VLDB Journal, 2, 407-453 (1993), Gunter Schlageter, Editor

Multi-Level Transaction Management for Complex Objects: Implementation, Performance, Parallelism

Gerhard Weikum and Christof Hasse

Received June 18, 1991; revised version received, January 28, 1992; accepted April 29, 1993.

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. They allow the exploitation of semantics of high-level operations to increase concurrency. As a consequence, undoing a transaction requires compensation of completed subtransactions. In addition, multi-level recovery methods must take into consideration that high-level operations are not necessarily atomic if multiple pages are updated in a single subtransaction. This article presents algorithms for multi-level transaction management that are implemented in the database kernel system (DASDBS). In particular, we show 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. We show that multi-level transaction management can be extended easily to cope with parallel subtransactions within a single transaction. Performance results are presented with varying degrees of inter- and intra-transaction parallelism.

Key Words. Atomicity, complex objects, inter- and intratransaction parallelism, multi-level transactions, performance, persistence, recovery.

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 (Beeri et al., 1988). The purpose of multi-level transactions is to allow the exploitation of the semantics of high-level operations to increase concurrency. For example, two "deposit" operations on a bank account are commutative and therefore could be admitted concurrently, such as transactions transferred on behalf of two funds. However, executing such high-level operations in parallel requires

Gerhard Weikum, Dr.-Ing., is Professor and Christof Hasse, Dipl.-Inform., is Research Assistant, Department of Computer Science, ETH Zurich, CH-8092, Zurich, Switzerland.

that a low-level synchronization mechanism take care of possible low-level conflicts (e.g., on indexes or data pages). In relational database management systems (DBMSs) in which records do not span pages, this low-level synchronization is usually implemented by page latches—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 because it cannot ensure the indivisibility of arbitrary multi-page update operations. Rather, high-level operations must 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 is independent of lower-level conflicts.

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

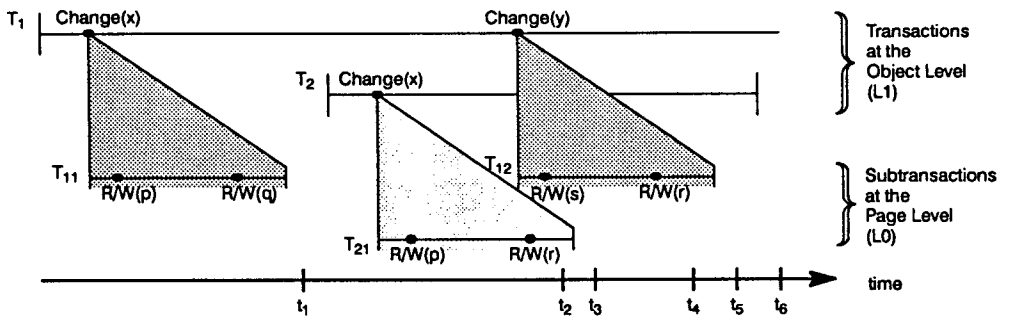
- At the *object level L1,* semantic locks are dynamically acquired and held until end-of-transaction (EOT) according to the strict two-phase locking protocol. The semantics of the high-level operations are exploited in the lock modes and the lock mode compatibility table, which is in turn derived from the commutativity properties or semantic compatibility (Garcia-Molina, 1983; Skarra and Zdonik, 1989) of the operations. In principle, one could even exploit state-dependent commutativity (O'Neil, 1986; Weihl, 1988), but this is beyond the scope of this article.

- 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 the locks of conventional nested transactions (Moss, 1985), 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. Assume 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, because they may access many subobjects of the document (e.g., when the layout of the entire document is recomputed), the potential conflicts at the lower level must be reconciled. In Figure 1, this is done by acquiring locks on the underlying pages that are released at the end of subtransactions T11, T12, and T21, respectively.

Similar examples arise in advanced business applications with large amounts of derived data. In foreign exchange transactions, a forward transaction (e.g., a

Figure 1. Parallel execution of two multi-level transactions



currency swap) may have to compute a large number of future positions for risk assessment, such as 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. Because 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* (Garcia-Molina, 1983; Moss et al., 1986; Garcia-Molina and Salem, 1987; Weikum, 1987; Beeri et al., 1988; Weihl, 1989; Korth et al., 1990; Shrivastava et al., 1991; Weikum, 1991; Gray and Reuter, 1993). In Figure 1, undoing transaction T1 would require two inverse *Change* operations on y and x, i.e., two additional subtransactions that compensate the completed subtransactions T12 and T11 (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 are 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 reflect only partially the primary updates. If a subtransaction modifies multiple pages (Figure 1), a low-level recovery mechanism at the page level is necessary 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 (Weikum and Schek, 1984, 1991, 1992; Shasha, 1985; Moss et al., 1986; Weikum, 1986, 1987, 1991; Martin, 1987; Garcia-Molina and Salem, 1987; Beeri et al., 1988, 1989; von Bueltzingsloewen et al., 1988; Fekete et al., 1988; Hadzilacos

and Hadzilacos, 1988; Shasha and Goodman, 1988; Broessler and Freisleben, 1989; Badrinath and Ramamritham, 1990; Cart and Ferrie, 1990; Rakow et al., 1990; Weikum et al., 1990; Muth and Rakow, 1991; Shrivastava et al., 1991; Muth et al., 1993.) However, to our knowledge, none of the previous work has presented a full implementation. Furthermore, only two articles have presented performance figures. Weikum (1991) reported 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. Badrinath and Ramamritham (1990) reported simulation results on multi-level concurrency control only; i.e., disregarding recovery issues.

