenance of entire pages—as they are stored on secondary storage. This scheme potentially suffers from bad buffer utilization if the object base is poorly clustered with respect to the corresponding application. The other extreme is a pure *object-based* organization in which all objects are extracted from their home page, i.e., the page on which they are stored on secondary storage. This organization may induce an unnecessarily high overhead for well-clustered pages, i.e., those pages that contain many relevant objects.

In this paper, we show that buffer segmentation should be combined with *dual buffering*, which allows to maintain entire pages as well as (isolated) objects in the buffer. Under dual buffering, the buffer pool could be segmented into page-based and object-based segments; each of them with a dedicated replacement policy. The dual-buffer organization allows to leave well-clustered pages in their entirety in a page-based segment while objects located on otherwise "useless" pages can be extracted and copied into an object-based segment.

Fig. 1 depicts a snapshot of a dual-buffer pool. This buffer is segmented into four segments: two purely object-based segments *osI* and *osII*, one page-based segment *ps*, and a special segment *global_free* that is automatically allocated for every buffer pool and contains both pages and objects. Every granule is contained in at most one segment that keeps the usage statistics of the granule. For example, the usage statistics for object *oid1* are kept in segment *osI* and for object *oid2* in *osII* while the usage statistics of their home page *pidA* are independently maintained by segment *ps*. The usage statistics of an object are updated every time an application accesses the object; the usage statistics of a page are updated when an object that is located in the page is accessed for the first time and the page is accessed to localize this object.

If a granule "leaves" the locality set of a page- or object-based segment, it is transferred into the segment *global_free*. The segment *global_free* is used to maintain granules that are subject for (immediate) displacement as soon as buffer space is needed.

The graphic indicates that page- and object-based segments may overlap. For example, the object *oid1* is contained in *osI* and, at the same time, its home page *pidA* in which it is still located is contained in the page-based segment *ps*. The overlapping of page- and object-based segments reduces copying overhead and increases buffer utilization. For example, objects *oid2* and *oid3* can be accessed equally in segment *osII*. Object *oid2*, however, was copied and is buffered twice: once within its resident home page *pidA* and once in the extra copy; in this case, the (useless) copy of object *oid2* is called a *duplicate*.

Shaded granules in Fig. 1 indicate a modification that needs to be flushed upon displacement. Flushing a (shaded) object whose home page is not currently memory-resident requires to bring in this page, copy the object into the page and then flush the page.

In a client-server system [DFMV90], dual buffering can be incorporated in several ways. Kim et al. [KBC$^+$88] investigated an object-server architecture in which the server buffers pages and "ships" objects into the client's object-oriented workspace. Dual buffering could also be embedded within an object server; i.e., the object server buffers objects and pages to reduce the number of page faults in the server. We, however, concentrated on a page-server architecture since this architecture appears to be used in most of today's object base systems [Win93]. In our prototype, a dual-buffer pool is located in the (diskless) client

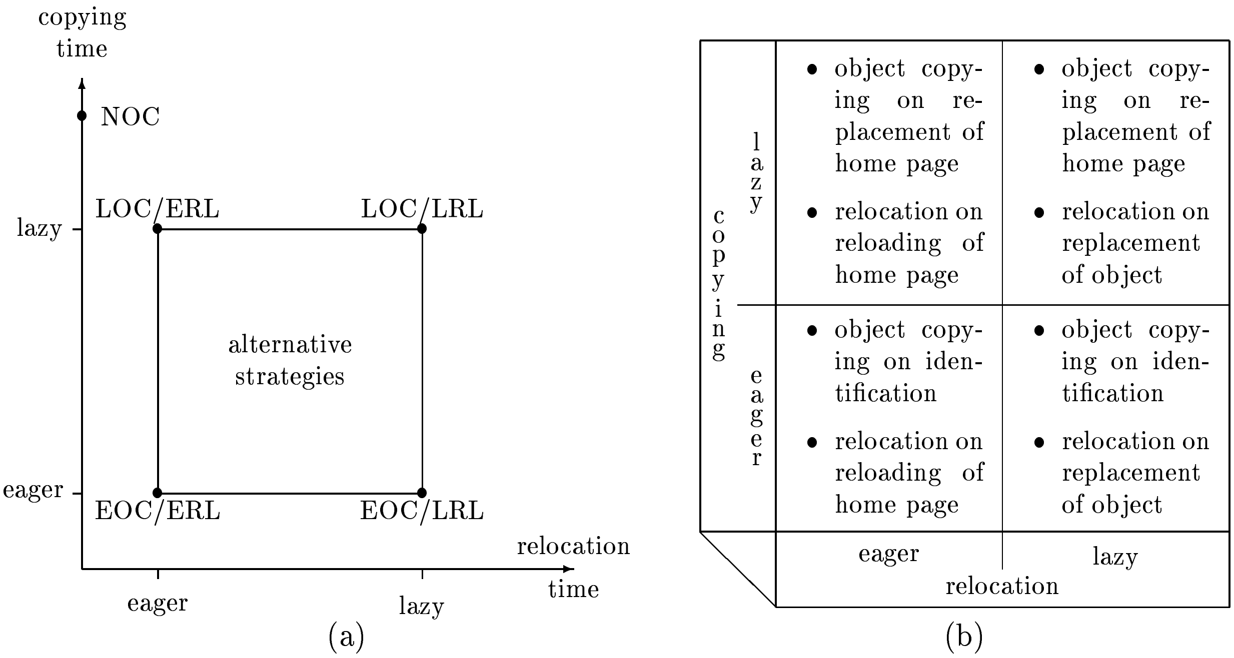

Figure 2: Classification Scheme

workstation to minimize the communication between client and server, and an (ordinary) page-buffer pool is located in the server.

3 Classification of Dual-Buffering Strategies

There are two dimensions that have to be taken into account by effective dual-buffering control strategies:

- *copying time*: This determines at what time—or more precisely, upon what event—an object is copied from its memory-resident home page into an object segment of the buffer pool.

- *relocation time*: This control dimension determines at what time an object copy—previously extracted from its home page—is "given up" and, if necessary because of modification, transferred back into its memory-resident home page.

In Fig. 2a these two control dimensions are sketched as time axes. For both dimensions two extremes are distinguished, namely an *eager* and a *lazy* strategy. Combining the two alternatives of either dimension renders a strategy space for dual-buffer management whose "corner stones" are indicated by bullets—their control strategies are outlined in Fig. 2b. Furthermore, the control strategy NOC under which no object copying takes place is classified as a bullet at the top of the copying time axis; since no object copying takes place, no relocation is required.

