A Performance Evaluation of Multi-Level Transaction Management*

Christof Hasse and Gerhard Weikum

Computer Science Department
ETH Zurich
CH-8092 Zurich, Switzerland
E-Mail: {hasse,weikum}@inf.ethz.ch

Abstract

Multi-level transactions are a variant of open nested transactions in which the subtransactions correspond to operations at different levels of a layered system architecture. The point of multi-level transactions is that the semantics of high-level operations can be exploited in order to increase concurrency. As a consequence, undoing a transaction requires compensation of completed subtransactions. In addition, multi-level recovery methods have to take into account that high-level operations are not necessarily atomic if multiple pages are updated in a single subtransaction. This paper presents a performance evaluation of the multi-level transaction management that is implemented in the database kernel system DASDBS. In particular, it is shown that multi-level recovery can be implemented in an efficient way. We discuss performance measurements, using a synthetic benchmark for processing complex objects in a multi-user environment.

1. Introduction

Multi-level transactions are a variant of open nested transactions in which the subtransactions correspond to operations at different levels of a layered system architecture [BSW88]. The point of multi-level transactions is that the semantics of high-level operations can be exploited in order to increase concurrency. For example, two "deposit" operations on a bank account are commutative and can therefore be admitted concurrently (e.g., on behalf of two funds transfer transactions). However, executing such high-level operations in parallel requires that a low-level synchronization mechanism takes care of possible low-level conflicts, e.g., on indexes or data pages. In relational DBMSs where records do not span pages, this low-level synchronization is usually implemented by page latches, i.e., cheap semaphores that are held while a page is accessed. For advanced DBMSs with complex high-level operations that may access many pages in a dynamically determined (i.e., not pre-defined) order, the simple latching method is not feasible since it cannot ensure the indivisibility of arbitrary multi-page update operations. Rather, high-level operations need to be executed as subtransactions that are dealt with by a general concurrency control mechanism at the lower level. This principle, which can be applied to an arbitrary number of levels, ensures that the semantic concurrency control at the top level need not care about lower-level conflicts.

In this paper, we address multi-level transaction management in advanced DBMSs that deal with complex objects, by applying multi-level transaction management to the following two levels:

- At the **object level L1**, semantic locks are dynamically acquired and held until end-of-transaction (EOT) according to the strict two-phase locking protocol. The semantics of the high-level operations is exploited in the lock modes and the lock mode compatibility table, which is in turn derived from the commutativity properties or semantic compatibility [Ga83, SZ89] of the operations. In principle, one could even exploit state-dependent commutativity [O'N86, We88], but this is beyond the scope of this paper.
- At the **page level L0**, page locks are dynamically acquired during the execution of a subtransaction and are released at end-of-subtransaction (EOS). Note that, unlike in conventional nested transactions [Mo85], the locks of a subtransaction are not inherited by the parent. Releasing the low-level locks as early as possible while retaining only a semantically richer lock at a higher level is exactly why multi-level transaction management allows more concurrency than single-level protocols.

An example of a (correct) parallel execution of two multi-level transactions is shown in Figure 1. Suppose an office document filing system where documents have a complex structure and can span many pages. Users modify documents by specific high-level operations such as 1) "change the font of all instances of a particular component type (e.g., text paragraphs)" and 2) "change the contents of a figure". These two *Change* operations on the same document are commutative; however, since they may access many subobjects of the document (e.g., because the layout of the entire document is recomputed), the potential conflicts at the lower level have to be dealt with. In Figure 1, this is done by acquiring locks on the underlying pages that

* This project is supported by the Union Bank of Switzerland (Schweizerische Bankgesellschaft).

are released at the end of the subtransactions T_{11}, T_{12}, and T_{21}, respectively.

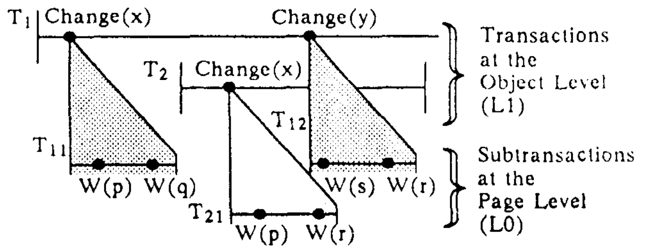


Fig.1: Parallel Execution of Multi-Level Transactions

Similar examples arise in advanced business applications with large amounts of derived data. For example, in foreign exchange transactions, a forward transaction (e.g., a currency swap) may have to compute a large number of future positions for risk assessment (e.g., to compute how many Japanese Yen a bank will hold at a particular date). In such an application, the potential data contention can be reduced by updating the derived data within subtransactions that release low-level locks early.

An inherent consequence of multi-level locking is that transactions can no longer be undone by simple state-oriented recovery methods at the page level. Rather, since page locks have been released at EOS, completed subtransactions must be compensated by inverse high-level operations. These operations are in turn executed as so-called **compensating subtransactions** [MGG86, Wei87, BSW88, GS87, KLS90, SDP91, We89, Wei91]. In the example of Figure 1, undoing transaction T_1 would require two inverse *Change* operations on y and x, i.e., two additional subtransactions that compensate the completed subtransactions T_{12} and T_{11} (reversing the order of the original subtransactions).

Compensating subtransactions are necessary for both handling transaction aborts and crash recovery after a system failure. An important prerequisite is that both regular subtransactions and compensating subtransactions have to be atomic. Otherwise, the recovery after a crash may be faced with a database state that is not sufficiently consistent to perform the necessary high-level undo steps. For example, the storage structures of a complex object may contain dangling pointers, or some derived data may only partially reflect the primary updates. If a subtransaction modifies multiple pages, as shown in Figure 1, a low-level recovery mechanism at the page level is necessary in order to provide subtransaction atomicity. This problem is challenging in that a straightforward implementation of multi-level recovery may cause excessive logging and could thus diminish the benefits of the enhanced concurrency of multi-level transactions.

Theoretical and practical issues of multi-level transaction management have been addressed by a variety of papers [BBG89, BF89, BR90, BSW88, Bü88, CF90, FLMW88, GS87, HH88, Ma87, MGG86, MR91, RGN90, SDP91, SG88, Sh85, WS84, WS91, Wei86, Wei87, WHBM90, Wei91]. However, to our knowledge, none of the previous work has presented a full implementation. Furthermore, only two papers have presented performance figures. [Wei91] contains performance measurements with a multi-level transaction manager built on top of the commercial Codasyl database system UDS; the results were strongly affected by the fact that UDS could not be changed in these experiments. [BR90] contains simulation results on multi-level concurrency control only; i.e., disregarding recovery issues.

This paper presents a performance evaluation of a full-fledged implementation of multi-level transaction management. The implementation is integrated in the database kernel system DASDBS [SPSW90]. The paper presents performance measurements, based on a synthetic benchmark for complex-object processing. In particular, it is shown that multi-level recovery can be implemented in an efficient way. The implemented system even allows that subtransactions of the same transaction are executed in parallel. However, for space limitations, intra-transaction parallelism is not discussed in this paper (see [WH91] for performance results with varying degrees of inter- and intra-transaction parallelism).

The rest of the paper is organized as follows. Section 2 presents our implementation of multi-level transaction management, with emphasis on the performance-critical recovery component. Section 3 discusses the results of a comprehensive series of performance experiments. Section 4 compares our implementation with related work, especially the ARIES recovery method [Mo89]. We conclude with an outlook on future work that could further enhance the performance of multi-level transaction management.

2. Implementation of Multi-Level Transaction Management in DASDBS

Our lock manager can manage multiple lock tables that are specified to handle particular types of lockable items (e.g., pages, objects, objects of different object types, index keys, keys of different indexes, conjunctive predicates, etc.). In addition to the usual lock modes *"shared"* and *"exclusive"*, semantic lock modes such as *"increment"* can be incorporated by specifying the lock mode compatibility matrix at the creation time of a lock table [SS84]. In the performance experiments that are described in Section 3, this feature was not exploited; rather, shared and exclusive locks were acquired on sets of object identifiers.