Our article makes the following novel contributions:

- It shows how multi-level transaction management can be implemented efficiently. The implementation is integrated in the database kernel system (DASDBS) (Schek et al., 1990).

- It presents performance measurements of the implemented system, based on a synthetic benchmark for complex-object processing.

- It shows how multi-level transaction management can be extended so that subtransactions of the same transaction can be executed in parallel. Performance results are presented with varying degrees of inter- and intra-transaction parallelism.

Parts of this article have been published (Hasse and Weikum, 1991). Here we discuss the implementation of multi-level recovery in much more detail, we discuss additional performance experiments, and we add the issue of intra-transaction parallelism including preliminary performance results. Our discussion of recovery covers transaction aborts, subtransaction aborts (i.e., partial rollbacks of transactions), and crash recovery from system failures (which implies losing all memory-resident data). For these cases, we assume that crash-resilient stable storage is available, and that writes to this stable storage persist beyond system failures. We do not discuss media recovery (i.e., recovery from media failures such as unreadable disk pages), because this issue does not require anything specific to multi-level transactions. Media recovery can be achieved for both flat and (open or closed) nested transactions by log-based techniques (Haerder and Reuter, 1983; Mohan et al., 1992; Gray and Reuter, 1993) or by RAID-like redundancy at the disk level (Patterson et al., 1988; Copeland and Keller, 1989; Gibson, 1992).

The rest of this article 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 simple extensions that we have made to cope with intra-transaction parallelism. Section 4 discusses the results of a comprehensive series of performance experiments. Section 5 compares our implementation with related work, especially the ARIES recovery method (Mohan et al.,

1992). Section 6 discusses several options for further improving the performance of multi-level transaction management.

2. Implementation of Multi-level Transaction Management in DASDBS

2.1 Lock Management

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, simple conjunctive predicates, etc.). Dynamic allocation of lock control blocks is implemented by a tunable shared-memory heap manager that is optimized toward frequent disposals and reallocations of memory fragments of particular sizes. Deadlock detection is implemented by using an algorithm for partial transitive closures, which is invoked on each lock conflict. This algorithm operates on an $m*m$ wait-for matrix where m is the maximum degree of multiprogramming that was specified at system startup time.

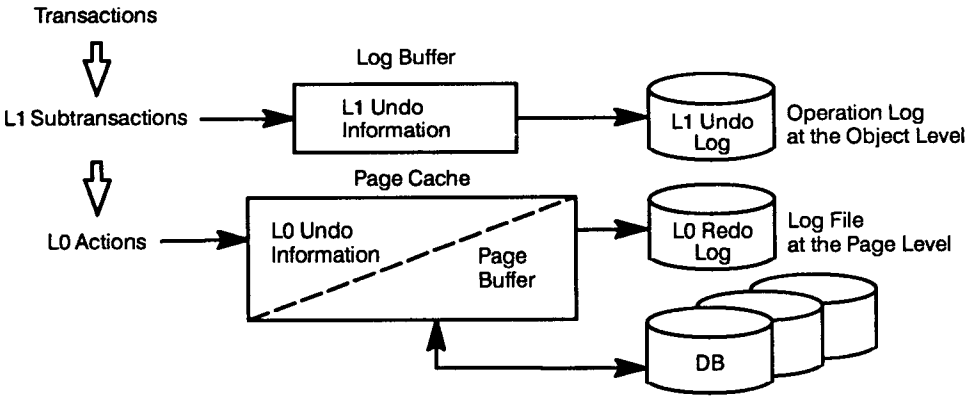
In addition to the usual "shared" and "exclusive" lock modes, semantic lock modes such as "increment" can be incorporated by specifying the lock mode compatibility matrix at the creation time of a lock table (Schwarz and Spector, 1984). In the performance experiments (Section 4), this feature was not exploited; rather, shared and exclusive locks were acquired on sets of object identifiers.

2.2 Recovery Management

This subsection contains an in-depth discussion of our implementation of multi-level recovery. The implemented algorithms are based on the methodical framework of Weikum et al. (1990); here we give an implementation-oriented algorithmic description. In Subsection 2.2.1, we present a simple "strawman" algorithm that is based on applying the DB Cache method (Elhardt and Bayer, 1984) to page-level subtransactions. The strawman algorithm provides correctness but has potential performance problems in that it may cause excessive log I/Os. Therefore, the algorithm is refined in Subsection 2.2.2 by adding the concept of deferred log writes. Note that deferring log writes may be straightforward in a single-level recovery method with page locking, but this incurs significant problems in multi-level transaction management where multi-page update subtransactions of incomplete (i.e., uncommitted) transactions may have modified common pages. In Subsections 2.2.2 and 2.2.3, we show how our implementation solves these problems, thus saving a substantial amount of log I/Os. Finally, in Subsection 2.2.4, we discuss the idempotence problem that arises during the warmstart, and present our approach to coping with non-idempotent high-level operations.