3.1 Object Copying

3.1.1 Eager Object Copying (EOC)

Under eager object copying (EOC), an object is extracted from its home page and copied into an object segment when it is accessed for the first time; i.e., upon object fault. Thus, access to objects is always carried out on copies. Under EOC, a buffer pool should be configured such that the object segments are large compared to the page segments in order to avoid wasting precious buffer space for idle pages.

If configured well, EOC will often reduce the number of page faults as compared to buffering pages only (NOC). Therefore, derivatives of EOC are used in ORION [KBC+88, KGBW90], its commercial successor Itasca [Inc93], and in the E Persistent Virtual Machine (EPVM 2.0) [WD92]. EOC, however, faces significant drawbacks for particular application profiles. For example, in applications that browse sequentially through a large volume of data, the number of page faults principally cannot be reduced. For these applications, EOC copies objects unnecessarily, thereby, inducing considerable CPU overhead to allocate main memory in the buffer pool for the copies. Even worse, if objects are modified in such a sequential scan, EOC may cause a tremendous number of additional page faults if the home pages are displaced before the modified objects are flushed.

Furthermore, EOC exhibits poor performance if the object base is clustered very well. In this case, it is advantageous to buffer on a per page basis since the clustering algorithm considered already the application's reference pattern [TN91, GKKM93]. For well-clustered object bases, EOC will suffer from the copying effort because the object extraction cannot significantly increase the buffer utilization.

To reduce the drawbacks of EOC but still allow buffering on a per object basis—whenever useful to enhance buffer utilization—we devised lazy object copying (LOC).

3.1.2 Lazy Object Copying (LOC)

The idea is to copy the objects as late as possible; i.e., objects are only copied when their home page is displaced from the buffer pool and they still are member of the locality set of an object segment.

For sequential scans, the best case of NOC, LOC performs equally well because no object is copied since in a sequential reference pattern, pages are accessed more often than objects and thus, the objects drop out of the locality set of their object segment before their home page is displaced. On the other hand, if an application repeatedly accesses a set of objects that are spread over a large number of pages, the best case of EOC, LOC will copy objects as necessary to achieve a good buffer utilization.

Since copying of objects takes place at the time a page fault occurs (i.e., when buffer space is allocated for the new page), the copying of objects is overlapped with the time the client waits for the new page. On the other hand, under EOC objects are copied just after a page was loaded and I/O waiting times cannot be exploited for object copying.

If objects are modified, however, LOC will not always have optimal performance. To describe the behavior of LOC in this case, *early* (i.e., while the home page is still resident) and *late* (i.e., after the displacement of the home page) updates of objects are distinguished. If only early updates are carried out, pure LOC is fine: the modifications are committed to the database when the page is displaced and afterwards no modifications need to be flushed anymore. However, if early and late updates are carried out, LOC will flush the home page twice: once when the home page is first displaced to commit the early updates and once again to flush the late updates. In comparison, EOC avoids the flushing of early updates, thereby, possibly incurring an additional page fault at a later time.

To deal with early and late updates, two derivatives of pure LOC were devised and evaluated. LOC^- precludes the displacement of pages as long as the page contains modified objects that belong to the locality set of an object segment; i.e., objects are not copied from modified pages. The alternative, LOC^+, copies objects as soon as they are modified for the first time, rather than waiting until the page is displaced; thus, LOC^+—like EOC—avoids the cost of flushing the home page for the first time to commit the early modifications.

3.2 Object Relocation

The relocation control has to deal with the event that an object is extracted from its home page, the home page is displaced from the buffer pool but eventually brought back into the buffer—referred to as reloading in Fig. 2b.

3.2.1 Eager Relocation (ERL)

Under eager relocation (ERL) an object is relocated regardless whether or not it was modified as soon as its home page is brought back into the buffer pool. For example, if the home page of object *oid5* is loaded into segment *ps* in Fig. 1 (e.g., to access object *oid6* that is not resident), ERL flushes the modifications of object *oid5*, marks the page as modified, and gives up the copy of object *oid5* so that segments *osII* and *ps* overlap at the location of object *oid5*.

In combination with LOC, ERL completely eliminates duplicates; a copy of an object only exists if the home page is not resident. Furthermore, the number of page faults due to flushing modified objects is minimized.

On the other hand however, following ERL, some objects are relocated unnecessarily; e.g., if the home page of object *oid5* is loaded and immediately displaced again, object *oid5* is relocated and subsequently copied in vain. Unnecessary relocation causes CPU overhead and unnecessary flushing of pages to the database if an object is modified, relocated, copied and finally, modified again.

The combination of ERL and EOC appears to follow two contradictory policies: on the one hand, access should always be carried out on copies (EOC), and on the other hand, copies are supposed to be given up as early as possible (ERL). For this reason, we safely disregarded the EOC/ERL strategy in our quantitative evaluation (cf. Section 6). EOC/ERL, however, represents a "corner stone" in the full range of possible dual-buffering strategies and therefore, is depicted in Fig. 2.

3.2.2 Lazy Relocation (LRL)

Lazy relocation (LRL) relocates objects as late as possible. Once a copy of an object is established, it will not be given up before the object drops out of the locality set of its object segment and is chosen as a victim in the segment *global_free* in times of scarce buffer space.

Whereas ERL is a novel technique, LRL in conjunction with eager object copying has already been devised in ORION [KGBW90]. In Exodus, also a *late* relocation scheme is used [WD92]. The objects are copied into virtual memory where they remain until an application commits. Shortage in physical main memory is overcome by the swapping facility of the operating system and relocation is carried out collectively for all cached objects at commit-time of an application.

Obviously, LRL avoids unnecessary relocation of objects. On the other hand, LRL might miss the last chance to flush a modified object without inducing a page fault. In addition, buffer utilization is generally lower because of the duplicates incurred when the home page of extracted objects is reloaded into the buffer to access some other object.

3.2.3 Fine-Tuning of Relocation Strategies

As indicated in Fig. 2a, there are many possible variations of relocation strategies. The performance of object relocation techniques strongly depends on object modifications. We devised two further strategies as compromise between LRL and ERL:

1. relocate non-modified objects eagerly and modified objects lazily

2. relocate modified objects eagerly and non-modified objects lazily

Although one is the exact opposite of the other, both make sense. Under the first strategy the number of times a page is flushed to the database is reduced because only pages containing at least one modified object need to be flushed upon displacement from the buffer pool. The second strategy reduces the number of page faults that could occur because of flushing an object whose home page was displaced.

4 Implementation Issues

In this section, some of the specifics of implementing dual buffering are addressed.