Our implementation of multi-level recovery is illustrated in Figure 2. To ensure the atomicity of transactions, undo log records are written at the object level L1. Each of these log records contains information about the compensating subtransaction that is necessary to undo an executed high-level operation. To ensure the persistence of transactions, redo log records are written at the page level L0. Compared to operation-oriented redo at the object level (as in System R [Gr81]), page-level redo saves work during a warmstart since it merely reconstructs pages rather than re-executing resource-intensive high-level operations.

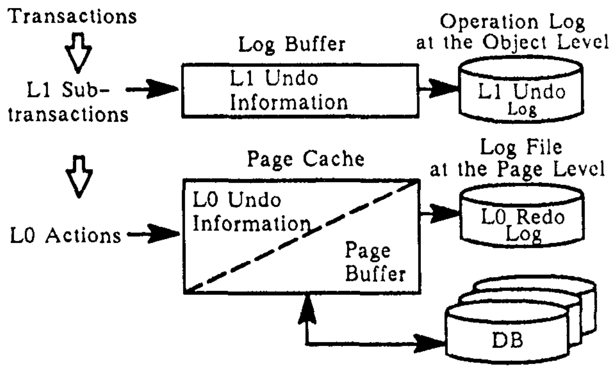


Fig.2: Architecture of the DASDBS Multi-Level Transaction Management

Our implementation of page-level redo logging is based on the DB Cache method [EB84]; that is, entire page after-images are written to a sequential log file. Compared to entry logging (i.e., logging page modifications), the DB Cache method generates more log volume. Note, however, that after-image logging does not cause a higher number of log I/Os, provided that multiple pages can be sequentially written in a single set-oriented I/O. On the other hand, during a warmstart, a recovery method with entry logging is slower than a method with after-image logging. This is because pages have to be fetched from the database before the update that is described in a log record can be installed, whereas after-images can be directly written into the database right after they have been read from the log. That is, after-image logging saves a substantial number of random I/Os during the warmstart. In addition, the DB Cache method has nice properties with respect to log space management, as the log file is dynamically compacted without having to take checkpoints [EB84]. For these reasons and for simplicity, we have adopted the DB Cache method in DASDBS. Note, however, that our multi-level recovery architecture would allow L0 entry logging (e.g., [MLC87]) as well.

Ensuring Subtransaction Atomicity

A requirement that makes multi-level recovery difficult is subtransaction atomicity. Essentially, this is also ensured by applying the DB Cache method to subtransactions at the page level L0 as follows:

- before-images of incomplete subtransactions are kept in main memory as temporary page versions in the buffer pool,
- modified pages are guaranteed to remain in the buffer pool until EOS (which is usually referred to as a No-Steal policy [HR83]), and
- the after-images of a subtransaction are written to the L0 log file atomically at EOS, by including a special (EOS) flag in the header of the last page of the written after-images.

In addition, before writing a subtransaction's after-images, the corresponding high-level undo log record must be written to the L1 log file, in order to ensure transaction atomicity.* The log records that are written for the example of Figure 1 are shown in Figure 3. Operations with an overbar denote inverse operations; "Change" operations are abbreviated to "C". For correctly handling non-idempotent high-level operations during a warmstart, the L1 log records and the after-images that are written at L0 include a subtransaction identifier in a special header field (see [WHBM90, WH91] for further discussion). The warmstart after a crash consists of the following two steps:

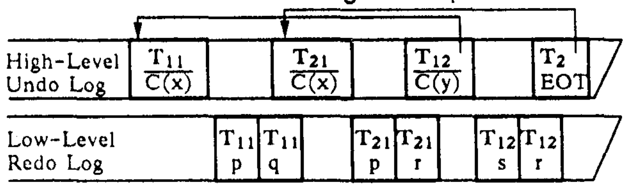


Fig.3: Log Contents for the Example of Fig.1

1) **Redo pass**: During a forward pass on the L0 redo log, after-images are loaded into the buffer pool and written into the database at the discretion of the buffer manager. The redo pass ensures transaction persistence and subtransaction atomicity. After-images after the latest EOS-flagged after-image are ignored since they belong to an incomplete write at EOS.

2) **Undo pass**: After the redo pass, a backward pass is performed on the L1 undo log. The undo pass ensures transaction atomicity. Transactions for which an EOT log record is found are winners and do thus not need any processing. For loser transactions, compensating subtransactions are performed according to the contents of their log records.

Deferred Log Writes

The multi-level recovery algorithm described above has been implemented in a former version of DASDBS. This algorithm has a potential performance problem in that it may cause excessive log I/Os for ensuring the atomicity of subtransactions. This is because after-images of a subtransaction are forced to disk immediately at EOS. In the example of Figure 1, this means that an after-image of page p is written to disk at the EOS of T_{11} and the EOS of T_{21}, as shown in Figure 3.

While there are generic techniques to reduce these I/O costs such as batching log I/Os of multiple transactions [GK85], there is a more fundamental way to cut

* This is simply a consequence of the write-ahead log principle.

down the log I/O costs of multi-level transactions. The general idea is to defer the writing of a subtransaction's after-images until EOT rather than forcing them at EOS. This would make multi-level logging as efficient as conventional single-level logging, e.g., the original DB Cache method [EB84]. However, deferring all L0 log writes until EOT is not a correct solution. The reason is that there may be subtransactions of different transactions such that the after-image sets of the subtransactions are overlapping, i.e., have a page in common. This is possible because page locks are released at EOS. In such a situation, forcing the after-images of one subtransaction at the EOT of its parent may violate the atomicity of other subtransactions.

Consider the example of Figure 1. Ideally, we would want to write the after-images of T_{11} and T_{12} not before the EOT of T_1. The EOT of T_2 requires writing the after-images of T_{21}, i.e., pages p and r as of the EOT time of T_2. Writing these pages to the L0 log, however, would implicitly write the modifications that T_{11} made on p, too. Then, if the system crashed before the EOT of T_1 (i.e., before the after-images of T_{11} are written), the redo pass of the warmstart would violate the atomicity of T_{11} by restoring the update on p while disregarding T_{11}'s update on q. Note that this problem would arise also with entry logging (rather than after-image logging) because subtransactions of different transactions may have modified a common byte through commutative high-level update operations.

The same problem, in a slightly different flavor, arises with respect to subtransaction T_{12}. For efficient memory use, the after-images reside in regular buffer frames rather than being copied to a separate L0 log buffer. For the same reason, we do not want to keep multiple versions of a page in the buffer pool when there is no active subtransaction that has modified the page; that is, the temporary before-images are discarded at EOS. Therefore, at the EOT of T_2, only one version of page r resides in the buffer pool. This version contains the updates of the completed subtransaction T_{12} even though T_{12} was serialized after T_{21}. Thus, writing this after-image of r to the L0 log file would violate the atomicity of T_{12}.

These and other related problems have been discussed more rigorously in [WHBM90] and have led to a solution that is based on the notion of **persistence spheres**. The persistence sphere $PS(T_{ij})$ of a subtransaction T_{ij} is defined as follows:

- $PS(T_{ij})$ contains all pages that were modified by T_{ij}.
- If there is a page p in $PS(T_{ij})$ and a subtransaction T_{kl} such that p is also in $PS(T_{kl})$ or T_{kl} has read p after the EOS of T_{ij}, then $PS(T_{ij})$ contains all pages of $PS(T_{kl})$; that is, the persistence spheres of the two subtransactions are merged.

The persistence sphere $PS(T_i)$ of a transaction T_i is the union of the persistence spheres of its subtransactions.