2.2.1 Requirements and Overall Approach. A method for multi-level recovery must satisfy the following requirements:

Figure 2. Architecture of the DASDBS multi-level transaction management



1. It must ensure that transactions are atomic.

2. It must ensure that transactions are persistent.

3. It must ensure that subtransactions are atomic. (Note that subtransactions need not be persistent before the commitment of their parent. In addition to the above requirements for correctness, the following performance requirement is reasonable to guarantee an acceptable recovery time and hence high availability of the DBMS.)

4. During a warmstart, redo (of committed transactions) should be performed at the bottom level L0 (i.e., by reconstructing pages rather than re-executing potentially resource-intensive high-level operations).

An architecture that meets the above requirements is shown in Figure 2. For requirement 1, 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. The log records of a transaction are chained together in a backward chain for handling transaction aborts and for performing transaction undo after a crash. In addition to the L1 operation log records, EOT log records are written for completed (i.e., committed or aborted) transactions.

For requirements 2 and 4, redo log records are written to an L0 log file. Requirement 4 can be implemented either by logging page modifications (i.e., modified bytes; Lindsay et al., 1979; Crus, 1984; Moss et al., 1987; Mohan et al., 1992), or by writing entire page after-images as in the DB Cache method (Elhardt and Bayer, 1984). The first option, which is usually referred to as "entry logging," causes less log volume (i.e., it saves log space) and may thus have shorter log I/Os. Note, however, that after-image logging does not cause a higher number of log I/Os,

given 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. Thus, after-image logging saves a substantial number of random I/Os during the warmstart. For this reason and for simplicity, we assume in the following that after-image logging is used, as it is actually implemented in DASDBS by applying the DB Cache method to page-level subtransactions. Note, however, that most considerations of this article fit with entry logging as well.

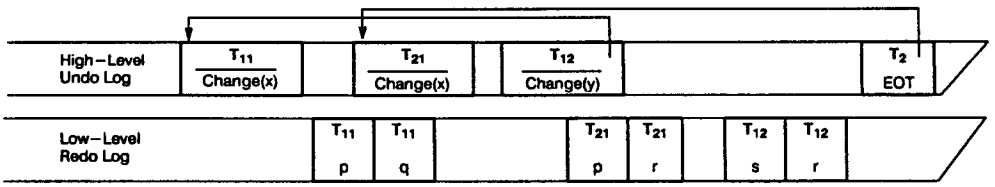
Ensuring Subtransaction Atomicity. Requirement 3, subtransaction atomicity, is the one that makes multi-level recovery difficult. Essentially, it is handled by using page before-images as the L0 undo information. Because these before-images are needed only for incomplete subtransactions, they are kept in main memory as temporary page versions in the buffer pool. This provides an efficient method for undoing a subtransaction (e.g., to resolve a page-level deadlock between subtransactions).

Unfortunately, a complete solution for subtransaction atomicity is not quite that simple. If a dirty (i.e., modified) page were replaced in the buffer pool and written back to the database before a subtransaction completes, the before-image of the page would have to be written to disk first, according to the write-ahead logging (WAL) rule (Bernstein et al., 1987). This problem is circumvented in our implementation by assuming (and having implemented) a No-Steal buffer manager (Haerder and Reuter, 1983) that does not replace a dirty page before EOS. Note that a No-Steal policy for subtransactions is feasible because subtransactions usually have bounded length, whereas the same assumption for arbitrarily long transactions may be debatable (Elhardt and Bayer, 1984).

A more severe problem is that replacing a dirty page is critical even after the completion of the subtransaction that has modified the page. Again, this is a violation of the WAL rule. Moreover, this would violate the atomicity of a subtransaction if it has modified multiple pages. One solution could be to keep the before-images of a subtransaction beyond EOS. Because transactions become persistent upon EOT and are no longer eligible for rollback after EOT, it seems to be sufficient to keep the before-images of a subtransaction until the EOT of its transaction. However, even this approach is not sufficient for ensuring subtransaction atomicity. To verify this claim, consider the following scenario.

Consider again the example in Figure 1. Assume that the dirty page p is replaced in the buffer pool and written back to disk at time t1, and that the system fails right after this point. Writing p back to the database violates the atomicity of subtransaction T11, because T11 has modified two pages, p and q. Because all memory-resident before-images would be lost after the crash, it would be impossible to undo the partial effect that T11 leaves on the permanent database.