4.1 Object and Page Descriptors

An application can (directly) access any object that is in the locality set of any object-based segment, regardless whether the object has been copied or not. To this end, an *object descriptor* is allocated for all these objects and registered in a hash table, the so-called resident object table ROT. Fig. 3 shows the ROT and some descriptors for the state of the dual-buffer pool depicted in Fig. 1. An object descriptor contains a variety of information; e.g., in the *ref* field the main memory address of the resident object is materialized to carry out the access or for pointer swizzling [KK93], the *cop* field indicates whether the object has been copied, the *mod* field says whether the object has to be flushed before it may be evicted because it has been copied and modified, and the *seg* field refers to the buffer segment the object is located in so that usage statistics can be maintained after the object has been accessed. A comprehensive description of object descriptors and additional information they keep (e.g., for locking) is given in [Kim90].

If an application intends to access an object that is not in the locality set of an object-based segment, an object fault is induced, and the object's home page has to be accessed. To this end, a *page descriptor* is allocated for every resident page that is in the locality

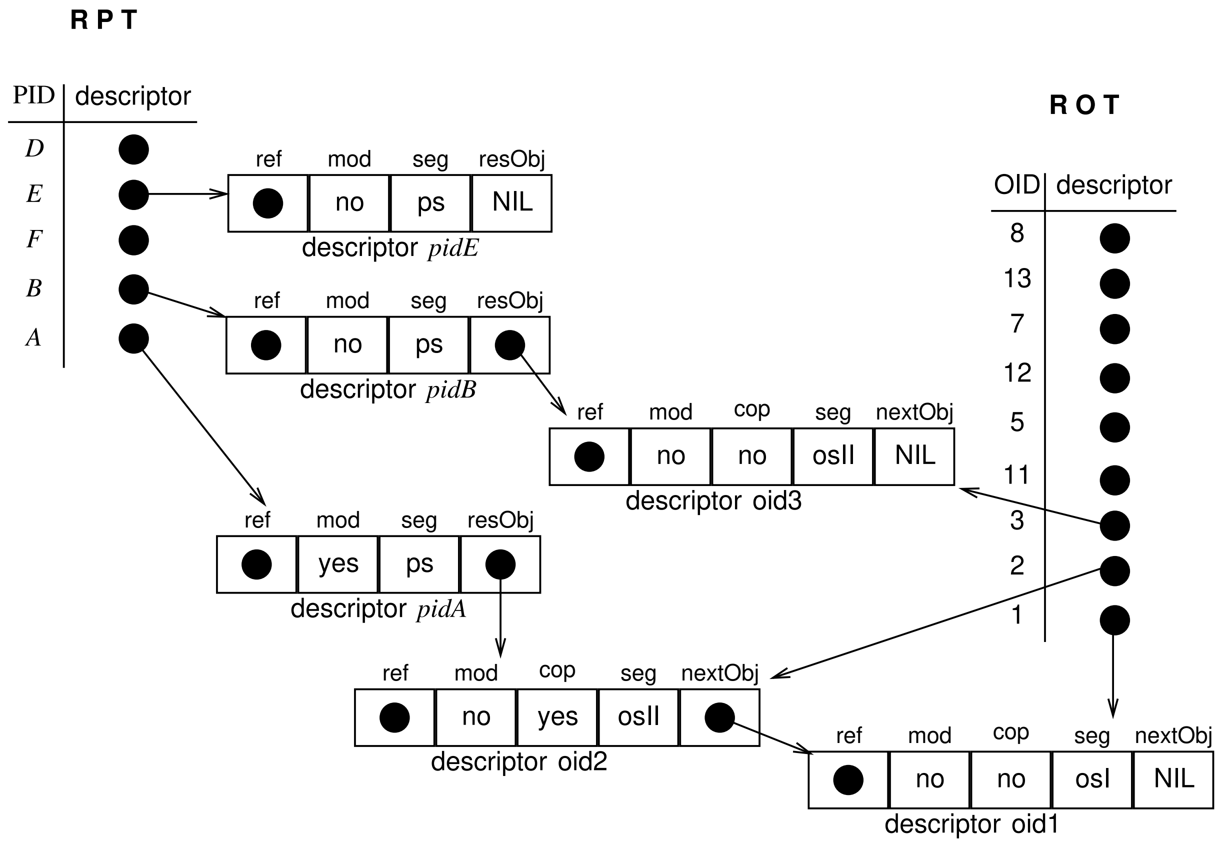

Figure 3: ROT, RPT, Object and Page Descriptors (cf. Fig. 1)

set of a page-based segment, and the page descriptors are registered in the resident page table RPT. Again, a page descriptor keeps all the information to determine the state of a page in the buffer pool.

Fig. 3 illustrates that the descriptors of objects that are located in the same home page are chained to a list via the *nextObj* field and that the descriptor of the home page is the anchor to this list via the *resObj* field. A similar mechanism has been devised in [WD92]. LOC makes use of this chaining to find all the objects that have to be copied when a page is displaced. That is, the chaining allows to efficiently find those objects that belong to the locality set of some object segment, have not been copied yet, and are located in the page to be replaced. Furthermore, when a modified page is flushed to the server, all the modified and copied objects whose descriptor is chained in this list are flushed as well—regardless which relocation policy is used—if the recovery algorithm allows this, i.e., a STEAL policy is used.

For ERL, all the resident and copied objects located in a page have to be identified when the page is reloaded. Consequently, the descriptor of a page, i.e, the anchor of the list to find these objects, has to be kept even if the page has been displaced. The descriptor of a displaced page can be garbage collected when all copies of objects located in this page have been given up. Following LRL, on the other hand, the page descriptor can be discarded as soon as the page is replaced.

4.2 Main-Memory Allocation

Buffering variable sized objects complicates main memory allocation significantly. In our prototype and thus, in the performance experiments, the client's main memory was managed by a buddy system [Knu73]. Pages were buffered without any off-cuts. As expected, EOC suffered more from memory fragmentation than LOC since LOC buffered more pages. From the point of view of the buddy system, RAM utilization was typically 80% for EOC, 85% for LOC, and always 100% for NOC. Thus, buffer fragmentation was serious for the dual-buffering strategies. As discussed in Section 6, however, from a *higher* point of view, *buffer* utilization was in most cases better for the dual-buffering strategies because objects could be buffered without buffering useless neighboring objects that were located in the same page. Therefore, even though the "gross" RAM utilization is lower, the "net" buffer utilization is typically a lot higher than in a pure page-based buffer.

4.3 Overlapping Ratio

If LOC is followed, object-based and page-based segments, and in particular, object-based segments and the segment *global_free* overlap extensively; e.g., in Fig. 1, segment *osII* and the segment *global_free* overlap for the objects *oid11* and *oid12*. When buffer space is scarce, it is possible that the replacement policy of the segment *global_free* chooses a page as a victim that is very much involved in this overlapment; e.g., page *pidF*. As a result, many objects are copied and buffer space is even scarcer due to the off-cuts caused by the copies.