Now, our solution to the deferred log write problem is the following. At the EOT of a transaction T_i, all pages in the persistence sphere of T_i must be written to the L0 log.* In the example of Figure 1, the subtransaction T_{21} has modified pages in common with T_{11} as well as T_{12}. Thus, at the EOT of T_2, the persistence sphere of T_2 contains the pages p and r that were modified by T_2's own subtransaction T_{21} and the pages q and s that were modified by T_{11} and T_{12}, respectively.

Persistence spheres are written atomically to the L0 log file, by setting the EOS flag in the header of the last page. As discussed above, a persistence sphere may contain updates of completed subtransactions that belong to incomplete transactions. These subtransactions will have to be compensated if the system crashes before the EOT of their parent. To be able to do so, the L1 undo log must be forced before the L0 log I/O. In addition, it seems that immediately after the writing of the persistence sphere is completed, another L1 log I/O is often necessary in order to force the EOT log record of the committing transaction that caused the writing of the persistence sphere. Fortunately, this second L1 log I/O can be avoided by including an additional **EOT flag** and the number of the committing transaction in the header of the last page of the persistence sphere. An EOT log record is nevertheless created in the L1 log buffer pool, but need not be forced before the next compaction of the L0 log file that would discard the after-image that contains the EOT flag. The log records for the example of Figure 1 are shown in Figure 4.

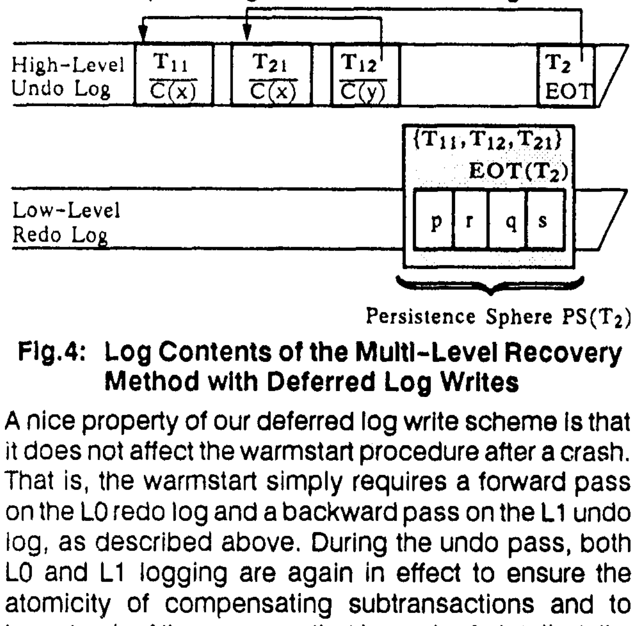

High-Level Undo Log
T_{11} $\overline{C(x)}$
T_{21} $\overline{C(x)}$
T_{12} $\overline{C(y)}$
T_2 EOT
$\{T_{11}, T_{12}, T_{21}\}$ EOT(T_2)
Low-Level Redo Log
p
r
q
s
Persistence Sphere PS(T_2)

Fig.4: Log Contents of the Multi-Level Recovery Method with Deferred Log Writes

A nice property of our deferred log write scheme is that it does not affect the warmstart procedure after a crash. That is, the warmstart simply requires a forward pass on the L0 redo log and a backward pass on the L1 undo log, as described above. During the undo pass, both L0 and L1 logging are again in effect to ensure the atomicity of compensating subtransactions and to keep track of the progress that is made. A detailed discussion of the warmstart procedure is contained in

* In addition, replacing a dirty page p in the buffer pool requires forcing to the log all pages in the persistence sphere of the last completed subtransaction that modified p (see [WHBM90]).

[WHBM90, WH91]. The presented method for deferred log writes has been implemented in DASDBS; the details of managing persistence spheres are described in [WH91].

3. Performance Evaluation

3.1 Description of the Experiments

In this subsection, we describe the experiments that were performed to evaluate the performance of our algorithms for multi-level transaction management. We compared the following three strategies, all of which are implemented in DASDBS:

- *strategy S1*, page-oriented single-level transaction management, using strict two-phase locking on pages and the DB Cache method for recovery,
- *strategy S2*, two-level transaction management with log writes at each EOS, and
- *strategy S2/PS*, two-level transaction management with deferred log writes based on the notion of persistence spheres, as described in Section.2.

Since the logging overhead was one of the main aspects that we wanted to investigate, we summarize the principal log I/O costs of the above three strategies in Figure 5.

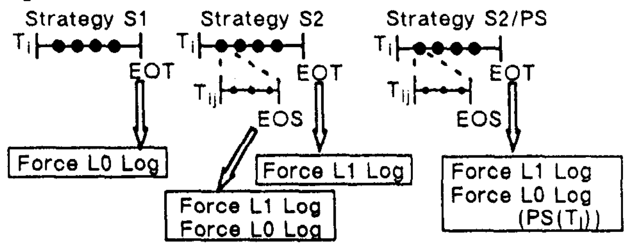


Fig.5: Log I/O Costs of Recovery Strategies

Our performance evaluation is based on a synthetic benchmark which follows some ideas proposed in [DFMV90]. The benchmark has the following characteristics, as illustrated in Figure 6.

Our test database consists of 1000 complex objects (COs) each of which consists of 1000 "own" subobjects (SOs) and 100 references to "foreign" subobjects, i.e., subobjects that are owned by other complex objects. Thus, SOs can be referentially shared by multiple COs; however, each SO is owned by exactly one CO. The foreign SO references of a CO are generated by selecting a CO according to an 80-20 rule and a SO within the selected CO according to a 50-50 rule. That is, 80% of the foreign SO references point to SOs that are owned by 20% of the COs in the database. This reflects the skewed distribution of object relationships in most real-life applications. In our benchmark, the 80-20 rule and the 50-50 rule were implemented by applying a linear transformation to a normal distribution of random numbers.

The 1000 "own" SOs of a CO constitute a storage cluster that consists of 10 contiguous pages, with a page

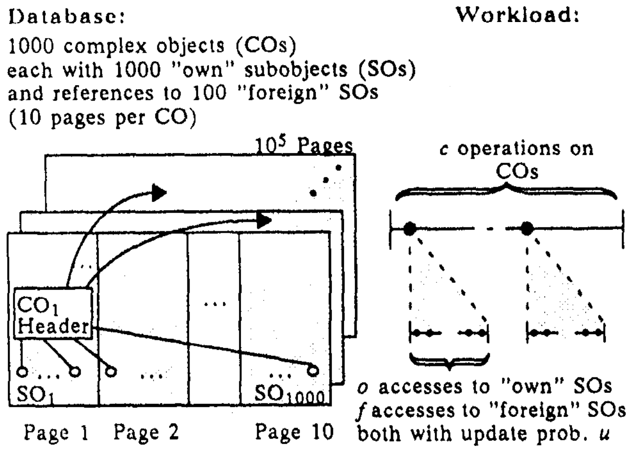

Database:
1000 complex objects (COs)
each with 1000 "own" subobjects (SOs)
and references to 100 "foreign" SOs
(10 pages per CO)
10^5 Pages
CO_1 Header
SO_1
SO_{1000}
Page 1
Page 2
Page 10
Workload:
c operations on COs
o accesses to "own" SOs
f accesses to "foreign" SOs
both with update prob. *u*

Fig. 6: Database and Workload of the Experiments

size of 2KBytes. The first page of each storage cluster contains the CO header, i.e., a directory of SO references. The total database size is 10000 pages, i.e., 20 MBytes.