Figure 3. Log contents for the example of Figure 1



To overcome this problem, one might consider writing before-images to stable storage immediately upon their generation. Unfortunately, even this fairly inefficient method cannot solve the general problem of subtransaction atomicity. Assume that page *p* (Figure 1) is replaced in the buffer pool at time t6. If a crash occurred right after t6, using the before-images of T11 to reestablish the atomicity of T11 would incidentally undo the updates of T21 on page *p*. This in turn would violate the atomicity of T21, and, even worse, would violate the persistence of T2, which is already committed at time t6. This example shows that a subtransaction cannot be undone in an isolated manner by means of page before-images once the subtransaction is completed and its modified pages become visible to and may be overwritten by other transactions.

Our solution for ensuring the atomicity of a subtransaction when a dirty page is replaced after EOS is to force the L0 redo information of the subtransaction to the L0 log file. The after-images of a subtransaction are written atomically, by including a special EOS flag in the header of the last page of the written after-images. This flag serves as an EOS log record.

So far, we have not discussed when the after-images of a subtransaction are written to disk. In fact, this is the most critical point of our recovery method. Because of its importance, this issue is discussed separately in Subsections 2.2.2 and 2.2.3. For now, we assume that a subtransaction's after-images are forced to the log file immediately after EOS. While this is obviously inefficient, it is a correct multi-level recovery method and was in fact the first method implemented in (a former version of) DASDBS. It is worthwhile to note that this method is essentially the DB Cache method (Elhardt and Bayer, 1984) applied to subtransactions. The DB Cache method is one of the most efficient recovery methods, and has nice properties with respect to how the log space is managed (i.e., dynamically compacted without having to take checkpoints; Elhardt and Bayer, 1984). Its main drawback is that it works only in combination with page locking. This disadvantage does not hold for our multi-level transaction management, because we employ the DB Cache method only to handle subtransactions at the page level.

The log records that are written for the example of Figure 1 are shown in Figure 3. Operations with an overbar denote inverse operations. For reasons discussed in Subsection 2.2.4, the L1 log records and the after-images that are written at L0 include a subtransaction identifier in a special header field.

Transaction aborts are implemented by scanning the backward chain of the L1 undo log records (starting from a memory-resident anchor not shown in Figure 3), and applying the recorded high-level undo operations to the database. This method is directly applicable if the aborted transaction does not have any incomplete subtransactions. If there is an incomplete subtransaction, then this subtransaction must be undone first by means of the before-images (not shown in Figure 3) that are kept in memory until EOS, and then the completed subtransactions are undone as described above. The memory-resident before-images also serve to abort individual subtransactions (e.g., when an L0 deadlock occurs); such aborted subtransactions may be re-executed automatically.

The warmstart after a crash consists of the following two steps:

1. *Redo pass:* Determine the relevant starting point in the L0 redo log by looking up a special master record (Elhardt and Bayer, 1984), and perform a forward pass on the L0 redo log. During this pass, 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 at acceptable performance (i.e., requirements 2, 3, and 4). After-images after the latest EOS-flagged after-image are ignored since they belong to incomplete writes 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 EOT log records are found are winners and thus do not need any processing. For loser transactions, compensating subtransactions are performed according to the contents of their log records.

2.2.2 Deferred Log Writes. The multi-level recovery algorithm of the previous subsection was implemented in a former version of DASDBS (Schek et al., 1990). 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 (which in turn requires forcing the L1 undo log before, so as to observe the WAL rule). In the example of Figure 1, this means that an after-image of page p is written to disk at the EOS of T11 and the EOS of T21, 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 (Gawlick and Kinkade, 1985; Helland et al., 1987), there is a more fundamental way to cut 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 be a significant gain in terms of the number of log I/Os, even if the number of after-images written for the entire transaction is not reduced (i.e., if the writesets of all subtransactions are disjoint). However, often it may be the case that subsequent subtransactions of the same transaction modify the same page. In this case, only the latest after-image

should be written at all. These optimizations would make multi-level logging as efficient as conventional single-level logging (e.g., the original DB Cache method; Elhardt and Bayer, 1984). The optimization to write only the latest after-image of a page has the additional benefit that after-images can be embedded in the regular buffers rather than in a separate L0 log buffer. So, for each page *p,* the latest after-image of *p* resides in a regular buffer frame, and there are no other versions of *p* in memory as long as none of the active subtransactions requests to modify *p.* When a subtransaction requests to modify *p,* then *p* is copied into a second buffer frame, and the updates are made on the copy, so that the previous after-image serves as the temporary before-image of *p* while the subtransaction is in progress. At EOS, the previous after-image (i.e., the current before-image) is discarded, and the newly modified copy becomes the latest after-image of *p.* There are never more than two versions of a page at the same time, since page locks are kept for the duration of a subtransaction.

Unfortunately, deferring all L0 log writes until EOT is not a correct solution. There may be subtransactions of different transactions so that the after-image sets of the subtransactions are overlapping (i.e., they 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 the other subtransaction.