To remedy this situation, the *overlapping ratio (or)* of a page is defined as the ratio of the total volume of the objects that are located in the page and belong to the locality set of some object segment and have not been copied relative over the size of the page. In Fig. 1, the *or* of page *pidF* is 1, 0.5 for pages *pidA* and *pidB*, and 0 for pages *pidD* and *pidE*. Pages whose *or* exceeds a certain threshold (e.g., 0.6) could be precluded from displacement.

Introducing such a threshold has several advantages: well-clustered pages can be identified, and thus, copying and relocation of objects located on these pages can be avoided. Furthermore, it is possible to preclude the copying of large objects (i.e., objects that are nearly as big as a page) or big portions of very large objects.

In general, an optimal choice of the threshold depends on how well the object base is clustered for a specific application. Our experiments with varying thresholds (not reported in this paper) indicated that usually very good performance is achieved with a threshold of 0.6. In the range between 0.5 and 0.7, the threshold appeared to be not a very critical parameter. It should, however, not drop below 0.4 since then, LOC often degenerated to no-copying (NOC), and it should not be greater than 0.8 to avoid the effect described above (i.e., buffer space becoming even scarcer after the eviction of a page).

5 Benchmark Environment

5.1 Software and Hardware Used

The performance experiments were carried out under SunOS 4.1.3 on two Sun Sparc 10 workstations that were connected by an (isolated) Ethernet. The server machine had a

424 MB disk drive (model Sun0424) and the client machine was diskless, i.e., every page fault in the client's buffer pool induced a request to the server.

The server was a page server developed within the GOM project; in the way we used it, it was very similar to, e.g., the page server of the Exodus system [Gro91]. The server maintained a page-buffer pool that was restricted to 1000 pages at 4096 bytes which was approximately half the size of the object base. Thus, buffer hit ratios in the server were non-trivial.

In the client, objects were accessed on the basis of physical object identifiers that were 12 bytes long and contained the object's permanent address, i.e., *page* and *slot*. In addition, the client contained a dual-buffer pool applying a buffer allocation scheme that is characterized as *local, static, adaptable* by [EH84]. To make the results more comprehensible, only one application was run at a time (single-user mode), and only one page-based and one object-based segment were allocated in addition to the segment *global_free* and their size was configured *as well as possible* given the application's profile and the total size of the client's buffer pool. The page-based as well as the object-based segment were maintained by an LRU replacement policy whereas for the segment *global_free*, a FIFO policy was used.

5.2 Benchmark Specification

The performance experiments were carried out using the *T1* (read-only), *T2a* (few updates), and *T2b* (many updates) traversals of the OO7 benchmark [CDN93]. To determine the response time, every run was repeated three times and the average was taken. In all the experiments reported in this paper, the benchmark was run against the *small* OO7 object base; i.e., an object base with 20 *CompositeParts* and 3 *Connections* per *AtomicPart*. In our implementation, the size of this object base was approximately 2000 pages disregarding the index structures that were not accessed by the traversals. The results scale to large object bases and large buffer pools as only the ratio of the object base's size to the size of the buffer pool is important.

As pointed out before, clustering has a severe effect on the performance of buffer management techniques. Therefore, the following three clusterings were investigated:

Time-of-creation (TOC): The objects were inserted sequentially into the object base in the order they were created by the *gendb* function of the original E/Exodus implementation of the OO7 benchmark. TOC provided a very good but not an optimal clustering.

Type-based (TB): For every type, a (logical) file was created. Objects were included into the corresponding file in the order of their time of creation—again, on the basis of the *gendb* function. For the *T2a* and *T2b* traversals, TB-clustering was particularly amenable since these traversals only modified *AtomicParts*, and thus, all the objects that were modified were clustered together. As a consequence, only a fraction of the referenced pages was modified and needed to be flushed whereas under TOC-clustering almost every page was modified.

RANDOM: Objects were placed randomly into pages.

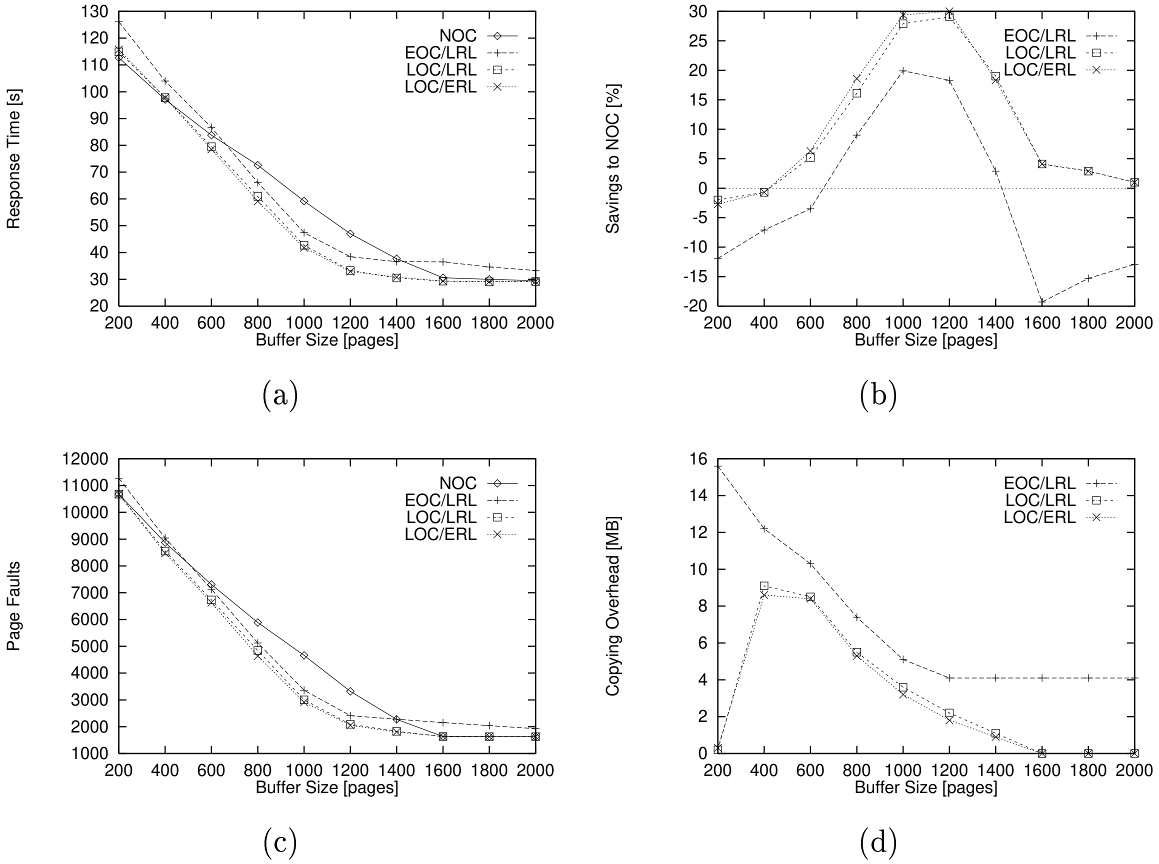

Figure 4: Cold T1 Traversal, TB-clustering

6 Performance Experiments

In this section, the main results of our performance experiments are presented. The novel copying strategy (pure) LOC and its two derivatives LOC^- (avoiding the copying of objects from modified pages) and LOC^+ (copying upon modification) in combination with lazy and eager relocation are evaluated and compared to page-based buffering (NOC) and EOC/LRL. Three different ways to cluster an object base, applications with and without modifications, and running the benchmark in cold and in warm buffers are considered.