The workload of our benchmark consists of a single transaction type which performs *c* complex high-level operations each on a different CO. Each of these synthetic high-level operations accesses *o* own subobjects and *f* foreign subobjects of a CO. A subobject is modified with probability *u*. These updates do not affect the CO header; that is, the header page of a CO is read-only to avoid an obvious data-contention bottleneck in the benchmark. The COs that are processed by a transaction are selected according to an 80-20 rule, the own SOs within a CO are selected according to a 50-50 rule, and the foreign SOs are selected to a uniform distribution as the references themselves are already non-uniformly distributed (see above). According to [Hä87], this skewed distribution is rather conservative compared to the access skew of many real-life applications.

In the multi-level transaction management strategies S2 and S2/PS, each high-level operation on a CO corresponds to a subtransaction. At the object level, each high-level operation acquires shared locks on the set of accessed SOs, using object identifiers as the actual lock items. For modified SOs, these locks are acquired in exclusive mode. At the page level, all accessed pages are locked in shared mode, with conversions to exclusive locks for modified pages. In the strategies S2 and S2/PS, all page locks are released at EOS (i.e., when a high-level operation completes), whereas in the single-level transaction management strategy S1, all page locks are held until EOT.

The experiments were designed as a stress test for transaction management on complex objects, with a small database and fairly long update transactions. All measurements were performed with DASDBS running on a 12-processor Sequent Symmetry shared-memory computer, with a page buffer pool of 2 MBytes. Each run of the experiments was driven by a fixed number of

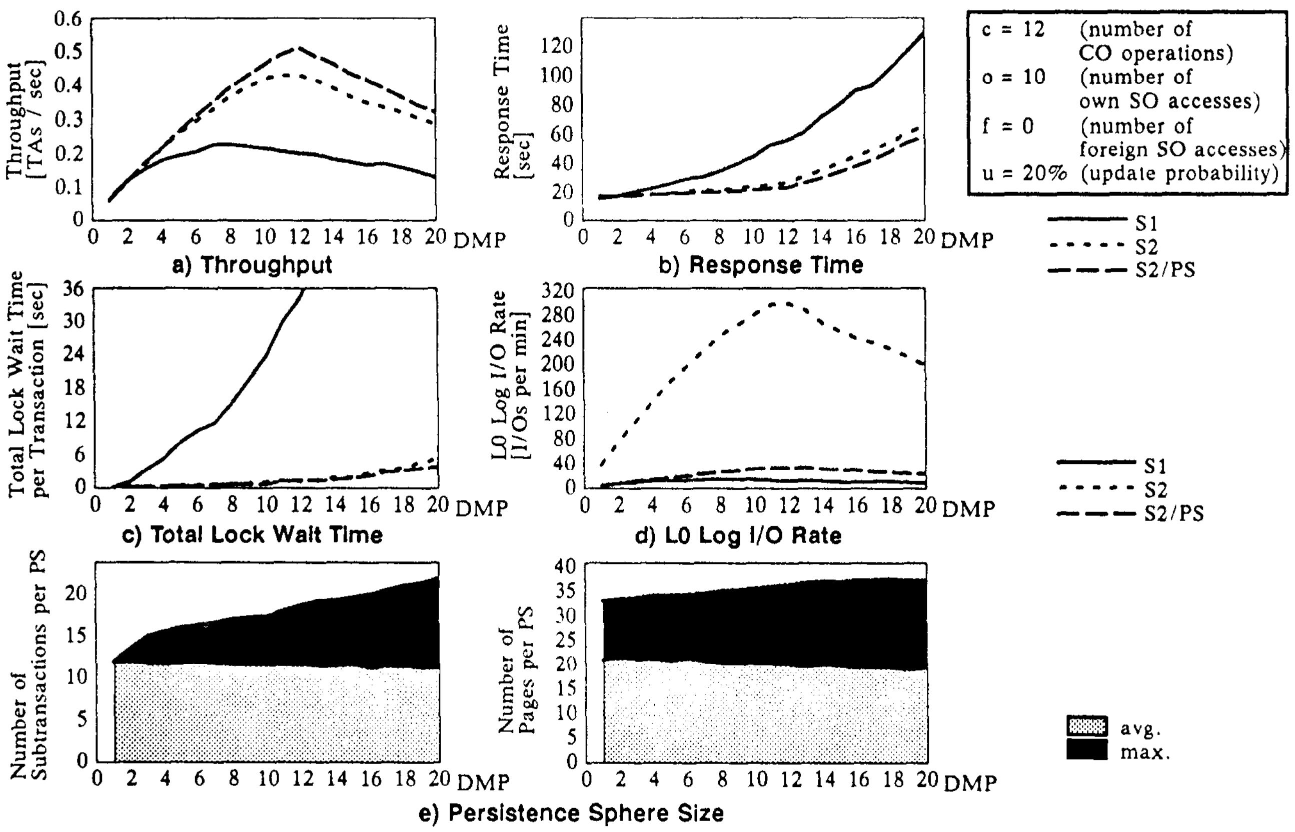


Fig. 7: Performance Results of the Baseline Experiment with Disjoint Complex Objects

processes that execute transactions. This number of processes restricts the maximum number of transactions that can be concurrently executing, and is referred to as the **degree of multiprogramming (DMP)**. In the experiments, the DMP was systematically varied for different runs.

3.2 Results for Disjoint Complex Objects

In this section, we discuss the performance results for the case without accesses to foreign subobjects (i.e., f was set to 0). We first discuss the results of a "baseline experiment" with $c=12$ complex-object operations per transaction, $o=10$ own-subobject accesses per complex-object operation, and update probability $u=20\%$. We have also performed a sensitivity analysis of these parameters, as discussed below. In the following, we discuss the key observations from these experiments.

Overall performance:
In all experiments, both two-level strategies S2 and S2/PS clearly outperformed the one-level strategy S1. Transaction throughput and response time were improved by factors of up to 2.5 (i.e., more than two times higher throughput) and 2.4 (i.e., more than two times shorter response time). Figures 7a and 7b show throughput and response time as a function of the DMP, where the DMP was varied between 1 and 20. Maximum throughput was reached at a DMP of 12.

Lock conflicts:
The performance gains of the two-level strategies result from the fact that the performance of S1 is limited by data contention whereas S2 and S2/PS have relatively few lock conflicts. For strategy S1, we observed a conflict rate (i.e., ratio of lock waits to lock requests) of 1.6 percent at a DMP of 12. This may appear acceptably low. However, the specific page reference pattern of our benchmark, with high locality within a complex object, seems to underrate the impact of the lock conflict probability. In fact, the total time that a transaction, on average, spent waiting for a lock is a more significant metric in this experiment. For example, with strategy S1 and a DMP of 12, an average transaction spent about 36 seconds waiting for locks, which is about 60 percent of a transaction's response time. With strategies S2 and S2/PS, on the other hand, this lock wait time was reduced to less than 3 seconds per transaction. Figure 7c shows the total lock wait time of all three strategies as a function of the DMP.

Log I/Os:
As the simple two-level strategy S2 performed log I/Os for each update subtransaction, its log I/O rate was dramatically higher than that of strategy S1 (see Figure 7d). This disadvantage of S2 was almost completely eliminated by strategy S2/PS. For example, at a DMP of 12, strategy S2/PS had about 2.7 times more page-level log I/Os than strategy S1; however, as it achieved

2.5 times the throughput of S1, the log I/O rates of the single-level strategy and the improved two-level strategy are actually quite comparable. Note that these results reflect the relative I/O performance of the investigated strategies. As for absolute performance, the log I/O rate did not have a significant effect on throughput or response time in any of our experiments. Even with strategy S2, the excessive number of log I/Os caused only about 5% utilization of each of the L0 log disk and the L1 log disk. Keep in mind, however, that with more or faster CPUs, log I/O could eventually become a performance-limiting factor. Then the savings in log I/Os that strategy S2/PS achieved would become a crucial performance advantage.