Ideally, we would not want to write the latest after-images of the writesets of T11 and T12 before the EOT of T1 (Figure 1). The EOT of T2 requires writing the latest after-images of the writeset of T21 (i.e., pages *p* and *r*) as of the EOT time of T2. Writing these pages to the L0 log, however, would implicitly write the modifications that T11 made on *p,* too. Then, if the system crashed before the EOT of T1 (i.e., before the latest after-images of the writeset of T11 are written), the redo pass of the warmstart would violate the atomicity of T11 by restoring the update on *p* while disregarding T11's update on *q.* Note that this problem would arise also with entry logging, because subtransactions of different transactions may have modified a common byte through commutative high-level update operations. The same problem arises with respect to subtransaction T12. At the EOT of T2, only one version of page *r* resides in the buffer pool. This version contains the updates of the completed subtransaction T12. Thus, writing this after-image of *r* to the L0 log file would violate the atomicity of T12.

These and other related problems have been discussed more rigorously by Weikum et al. (1990) and have led to a solution based on the notion of *persistence spheres.* The basic idea to ensure subtransaction atomicity is that the writing of a page to the log (or to the database) causes other pages to be forced to the log. The set of pages that need to be forced to the L0 log when one of the pages in the writeset of a subtransaction Tij is written is called the persistence sphere of Tij. The definition of a persistence sphere PS(Tij) of a subtransaction Tij is based on the following "forces" relationship $\rightarrow$ between completed subtransactions. Tij $\rightarrow$ Tkl (pronounced: Tij forces Tkl) if Tij has modified a page that has been

modified by Tkl but has not been written to the L0 log or to the database since these modifications, or if Tij reads a page that was previously modified by Tkl and was not yet written to the L0 log or to the database. So we have the relationship that Tij $\rightarrow$ Tkl if there is a write-write or write-read dependency from Tkl to Tij.

Note that the relationship $\rightarrow$ is asymmetric and is defined dynamically (in the sense that it varies with the progress of the transaction execution and also depends on the log I/Os and buffer replacements that take place). The relation $\rightarrow$, which is defined between subtransactions, can be extended to pages in the following way. For pages p and q, we have $p \rightarrow q$ if (1) p and q have been modified by subtransactions Tij and Tkl, respectively, such that Tij $\rightarrow$ Tkl and (2) neither of the two pages has been written to the L0 log or back into the database since these modifications.

It is fairly obvious to see why subtransactions with overlapping writesets must be forced to the log together (i.e., combined into the same persistence sphere). This is the key to solving the problem of subtransaction atomicity discussed above. Write-read dependencies need to be considered, too, because the implementation of Tij (i.e., the read and write actions issued by Tij) may depend on the fact that Tkl performed a particular update. If Tij needs to be redone after a crash, this update of Tkl must be redone as well (Weikum et al., 1990; see also Mohan et al., 1992, for a discussion of the "repeating of history" paradigm).

The notion of a persistence sphere is defined as follows. The persistence sphere PS(Tij) of a completed subtransaction Tij is the smallest set of pages that satisfies both of the following two properties:

1. PS(Tij) contains all pages that have been modified by Tij and have not yet been written to the L0 log since the EOS of Tij, and

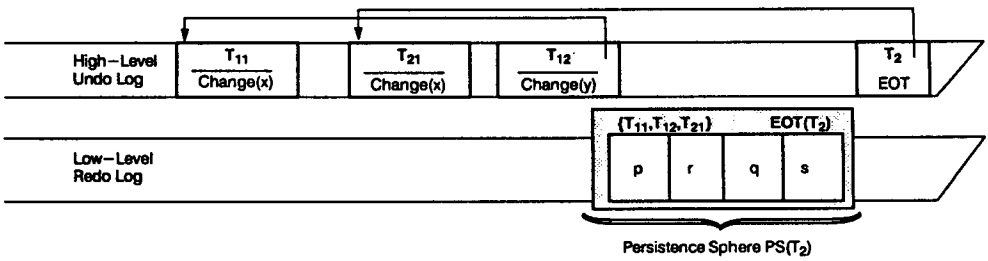
2. PS(Tij) is transitively closed with respect to the "forces" relationship $\rightarrow$.

The second condition simply states that, if a subtransaction Tij "forces" the subtransaction Tkl, then the persistence sphere of Tij contains the persistence sphere of Tkl. Finally, the persistence sphere PS(Ti) of a transaction Ti is defined as the union of the persistence spheres of its subtransactions.

Our solution to the deferred log write problem is the following. At the EOT of a transaction Ti, all pages in the persistence sphere of Ti must be written to the L0 log. 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. So writing a dirty page to the database and writing its after-image to the log are equivalent steps as far as the atomicity (and persistence) of subtransactions is concerned. At time t5 in Figure 1, the subtransaction T21 "forces" both T11 and T12. Thus, at the EOT of T2, the persistence sphere of T2 contains the pages p and r that were modified by T2's own subtransaction T21 and the pages q and s that were modified by T11 and T12, respectively.

Persistence spheres are written atomically to the L0 log file by setting a flag in the header of the last page (Subsection 2.2.1). A persistence sphere may contain

Figure 4. Log contents of the multi-level recovery method with deferred log writes



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 (thus observing the WAL rule). In addition, 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. On the other hand, it may turn out, at the EOT of a transaction, that all after-images of that transaction have already been written to the L0 log as parts of the persistence spheres of other transactions. In this case, the EOT log record of the committing transaction is forced to the L1 log disk, rather than performing an additional L0 log write. The log records for Figure 1 are shown in Figure 4.