6.1 Read-only Applications (T1 Traversal)

Fig. 4a plots against the y-axis the response time for the *cold* T1 traversal in the type-based (TB) clustered object base for varying buffer sizes in the client (x-axis). In this experiment, the client's as well as the server's buffer pool were initially empty. Comparing page-based buffering (NOC) to dual buffering, three cases can be distinguished (cf. Fig. 4b). If the buffer was very small (less than 400 pages), most of the objects were not buffered long enough so that they were replaced before they were accessed a second time. As a consequence, no performance gain was achieved by dual buffering. EOC/LRL was outperformed by more than 10% due to its significant copying overhead (cf. Fig. 4d that depicts the number of megabytes copied by the dual-buffering strategies). On the other hand, the LOC strategies (combined with LRL as well as ERL) only copied very few objects because most objects were evicted from the object buffer's locality set before their home page was replaced. Anyhow, lazy object copying was also slightly outperformed by NOC for maintaining usage statistics on a per object basis. In Fig. 4 and for all the experiments with the T1 traversal, the results of pure LOC also represent those of LOC^- and LOC^+ since these three techniques only differ if updates are carried out.

Compared to NOC, dual buffering was particularly effective if the buffer was large enough to hold a large number of objects of the application's working set and yet too small to hold all the corresponding home pages. In Fig. 4, the range from 800 to 1400 pages can be observed where dual buffering really pays off. In this case, dual buffering under LOC/ERL reduced the number of page faults by up to 40% (cf. Fig. 4c) and the response time by a maximum of 30%. Even in this best case of EOC/LRL, the LOC strategies outperformed EOC/LRL significantly because LOC avoided copying objects from pages that were clustered very well, object copying was overlapped with I/O activity, and buffer utilization was higher.

When the client's buffer was almost as large as the object base (larger than 1600 pages), again, no or only marginal gains were achieved by dual buffering. Hardly any objects were copied under LOC because only few pages had to be replaced from the client's (large) buffer pool and thus, object copying was seldom initiated. In this case, EOC/LRL suffered from (unnecessarily) copying every object that was accessed and from duplicates that induced a considerable number of page faults. As a result, EOC/LRL was outperformed by 20% in the worst case.

For the OO7 benchmark, TB-clustering was very effective. Therefore, the results for lazy and eager relocation under LOC depicted in Fig. 4 did not differ significantly. This is due to the fact that the home page of a resident and copied object was rarely reloaded to access another object that was located in the same page. Because of the good clustering, objects that were located in the same page were copied and then replaced together so that eager relocation seldom took effect. If, however, the object base is poorly clustered, pages have to be reloaded several times before all relevant objects are extracted. In this scenario, eager relocation becomes an important issue to avoid duplicates. To illustrate this effect, Fig. 5a plots the response time of the cold T1 traversal in the randomly clustered OO7 object base. LOC/ERL was consistently the best strategy because it induced the least number of page faults due to its superior buffer utilization (cf. Fig. 5c). Fig. 5d illustrates how extensively LOC/ERL copied and relocated objects; e.g., for a buffer of 800 pages the copying overhead was more than 12 times higher than under LOC/LRL. In these single user experiments, however, the performance drawbacks were not immense since much of the object copying was overlapped with I/O activity.

As expected, in the best case, dual buffering outperformed NOC to a higher rate (more than 60%) in the poorly clustered (RANDOM) object base than in the well-clustered (TB) object base (compare Fig. 4b with 5b). For a buffer of 2000 pages, however, EOC/LRL and LOC/LRL suffered so severely from duplicates that their buffer utilization even dropped below that of NOC and their performance was significantly worse.

A buffer management technique is best characterized by its space-time product [Den80]. As an example, Fig. 6 depicts the space-time products for the cold T1 traversal in the TB object base. Since a static buffer allocation scheme was used, the space-time product was computed by simply multiplying the response time with the client's buffer size. For all