Strategy S2/PS was even superior to strategy S1 in terms of the number of pages that are written in one page-level log I/O. Because update subtransactions are dynamically combined into persistence spheres, it was often the case that a page that was modified by multiple subtransactions of different transactions was written to the log only once. This main feature of our improved multi-level logging approach led to an effect similar to group commit. With strategy S2/PS, on average only 19.9 pages rather than 22.3 pages were written in one L0 log I/O, at a DMP of 12. As the decreasing average persistence sphere size in Figure 7e shows, this nice effect increases with the DMP. Note, however, that, in contrast to group commit, our method does not impose any delays on transaction commits other than the log I/O itself. In fact, group commit and our deferred log write approach are orthogonal steps toward reducing log I/O costs.

Performance Impact of internal latches:
As the throughput and response time curves in Figures 7a and 7b show, strategy S2/PS performs slightly better than strategy S2. Even though one might think that this is the effect of the savings in log I/Os, the absolute costs of log I/O are actually negligible in both strategies. Rather the performance difference is because strategy S2/PS saves calls to the buffer manager as it defers the writing of after-images. This reduces some CPU overhead, and decreases the contention on internal latches that are used to synchronize the access to the buffer manager's frame control blocks (see also [GT90] for similar experiences). Such latch contention is also the major reason for the drop of performance that both S2 and S2/PS suffer when the DMP exceeds 12 (i.e., the number of processors). Since we implemented latches by spin locks, latch contention actually led to wasted CPU cycles; and since the CPU utilization was almost 100% at DMP 12, increasing the DMP beyond 12 caused a significant decrease of performance.

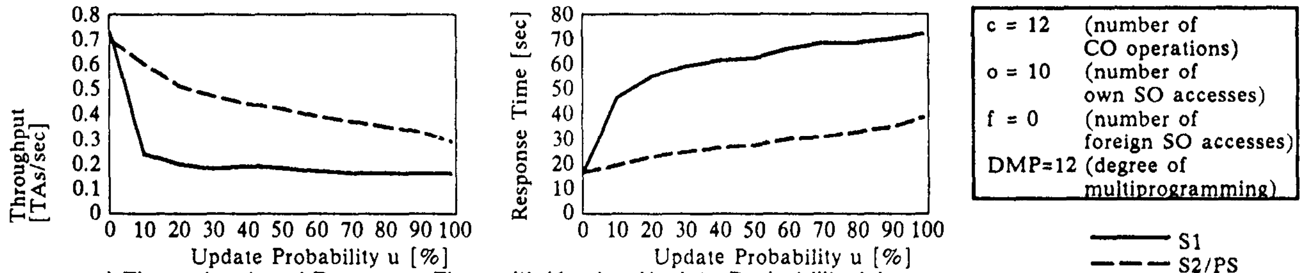
Sensitivity of baseline parameters:
We performed additional experiments to study the sensitivity of the various parameters of our baseline experiment. In particular, we varied the update probability u, the number o of own-subobject accesses per complex-object operation, and the number c of complex-object operations per transaction. The results are shown in Figure 8. These experiments essentially reconfirmed the observations discussed above. In interpreting the slope of the curves, one should note that the number of modified pages per complex-object operation increases only slowly with the number of updated subobjects because of the high locality within a complex object.

3.3 Results for Complex Objects with Referentially Shared Subobjects

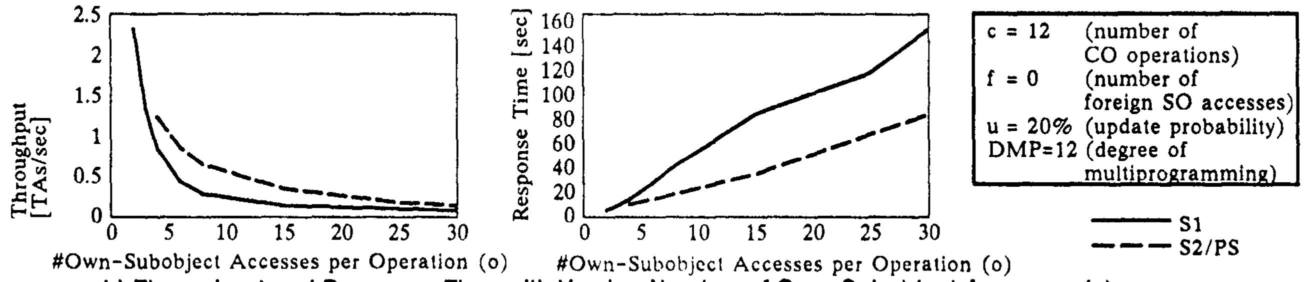
In this section, we discuss the performance results for the case with accesses to foreign subobjects. We first discuss the performance when all subobjects that are accessed by a complex-object operation are foreign subobjects (i.e., subobjects that are physically clustered with other complex objects). In the discussed experiments, $f = 10$ foreign subobjects were accessed per complex-object operation with update probability $u = 20\%$. We have also performed a sensitivity analysis of the f parameter, by keeping the sum $o + f$ (i.e., the total number of SO accesses per CO operation) constantly at 10 and varying f from 0 to 10. In the following, we discuss to what extent foreign-subobject accesses changed the results obtained in Section 3.2. Strategy S2 is no longer considered here since it was always outperformed by S2/PS.

Overall performance and lock conflicts:
As shown in Figure 9, the performance difference of S1 and S2/PS became even bigger, compared to the case without foreign-subobject accesses. For example, at a DMP of 12, S2/PS achieved 16 times higher throughput and 10 times shorter response time than S1. As Figure 9c shows, this performance difference is mostly caused by data contention. For strategy S1, both the total lock wait time and the conflict rate were substantially higher than in the experiment of Section 3.2. In addition, the number of deadlocks increased considerably.

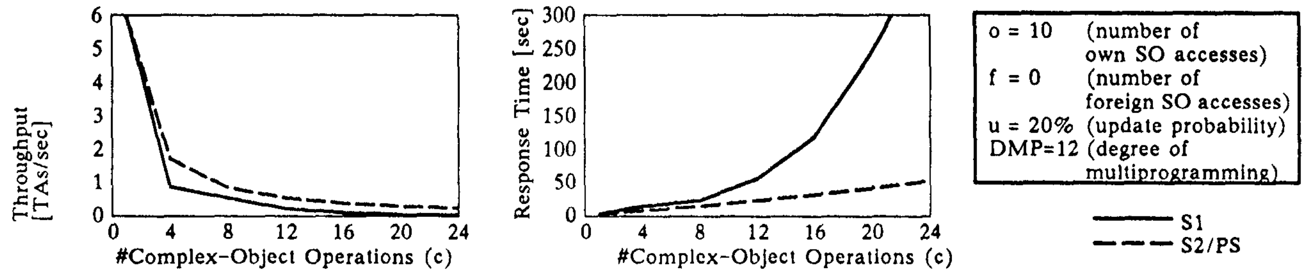
With foreign-subobject accesses, the subobjects that are accessed by a subtransaction are scattered across the entire database. Compared to the results of Section 3.2, this fact destroyed the locality in the page accesses of a subtransaction. Thus, the total number of pages that are accessed within a transaction was increased, and the page access pattern was better randomized. For example, in the experiment of Section 3.2, the first SO access within each complex-object operation had a higher probability of getting blocked than the other SO accesses within the same CO, as the latter benefit from the already acquired locks because of the high locality of subobject (and hence page) ac-



a) Throughput and Response Time with Varying Update Probability (u)



b) Throughput and Response Time with Varying Number of Own-Subobject Accesses (o)



c) Throughput and Response Time with Varying Number of Complex-Object Operations (c)

Fig. 8: Sensitivity of Baseline Parameters with Disjoint Complex Objects

cesses.* Destroying this locality led to the disastrous performance of strategy S1.