2.2.3 Managing Persistence Spheres. In DASDBS, persistence spheres are implemented by means of the following types of control blocks:

- For each active transaction, a *transaction control block* (TCB) contains pointers to the subtransaction control blocks of its own subtransactions.

- For each subtransaction of an active transaction, a *subtransaction control block* (STCB) contains writeset pointers to the buffer frame control blocks of the pages that were modified by the subtransaction, and a readset list of the pages that were only read. The writeset pointers have backward pointers associated with them; i.e., the frame control block of a page points to the STCBs of all subtransactions that modified the page. The readset list is needed only for active subtransactions and can be discarded upon EOS (i.e., when the locks are released). An STCB is discarded as soon as the subtransaction's after-images are written to the L0 redo log file.

- For each buffer frame, a frame control block (FCB) contains status information about the page that is held in the buffer frame. The status can be

  1. "modified," which means that an incomplete (i.e., running) subtransaction has modified the page,
  2. "dirty," which means that a completed subtransaction has modified the page but the modified page has not yet been written to the L0 log or to the database,
  3. "forced," which means that a completed subtransaction has modified the page and this page has already been written to the L0 log, or
  4. "clean," which means that no incomplete transaction has modified the page and an identical version of the page resides in the database.

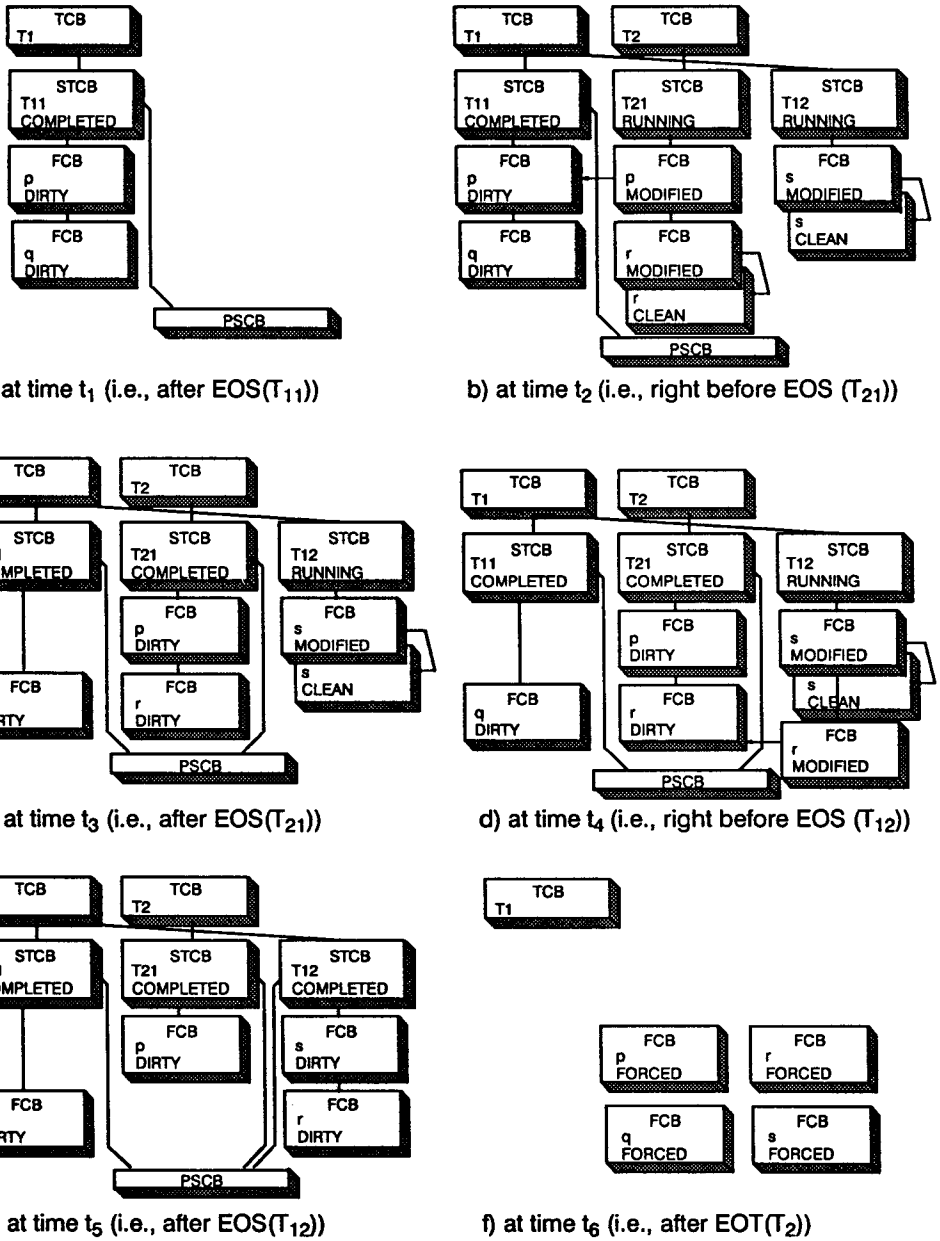
  FCBs with status "modified" contain a pointer (referred to as *bfim*) to another FCB that points to a before-image frame. For frequently modified pages, the before-image FCB is usually a "dirty" FCB of a previously completed subtransaction.