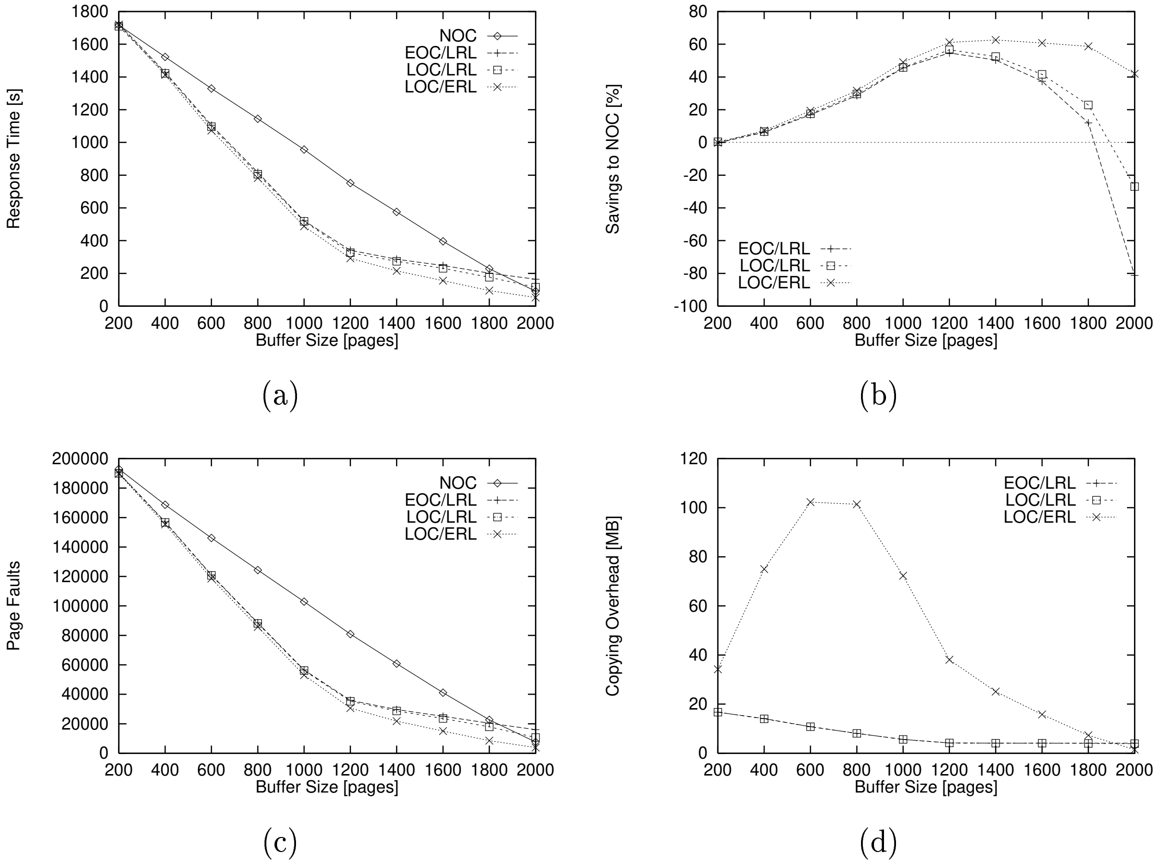

Figure 5: Cold T1 Traversal, RANDOM-clustering

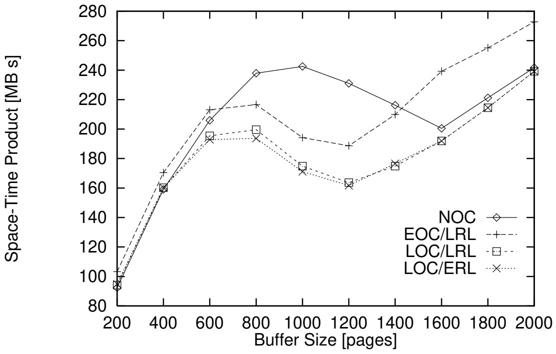

Figure 6: Space-Time Product of Cold T1 Traversals, TB-clustering

	COLD			WARM		
	TB	TOC	RANDOM	TB	TOC	RANDOM
NOC	200.5	244.8	743.8	141.1	158.1	643.6
EOC/LRL	188.7	173.5	1348.4	73.7	74.2	100.0
LOC/LRL	163.7	165.6	944.5	73.7	74.2	97.5
LOC/ERL	161.7	167.1	430.9	73.7	74.2	97.5

Table 1: Minimum Space-Time Products [MB∗s], T1 Traversals

the dual-buffering techniques, a (local) minimum could be observed for a buffer of 1200 pages; at this size, the (object-based) buffer was just large enough to hold the hot sets. Although a larger buffer reduced the response time, it was not economical.

In Fig. 6, a (global) minimum can be observed for a buffer of less than 200 pages for all strategies. Running the T1 traversal, however, in such a small buffer would result in an unacceptable response time. Therefore, Table 1 summarizes the minimum space-time products for a buffer in the range from 1000 to 2000 pages.

For the cold T1 traversal, LOC/ERL was the overall best technique. Although EOC/LRL and LOC/LRL reduced the response time in the RANDOM object base in most cases compared to NOC (cf. Fig. 5b), they were not economical if the system was well-tuned. In the fairly well clustered TB and TOC object bases, on the other hand, EOC/LRL should be preferred to NOC even though it had higher response times in many cases (cf. Fig. 4b).

To determine the space-time products of the *warm* traversals, the T1 traversal was run twice in succession and only the running time of the second run was measured. For dual buffering, most of the object copying was carried out in the warm-up phase that was not measured and thus, the speed-up in comparison to NOC reached its maximum. In addition, no objects were updated and no object flushing and hardly any relocation took place during the warm T1 traversal and, therefore, all the dual-buffering strategies had almost identical performance.

6.2 Applications with Updates (T2a and T2b Traversals)

For update-intensive applications, EOC/LRL can have severe performance drawbacks due to the flushing of modified objects. Again three cases can be distinguished. If the client's buffer is small, the home page of a modified object has often been replaced before the object is flushed. As a result, EOC/LRL induced up to 23% more page faults than NOC for the cold T2b traversal of the OO7 benchmark in the TB object base (cf. Fig. 7c), and the response time was almost twice as high in the worst case (cf. Fig. 7a).

For relatively large buffers (larger than 1600 pages), object replacement and thus, flushing modified objects was a rare event. But in this case, EOC/LRL suffered from the same drawbacks as for the T1 traversal. Consequently, only for a buffer of 1200 pages, EOC/LRL outperformed NOC due to its superior buffer utilization.

On the other hand, the best LOC technique, LOC^- in this experiment, had no significant performance drawbacks as compared to NOC for the T2b traversal (cf. Fig. 7b and Fig. 4b). On the contrary, the speed-up was sometimes even larger than for the T1 traversal because not only the number of page faults was reduced but, at the same time, also the number of times that a modified page that was replaced had to be flushed to the