Log I/Os:

The most interesting aspect of the experiment with foreign-subobject accesses is the relationship between the DMP and the size of persistence spheres, as shown in Figure 9e. Whereas the average size of persistence spheres was not much affected by the DMP, the maximum persistence sphere size increased quite significantly with increasing DMP. As pointed out in Section 3.2, this effect can be quite beneficial, for it amounts to more batching of log I/Os (i.e., less but longer log I/Os). However, batching log I/Os is desirable only up to a certain point. If persistence spheres become too large, then the writing of a persistence sphere adds a significant delay to the response time of the committing transaction that caused the log I/O. In our experiments, the maximum persistence sphere at a DMP of 12 contained about 95 pages (each of size 2K). Writing this persistence sphere to a single log disk takes about 100 milliseconds, which is still negligible in our experiment but may be unacceptable in a different environment (e.g., with much faster CPUs).

Of course, writing the after-images in a persistence sphere is unavoidable in order to commit a transaction. In fact, our deferred write approach minimizes the number of pages that need to be written. The point, however, is that our method may cause *unpredictable* delays. The reason is that a large amount of log I/O work may be imposed on a transaction that has not done much work itself but happens to have a large persistence sphere constituted mostly by subtransactions of other active transactions. These unpredictable delays should be avoided in a high performance environment with response-time constraints. Note, however, that the delay caused by writing a large persistence sphere is still much shorter and therefore less severe than the delay that a synchronous checkpoint mechanism (e.g., [Gr81]) would cause.

There are two ways to eliminate or alleviate the described effect (none of which is currently implemented in DASDBS, though). The first way is to prevent the formation of large persistence spheres. This can be achieved by asynchronously writing persistence spheres whenever their size exceeds a certain threshold, even if the log I/O could be further deferred. Such a

* The net effect is similar to preclaiming, even though no preclaiming is actually performed.

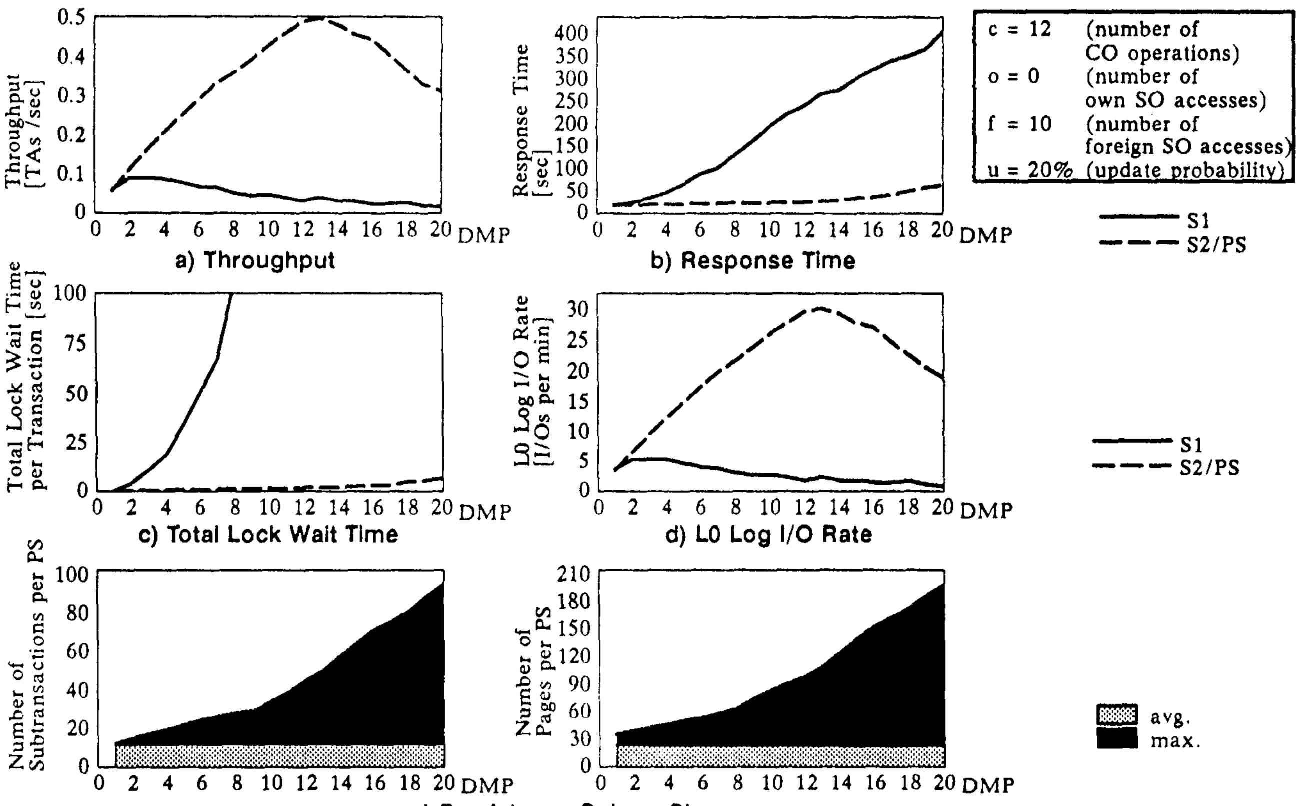


Fig. 9: Performance Results of the Experiment with Foreign-Subobject Accesses (f = 10)

mechanism may actually increase the total amount of work since it may write more pages, but it has the advantage that it can distribute the log I/O load more evenly over time. The second way to cope with large persistence spheres is to make their writing more efficient. This can be achieved by striping the log over multiple disks in a round-robin fashion (i.e., RAID-like striping) with a sufficiently large striping unit (e.g., a track). By exploiting the I/O parallelism of such a multidisk log, the response time penalty of the deferred write approach could be eliminated, even with much larger persistence spheres than we observed in our experiments.

Sensitivity of the number of foreign-subobject accesses:

The performance results with varying numbers of foreign-subobject accesses per complex-object operation are shown in Figure 10. These results essentially reconfirm the above observations. That is, with increasing number of foreign-subobject accesses, transactions loose locality which leads to more conflicts with S1 and potentially larger persistence spheres with S2/PS.

4. Comparison with Related Work

Multi-level transaction management methods are implemented in the commercial database systems SQL/DS (which is essentially System R [Gr81]), Synapse [Ong84], and Informix-Turbo [Cu88]. These systems deal with transaction management at two levels: the record level and the page level. Their recovery methods use record-level redo, which slows down recovery at a warmstart; and they ensure the atomicity of record-level operations (including index updates) by periodically taking operation-consistent checkpoints that write all dirty pages back into the database. Such checkpoints adversely affect transaction response time, and become increasingly unacceptable with evergrowing buffer pool sizes.

An interesting unconventional multi-level recovery architecture has been implemented in the research prototype Kardamom [Bü88]. In this system, high-level update operations are performed on an object cache, and the propagation of updates onto pages is deferred until EOT. Thus, no high-level undo log records are needed, at the expense of performing redo at the object level. This approach may be well suited for a server-workstation environment where data is exchanged at the object level. However, it does not become clear from the description of the algorithm if and how the approach can ensure the atomicity of high-level updates that are propagated onto pages during a transaction's commit phase.