- Finally, for each persistence sphere, a persistence sphere control block (PSCB) points to the STCBs of those subtransactions that constitute the persistence sphere. These pointers have backward pointers associated with them (i.e., an STCB also points to its PSCB). Note that each STCB belongs to exactly one PSCB (see below). A PSCB and the STCBs that it points to are discarded as soon as the persistence sphere has been written to the log.

To keep track of pages that belong to a persistence sphere, we have chosen to use a simplified variant of the notion of a persistence sphere. The definition of a persistence sphere is based on the transitive closure of the "forces" relation $\rightarrow$ between subtransactions. In our implementation, we actually use the symmetric and transitive closure of the "forces" relation. The advantage of using a symmetric relation between subtransactions is that we can now simply merge the persistence spheres of two subtransactions whenever they have a page in common that is modified by one or both subtransactions. A consequence of this simplification is that some persistence spheres may become larger than they need to be. However, this disadvantage is outweighed by the simplification and the reduction of the bookkeeping overhead.

Managing persistence spheres by means of the control blocks introduced above is illustrated in Figure 5 (based on Figure 1). Figure 5 shows snapshots of the necessary control blocks at different points of time. Figure 6 shows pseudocode for the complete handling of BOT, BOS, page reads (i.e., fixing a page for read),

Figure 5. Snapshots of control blocks for the scenario of Figure 1



page modifications (i.e., fixing a page for write), EOS, EOT, and dirty page buffer replacements. Note that these procedures ensure that, at each point of time, a "dirty" page belongs to exactly one PSCB, but possibly to multiple STCBs, which are necessarily attached to the same PSCB. That is, if the page had been modified by

multiple completed subtransactions, then the persistence spheres of these subtransactions would already have been merged by the corresponding EOS procedure calls. Further note that the persistence spheres of a transaction's subtransactions are not merged before EOT or the occurrence of a page dependency (as checked at EOS). This "just-in-time" merging of persistence spheres aims to minimize the impact of transitive dependencies between subtransactions. This in turn keeps the number of pages in a persistence sphere as small as possible, and thus avoids excessively long log writes that might adversely affect transaction response time. Finally, note that the maintenance of the various lists and, especially, the managing of persistence spheres requires latching to protect critical sections. Note, however, that all of these critical sections have fairly short path lengths.

2.2.4 Warmstart Procedure. A nice property of our multi-level recovery algorithm is that, even with deferred log writes, the warmstart procedure after a crash is fairly simple. In fact, the warmstart processing can be directly adopted from the multi-level recovery algorithm without deferred log writes (Subsection 2.2.1). During the redo pass, the after-images of the L0 redo log are loaded into the buffer pool and are written into the database according to the buffer manager's write policy. Thus, all completed transactions and all subtransactions that were in the persistence sphere of a completed transaction are redone. During the subsequent backward pass on the L1 undo log, compensating subtransactions are invoked for those subtransactions that belong to loser transactions.

During the undo pass, a problem arises because the high-level undo log record of a subtransaction is always forced to the L1 log file before the subtransaction's after-images are written to the L0 log file. Because these two write operations are not performed separately, the undo pass during a warmstart may encounter an undo log record of a subtransaction, the after-images of which were not yet written to the L0 log file when the crash hit. For example, in Figure 4, assume that the system crashes right after the undo log record for T12 was written to the L1 log file. Then, since T11, T21, and T12 will not be redone during the warmstart, we must take care that the inverse high-level operations for these subtransactions will have no effect on the database. To guarantee this property even for arbitrary high-level operations, the recovery manager itself must figure out which high-level log records must be skipped during the undo pass. Note that this problem is essentially the problem of ensuring idempotence for non-idempotent operations such as "increment" operations. If we consider that a subtransaction's updates were lost in a crash as a fictitious undo operation, then we must guarantee that a "second" execution of the undo operation is prohibited or has no effect.