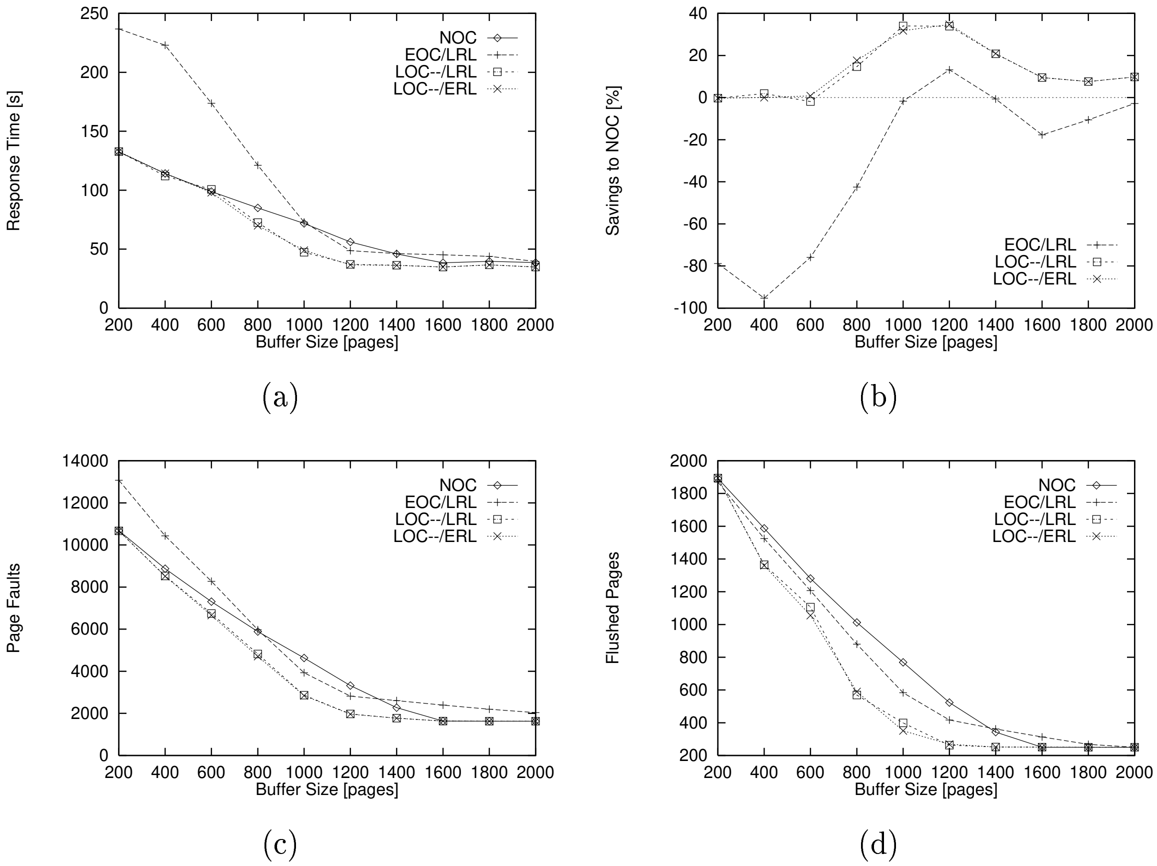

Figure 7: Cold T2b Traversal, TB-clustering

server was significantly reduced (cf. Fig. 7d). Compared to NOC, EOC/LRL also reduced the flushing of modified pages; but this effect was ruled out by the other drawbacks.

For the same reasons as in the T1 traversal, hardly any object copying was carried out under LOC$^-$ (and pure LOC) with small or large buffers. Thus, flushing modified objects was not critical in these cases. Furthermore, in the well-clustered TB object base, again no significant performance gap between lazy and eager relocation could be observed. In Fig. 8, thus, LOC$^-$, pure LOC, and LOC$^+$ are analyzed only in combination with lazy relocation.

For the TB object base, LOC$^-$ was the best object copying strategy (cf. Fig. 8a) because all the modified objects, i.e., the *AtomicParts*, were clustered together, and restricting the copying of these objects was highly desirable. For this reason, LOC$^+$ that copied these objects eagerly induced the most page faults (cf. Fig. 8c) and was the worst lazy object copying technique in this case. Fig. 8b confirms the fact that generally—disregarding some slight irregularities—LOC$^+$ has the highest and LOC$^-$ the lowest copying overhead, and Fig. 8d demonstrates that both, LOC$^+$ and LOC$^-$, reduce the number of modified pages that have to be flushed compared to pure LOC since LOC$^-$ precludes late updates and LOC$^+$ prevents the flushing of early updates, as stated in Section 3.1.

To identify a winner, all the strategies were tuned to minimize the space-time product. Tables 2 and 3 summarize the resulting minimum space-time products for cold and warm

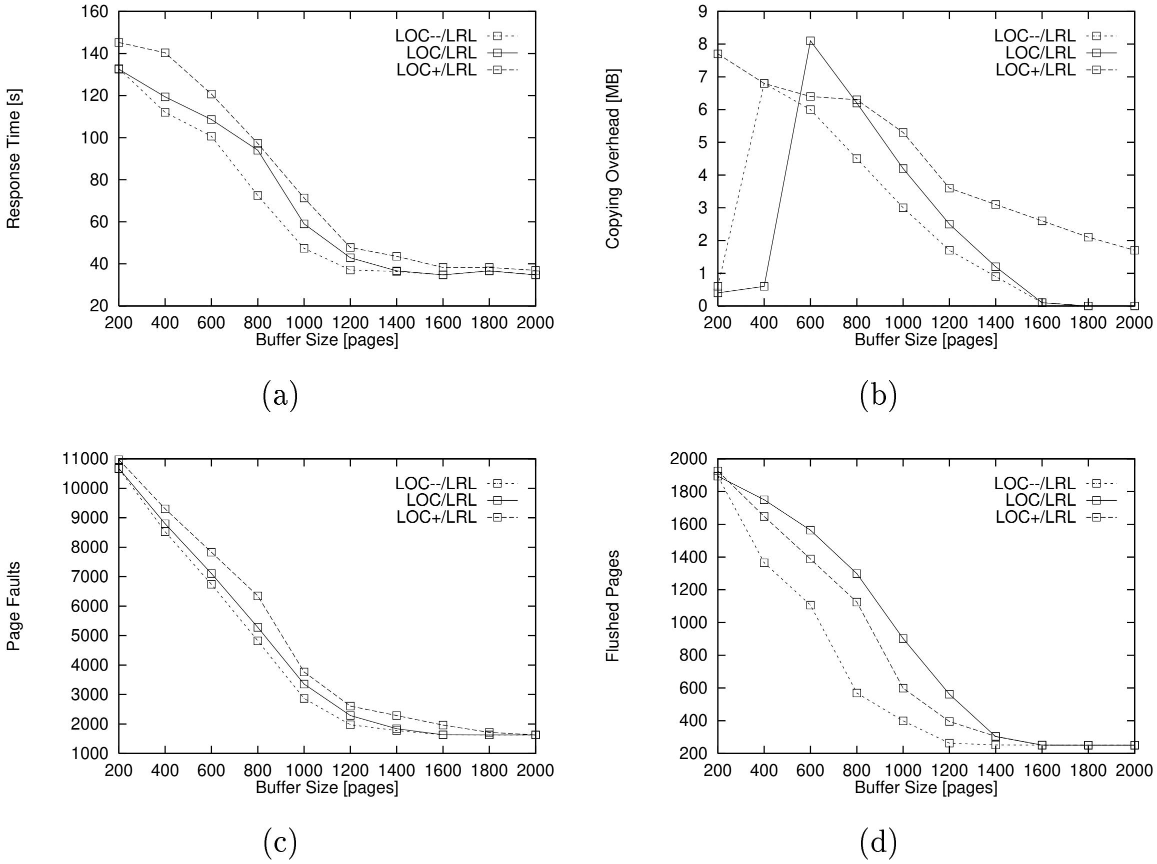

Figure 8: Lazy Object Copying Techniques, Cold T2b Traversal, TB-clustering

	T2a			T2b		
	TB	TOC	RANDOM	TB	TOC	RANDOM
NOC	248.6	313.8	799.8	251.7	336.2	1039.6
EOC/LRL	221.7	237.4	1404.9	239.4	318.0	1951.3
LOC^{-}/LRL	185.3	244.8	1017.4	182.4	323.4	1382.0
LOC/LRL	209.4	225.1	1057.9	210.5	313.1	1492.6
LOC^{+}/LRL	203.6	206.7	1030.0	234.9	308.7	1631.8
LOC^{-}/ERL	192.2	233.5	489.9	180.4	330.9	632.4
LOC/ERL	198.1	220.7	483.3	199.1	304.3	634.1
LOC^{+}/ERL	199.1	236.4	493.2	212.8	300.8	911.0