Our method of multi-level recovery is most closely related to the ARIES method [Mo89, ML89, MP91]. Even though the two methods were independently devel-

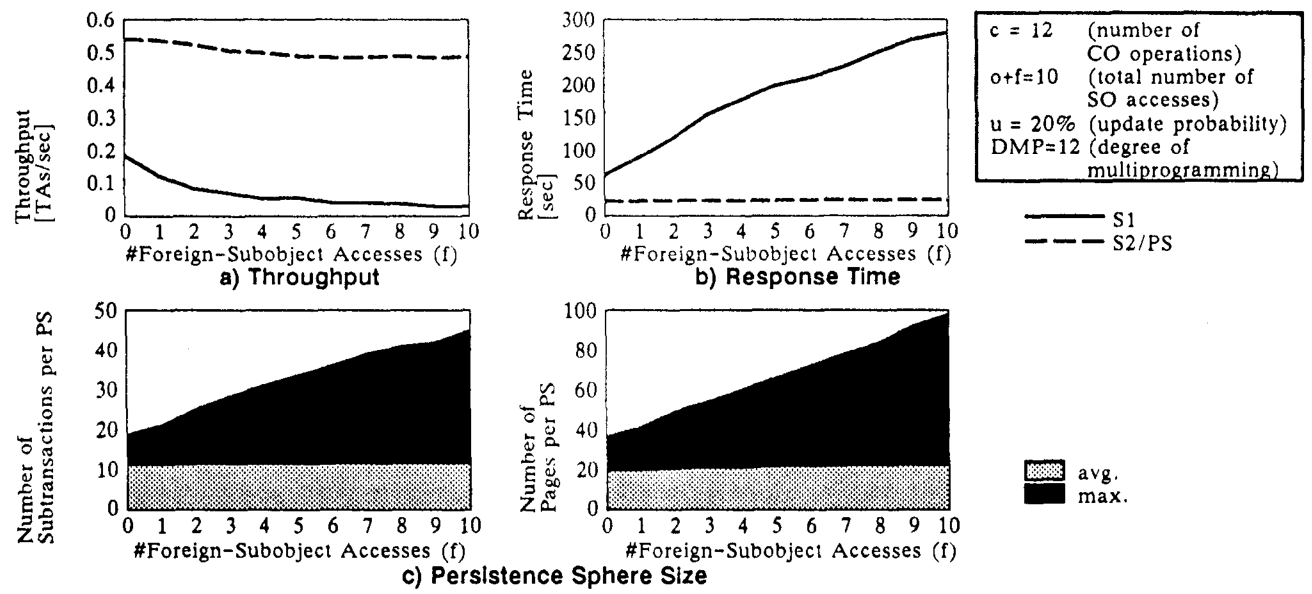


Fig. 10: Sensitivity of the Number of Foreign-Subobject Accesses (f)

oped with very different design objectives, they have quite a few properties in common, as discussed in the following.

- Both methods perform redo at the page level (i.e., "physical redo" in the terms of [Mo89]), thus minimizing the redo costs during a warmstart.
- Both methods support semantic concurrency control in that they allow commutative update operations on the same object to be performed concurrently. In such a case, both methods consequently perform transaction undo by compensation rather than restoring previous object states.
- As an unavoidable consequence of the above two properties, both methods may have to redo updates of "loser transactions" that are afterwards undone by compensation during a warmstart. This principle is called the "repeating of history" paradigm in [Mo89].
- To keep track of the modifications that are made by compensating (subtrans-) actions, both methods write a high-level log record when performing a compensating (subtrans-) action. These log records are called "compensation log records" (CLRs) in [Mo89].

Given these common properties, a simplified comparative view of our multi-level recovery method and the ARIES method is the following. Our method could "emulate" ARIES by 1) performing entry logging rather than after-image logging at the page level, 2) combining the L1 log and the L0 log into a single physical log file, 3) adding a compensation backward chain between L1 log records to avoid undoing undo operations [Mo89], and 4) simply flushing all buffered log records whenever a persistence sphere has to be written. While the first three of these points would be (relatively simple) modifications or extensions of our method (see also [WH91]), the fourth point would actually be a simplification, at the expense of writing more log records (see below).

The similarity of ARIES and our method is especially remarkable because the two methods have been developed with very different design goals in mind. ARIES is an industrial-strength recovery method for relational DBMSs that is tailored to the prevalent storage structures of relational systems. The multi-level recovery method, on the other hand, evolved from a theoretically well-founded but relatively puristic framework, aiming at high modularity and generality in that it can handle arbitrarily complex high-level operations. These different objectives have led to the following two important, subtle differences.

- ARIES presumes that each high-level undo log record refers to exactly one page. This ensures the atomicity of high-level operations. The restriction is acceptable if one has only relational DBMS storage structures in mind, for a regular tuple always resides in one page, and page locking is considered to be good enough for long fields. The only significant problem arises with indexes, where a single high-level operation such as inserting a key may update multiple index pages because of index restructuring such as splits. In ARIES [ML89], such cases are dealt with by dividing a multi-page high-level update operation into a multi-page update that does not require (high-level) undo at all, and a single-page update that can be compensated.

  In the example of inserting a new key that causes a split, the multi-page update is the split itself without inserting the key, and the atomic single-page update is the insertion of the new key into the restructured index. The atomicity of the multi-page update is in turn ensured by executing it as a "nested top action", which means that it is made persistent immediately upon its completion (and independently of the transaction to which it belongs). This method entails that page-level undo and redo information is recorded during the operation's execution, but no high-level undo log record is written. Thus, the split operation cannot be undone once it is completed. While this is perfectly reasonable for index restructuring operations, bundling subtransaction atomicity together with persistence is clearly unacceptable for arbitrary multi-page update operations on complex objects. In our implementation of multi-level transaction management, ensuring subtransaction atomicity so that high-level compensation is feasible even for complex multi-page updates was one of the major challenges. It has been solved efficiently without sacrificing generality.

- A less important yet remarkable difference between ARIES and our method is the amount of redo processing during a warmstart. Persistence spheres, as used in our method, are the minimal sets of redo log records that need to be written in order to ensure transaction persistence by page-level redo while observing subtransaction atomicity. ARIES, on the other hand, writes all generated log records to disk, which is much simpler. During the warmstart, ARIES therefore redoes all updates up to the point of the crash. The enhanced version called ARIES/RRH [MP91] avoids some of this redo work by checking, during the redo pass, if a redo log record of a loser transaction is followed by a redo log record of a winner transaction that refers to the same page. The update of the loser transaction need not be redone if (and, in ARIES/RRH, only if) this is not the case. In our method, such a check (which may even require look-ahead in the log [MP91]) is unnecessary because the critical redo log record would have been written to the log file only if the subtransaction that generated the log record were followed by a winner transaction that modified the critical page or if the dirty page were written back into the database before the crash occured.

5. Conclusions

The implemented method of multi-level transaction management has the following advantages.

- It allows exploiting the semantics of high-level operations to enhance concurrency.
- Our algorithms can deal with complex high-level operations on arbitrarily complex objects. In particular, it ensures the atomicity of high-level operations that modify multiple pages. This is a fundamental prerequisite for correctly dealing with compensation of high-level operations.
- These advantages are achieved at about the same log I/O costs that an efficient page-oriented single-level recovery method has. Our method does not require a costly checkpoint mechanism, and it provides fast recovery after a crash.
- Our implementation supports also parallelism within a transaction.

The performance of our implementation, within the research prototype DASDBS, is encouraging despite an obvious lack of fine-tuning at the code level. Nevertheless, we are investigating various issues for improving the performance under specifically heavy load situations. These issues include (see [WH91, WHMZ90] for further discussion):

- "light-weight" subtransactions that exploit specific reference patterns at the page level,
- multi-granularity locking on complex objects,
- alternative organizations of log buffers and log files (e.g., combining the L1 log and the L0 log into a single physical file),
- log file partitioning such that all partitions can be processed independently and in parallel during a warmstart, and
- applying multi-level transaction management in a distributed environment with both data distribution and function distribution (as in a server-client architecture).

Acknowledgement

The method for deferred log writes and the concept of persistence spheres were designed jointly with Peter Brössler and Peter Muth. Their contribution is gratefully acknowledged. Further, we would like to thank the UBILAB of the Union Bank of Switzerland (Schweizerische Bankgesellschaft) and especially Prof. Rudolf Marty for supporting our work.