Table 2: Minimum Space-Time Products [MB∗s], Cold Traversals with Updates

	T2a			T2b		
	TB	TOC	RANDOM	TB	TOC	RANDOM
NOC	141.6	163.8	681.8	156.3	243.0	873.5
EOC/LRL	105.7	108.6	132.7	111.6	202.2	701.2
LOC^-/LRL	79.1	107.2	124.8	89.0	212.8	474.0
LOC/LRL	92.3	108.1	131.1	98.8	196.1	552.1
LOC^+/LRL	92.3	109.1	137.8	118.7	192.2	635.2
LOC^-/ERL	79.6	112.1	118.7	85.0	217.6	254.6
LOC/ERL	78.6	109.1	120.0	84.5	191.2	256.1
LOC^+/ERL	92.3	104.7	120.8	109.5	227.8	279.8

Table 3: Minimum Space-Time Products [MB$*$s], Warm Traversals with Updates

T2a (few updates) and T2b (many updates) traversals. Again, the warm traversals were measured immediately after having carried out a warm-up T1 traversal which has the same access behavior as the T2a and T2b traversals.

The trends were very similar to those of read-only applications (cf. Table 1): in any case, some dual-buffering technique could be found that outperformed NOC, and for warm traversals, any dual-buffering technique was more economical than NOC if it was well-tuned. When EOC/LRL was used, however, the gains were often not as striking due to the drawbacks of flushing modified and copied objects. For warm traversals in the RANDOM object base, for example, EOC/LRL reduced NOC's space-time product by only 19.7% for the T2b traversal compared to 84.5% for the T1 traversal. Sometimes, also the gap between the LOC techniques and NOC decreased with an increasing number of updates, but in general, the performance of lazy object copying appears to react significantly less sensitive to updates than eager object copying does.

In almost every experiment, eager relocation had the best performance. It should, therefore, always be used. No clear winner, however, could be determined among LOC^-, pure LOC, and LOC^+, but the differences were not immense. Pure LOC was seldom the best technique, but it was usually not far behind. In the worst case (cold T2b traversal in the TB object base), LOC/ERL was outperformed by only 9.4% by the best strategy (LOC^-/ERL). LOC^-, on the other hand, was often too restrictive when copying objects (e.g., for the TOC object base), and LOC^+ was often too *eager* (e.g., for the RANDOM object base).

7 Conclusions and Future Work

In this work, we devised and evaluated control strategies for combining buffer segmentation with dual buffering. Buffer segmentation was proven effective by previous work in the relational context, because of the flexibility to employ dedicated replacement policies for specific reference patterns. Dual buffering extends the flexibility of buffer segmentation: it allows to isolate objects from otherwise infrequently or not used pages while leaving heavily used pages in their entirety in the buffer.

Two control dimensions for maintaining such a segmented dual-buffer pool were distinguished: the copying time and the relocation time. Along both dimensions, we devised an eager and a lazy strategy, and alternative techniques in between these two extremes.

To assess the range of control strategies an experimentation platform with exchangeable object copying and relocation mechanisms was developed. The experiments carried out with the OO7 benchmark indicate that lazy object copying (and its derivatives) in conjunction with eager relocation "is the winner." This combination is very robust (i.e., we never experienced significant performance drawbacks as compared to pure page-based buffering) and very often outperforms page-based buffering by substantial amounts (up to 60% savings in running time).

In our assessment, we concentrated on a rather simple setting: only two buffer segments in a single-user environment. This scenario was sufficient to demonstrate the potential of the dual-buffering strategies. In practice, however, buffer pool segmentation becomes more complex, and tuning is an important task. In future work, thus, we will concentrate on showing how existing buffer allocation algorithms apply to dual buffering.

Acknowledgements

Carsten Gerlhof designed and implemented the page server used in the performance experiments. We thank Carl-Arndt Krapp who reviewed a draft of this paper.

References

[CD73] E. G. Coffman and P. J. Denning. *Operating Systems Theory.* Prentice Hall, 1973.

[CD85] H. Chou and D. DeWitt. An evaluation of buffer management strategies for relational database systems. In *Proc. of the Conf. on Very Large Data Bases (VLDB)*, volume 11, pages 127–141, 1985.

[CDN93] M. J. Carey, D. J. DeWitt, and J. F. Naughton. The OO7 benchmark. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, Washington, DC, May 1993.

[Den80] P. J. Denning. Working sets past and present. *IEEE Trans. Software Eng.*, 6(1):64–84, Jan 1980.

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

[EH84] W. Effelsberg and T. Haerder. Principles of database buffer management. *ACM Trans. on Database Systems*, 9(4):560–595, 1984.

[GKKM93] C. Gerlhof, A. Kemper, C. Kilger, and G. Moerkotte. Partition-based clustering in object bases: From theory to practice. In *Proc. of the Intl. Conf. on Foundations of Data Organization and Algorithms (FODO)*, volume 730 of *Lecture Notes in Computer Science (LNCS)*, pages 301–316, Chicago, Illinois, Oct 1993. Springer-Verlag.

[Gro91] EXODUS Project Group. EXODUS storage manager architectural overview. Exodus project document, Univ. of Wisconsin - Madison, Nov 1991.

[Inc93] Itasca Systems Inc. Technical summary for release 2.2, 1993. Itasca Systems, Inc., 7850 Metro Drive, Mineapolis, MN 55425, USA.

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

[KGBW90] 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):109–124, Mar 1990.

[Kim90] W. Kim. *Introduction to Object-Oriented Databases.* The MIT Press, Cambridge, MA, 1990.

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

[Knu73] D. E. Knuth. *The Art of Computer Programming*, volume I: Fundamental Algorithms. Addison Wesley Publishing Company, 1973.

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

[NFS91] R. Ng, C. Faloutsos, and T. Sellis. Flexible buffer allocation based on marginal gains. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 387–396, May 1991.

[OOW93] E. J. O'Neil, P. E. O'Neil, and G. Weikum. The LRU-K page replacement algorithm for database disk buffering. In *Proc. of the ACM SIGMOD Conf. on Management of Data*, pages 297–306, Washington, DC, May 1993.

[SS86] G. Sacco and M. Schkolnick. Buffer management in relational database systems. *ACM Trans. on Database Systems*, 11(4):473–498, Dec 1986.

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

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

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