References

[Bü88] von Bültzingsloewen, G., Iochpe, C., Liedtke, R.-P., Dittrich, K.R., Lockemann, P.C., Two-Level Transaction Management in a Multiprocessor Database Machine, 3rd Int. Conf. on Data and Knowledge Bases, Jerusalem, 1988

[BBG89] Beeri, C., Bernstein, P.A., Goodman, N., A Model for Concurrency in Nested Transactions Systems, Journal of the ACM Vol.36 No.1, 1989

[BR90] Badrinath, B.R., Ramamritham, K., Performance Evaluation of Semantics-based Multilevel Concurrency Control Protocols, ACM SIGMOD Conf., 1990

[BF89] Broessler, P., Freisleben, B., Transactions on Persistent Objects, Int. Workshop on Persistent Object Systems, Newcastle, Australia, 1989

[BSW88] Beeri, C., Schek, H.-J., Weikum, G., Multi-Level Transaction Management, Theoretical Art or Practical Need?, 1st Int. Conf. on Extending Database Technology, Venice, 1988, Springer, LNCS 303

[CF90] Cart, M., Ferrie, J., Integrating Concurrency Control into an Object-Oriented Database System, 2nd Int. Conf. on Extending Database Technology, Venice, 1990, Springer, LNCS 416

[Cu88] Curtis, R.B., Informix-Turbo, IEEE COMPCON, 1988

[DFMV90] DeWitt, D.J., Futtersack, P., Maier, D., Velez, F., A Study of Three Alternative Workstation-Server Architectures for Object Oriented Database Systems, VLDB Conf., 1990

[EB84] Elhardt, K., Bayer, R., A Database Cache for High Performance and Fast Restart in Database Systems, ACM TODS Vol.9 No.4, 1984

[FLMW88] Fekete, A., Lynch, N., Merritt, M., Weihl, W., Commutativity-Based Locking for Nested Transactions, Technical Report MIT/LCS/TM-370, MIT, Cambridge (Mass.), 1988, to appear in: Journal of Computer and System Sciences

[Ga83] Garcia-Molina, H., Using Semantic Knowledge for Transaction Processing in a Distributed Database, ACM TODS Vol.8 No.2, 1983

[Gr81] Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., Traiger, I., The Recovery Manager of the System R Database Manager, ACM Computing Surveys Vol.13 No.2, 1981

[GK85] Gawlick, D., Kinkade, D., Varieties of Concurrency Control in IMS/VS Fast Path, IEEE Database Engineering Vol.8 No.2, 1985

[GS87] Garcia-Molina, H., Salem, K., Sagas, ACM SIGMOD Conf., 1987

[GT90] Graefe, G., Thakkar, S.S., Tuning a Parallel Database System on a Shared-Memory Multiprocessor, Technical Report, University of Colorado at Boulder, 1990

[Hä87] Härder, T., On Selected Performance Issues of Database Systems, 4th German Conf. on Performance Modeling of Computing Systems, Springer, 1987

[HH88] Hadzilacos, T., Hadzilacos, V., Transaction Synchronization in Object Bases, ACM PODS Conf., 1988

[HR83] Härder, T., Reuter, A., Principles of Transaction-Oriented Database Recovery, ACM Computing Surveys Vol.15 No.4, 1983

[KLS90] Korth, H.F., Levy, E., Silberschatz, A., Compensating Transactions: A New Recovery Paradigm, VLDB Conf., 1990

[Ma87] Martin, B.E., Modeling Concurrent Activities with Nested Objects, Int. Conf. on Distributed Computing Systems, Berlin, 1987

[Mo85] Moss, J.E.B., Nested Transactions: An Approach to Reliable Distributed Computing, MIT Press, 1985

[Mo89] Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., Schwarz, P., ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging, IBM Res. Report RJ6649, San Jose, 1989, to appear in: ACM TODS

[MGG86] Moss, J.E.B., Griffeth, N.D., Graham, M.H., Abstraction in Recovery Management, ACM SIGMOD Conf., 1986

[ML89] Mohan, C., Levine, F., ARIES/IM: An Efficient and High Concurrency Index Management Method Using Write-Ahead Logging, IBM Res. Report RJ6846, San Jose, 1989

[MLC87] Moss, J.E.B., Leban, B., Chrysanthis, P.K., Finer Grained Concurrency for the Database Cache, IEEE Conf. on Data Engineering, 1987

[MP91] Mohan, C., Pirahesh, H., ARIES-RRH: Restricted Repeating of History in the ARIES Transaction Recovery Method, IEEE Conf. on Data Engineering, 1991

[MR91] Muth, P., Rakow, T., Atomic Commitment for Integrated Database Systems, IEEE Conf. on Data Engineering, 1991

[Ong84] Ong, K.S., Synapse Approach to Database Recovery, PODS Conf., 1984

[O'N86] O'Neil, P.E., The Escrow Transactional Method, ACM TODS Vol.11 No.4, 1986

[RGN90] Rakow, T.C., Gu, J., Neuhold, E.J., Serializability in Object-Oriented Database Systems, IEEE Conf. on Data Engineering, 1990

[Sh85] Shasha, D., What Good Are Concurrent Search Structure Algorithms for Databases Anyway?, IEEE Database Engineering Vol.8 No.2, 1985

[SDP91] Shrivastava, S.K., Dixon, G.N., Parrington, G.D., An Overview of the Arjuna Distributed Programming System, IEEE Software, January 1991

[SPSW90] Schek, H.-J., Paul, H.-B., Scholl, M.H., Weikum, G., The DASDBS Project: Objectives, Experiences, and Future Prospects, IEEE Transactions on Knowledge and Data Engineering Vol.2 No.1, 1990

[SG88] Shasha, D., Goodman, N., Concurrent Search Structure Algorithms, ACM TODS Vol.13 No.1, 1988

[SS84] Schwarz, P.M., Spector, A.Z., Synchronizing Shared Abstract Types, ACM Transactions on Computer Systems Vol.2 No.3, 1984

[SZ89] Skarra, A.H., Zdonik, S.B., Concurrency Control and Object-Oriented Databases, in: W. Kim, F.H. Lochovsky (eds.), Object-Oriented Concepts, Databases, and Applications, ACM Press, 1989

[We88] Weihl, W.E., Commutativity-Based Concurrency Control for Abstract Data Types, IEEE Transactions on Computers Vol.37 No.12, 1988

[We89] Weihl, W.E., The Impact of Recovery on Concurrency Control, ACM PODS Conf., 1989

[WS84] Weikum, G., Schek, H.-J., Architectural Issues of Transaction Management in Layered Systems, VLDB Conf., 1984

[WS91] Weikum, G., Schek, H.-J., Multi-Level Transactions and Open Nested Transactions, IEEE Data Engineering Vol.14 No.1, 1991

[Wei86] Weikum, G., A Theoretical Foundation of Multi-Level Concurrency Control, ACM PODS Conf., 1986

[Wei87] Weikum, G., Enhancing Concurrency in Layered Systems, Proc. 2nd Int. Workshop on High Performance Transaction Systems, 1987, Springer, LNCS 359, 1989

[Wei91] Weikum, G., Principles and Realization Strategies of Multilevel Transaction Management, ACM TODS Vol.16 No.1, 1991

[WH91] Weikum, G., Hasse, C., Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism, Technical Report, Computer Science Dept., ETH Zurich, 1991

[WHBM90] Weikum, G., Hasse, C., Broessler, P., Muth, P., Multi-Level Recovery, ACM PODS Conf., 1990

[WHMZ90] Weikum, G., Hasse, C., Mönkeberg, A., Zabback, P., The COMFORT Project: A Comfortable Way to Better Performance, Technical Report 137, Computer Science Dept., ETH Zurich, 1990