Figure 6. Pseudocode for multi-level logging

```
BOT(Ti):
    Create TCB

BOS(Tij):
    Create STCB
    Attach STCB(Tij) to TCB(Ti)

Fix_for_Read(Tij, q):
    Add q to STCB(Tij).readset

Fix_for_Write(Tij, q):
    Allocate a new buffer frame with FCB f
    Copy q into the new frame
    Attach the original FCB of q to f
    Set the status of f to "modified"
    Attach f to STCB(Tij).writeset

EOS(Tij):
    Initialize L, a list of PSCBs, to be empty
    for each
        FCB f in STCB(Tij).writeset
    do
        for each STCB s that points to bfim(f)
        do
            Add the PSCB of s to the list L
        od
        Set the status of FCB f to "dirty"
        Drop the FCB bfim(f)
    od
    for each
        page r in STCB(Tij).readset
    do
        if  there is an FCB f for r such that
            the status of f is "dirty"
        then
            for each STCB s that points to f
            do
                Add the PSCB of s to the list L
            od
        fi
    od
    if L is not empty
    then
        Merge_Persistence_Spheres(L)
        Let p be the resulting PSCB
    else
        Create a new PSCB p
    fi
    Attach STCB(Tij) to PSCB p

Replace_Dirty_Page(q):
    Determine an (arbitrary) STCB s that
      points to FCB(q)
    Determine the PSCB p that s points to
    Write persistence sphere p
    Write page q back into the database
    Set the status of FCB(q) to "clean"

EOT(Ti):
    Initialize L, a list of PSCBs, to be empty
    for each STCB s that points to TCB(Ti)
    do
        if s points to a PSCB
        then
            Add this PSCB to the list L
        fi
    od
    if L is not empty
    then
        Merge_Persistence_Spheres(L)
        Let p be the resulting PSCB
        Write_Persistence_Sphere(p)
    fi
    Drop TCB(Ti)

Write_Persistence_Sphere(p):
    Force the L1 log buffer
    Collect a list L of "dirty" FCBs
      by traversing all FCBs of all STCBs
      that are attached to PSCB p
    Write the pages that are pointed to
      by the FCBs in L to the L0 log,
      and set the status of the FCBs to "forced"
    Drop all STCBs that point to p
    Drop PSCB p

Merge_Persistence_Spheres(L):
    Merge all PSCBs in L by attaching all
      STCBs of the 2nd through last PSCB in L
      to the first PSCB p in L
    Drop the 2nd through last PSCB in L
```

Fig.6: Pseudocode for Multi-level Logging

Our solution to the described problem is to store the numbers of the executed subtransactions in both the L0 redo log records and the L1 undo log records (Figures 3 and 4). These subtransaction numbers can be viewed as log sequence numbers (LSNs) of the L1 log. Recall the original reason for using LSNs in log-based database recovery: Assume that the L0 log is based on entry logging with entries of the type "shift 100 bytes by 10 bytes to the right" describing non-idempotent page-level operations. Because we do not know which of these operations is reflected in the database after a crash, an additional handshake is needed between the L0 log and the database itself to provide idempotent redo. This additional handshake is usually implemented by storing the highest LSN of a page's L0 update log records in the header of the page (Gray, 1978; Crus, 1984; Mohan et al., 1992). By comparing the LSN of a log record with the LSN of the page, the warmstart procedure can decide whether the log record must be skipped or not.

Implementing the handshake between the L0 log and the L1 log in our case is a bit more difficult, because the order of the subtransactions' after-images in the L0 redo log may be different from the order of the same subtransactions' log records in the L1 undo log. Thus, during the redo pass of the warmstart, it is not sufficient to keep track of the highest subtransaction number (i.e., L1 LSN) that is contained in the L0 log. Rather we must collect a list of "winner subtransactions" that is afterwards used by the undo pass for checking the applicability of the L1 log records.

In the multi-level recovery method without deferred log writes, each after-image in the L0 log belongs to exactly one subtransaction. With deferred log writes, each after-image belongs to one persistence sphere, which may consist of multiple subtransactions. Hence, we actually record a list of subtransaction numbers that is spread across the headers of a persistence sphere's after-images rather than merely a single subtransaction number. If this list becomes unusually long, an additional page with such bookkeeping information is included in the set-oriented I/O that writes the persistence sphere. The header of the last after-image of a persistence sphere contains a flag to mark the end of the persistence sphere so that the writing of the entire persistence sphere is made atomic, and it contains, in a separate header field, the number of the committing transaction that caused the writing of the persistence sphere. The committed transaction numbers are also collected during the redo pass, and are needed by the undo pass to handle the case of non-forced (and thus missing) EOT log records in the L1 log (Section 2.2.2).

Logging During the Warmstart. During the redo phase of a warmstart, no logging is necessary because restoring page after-images is idempotent and can therefore be repeated as often as necessary. During the undo phase, however, the problem arises that a compensating subtransaction is neither atomic nor idempotent. Hence, a crash in the middle of the undo phase makes it impossible to know which compensating subtransactions have been executed and should not be repeated; also, it is possible that some compensating subtransactions were executed only partially. Therefore, L0

Figure 7. Operations during the warmstart

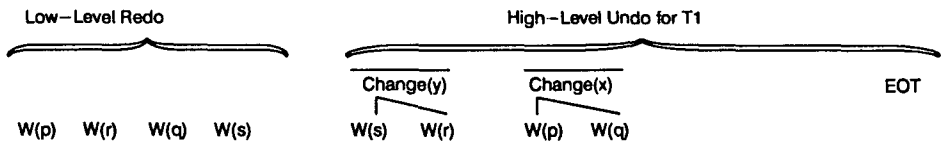
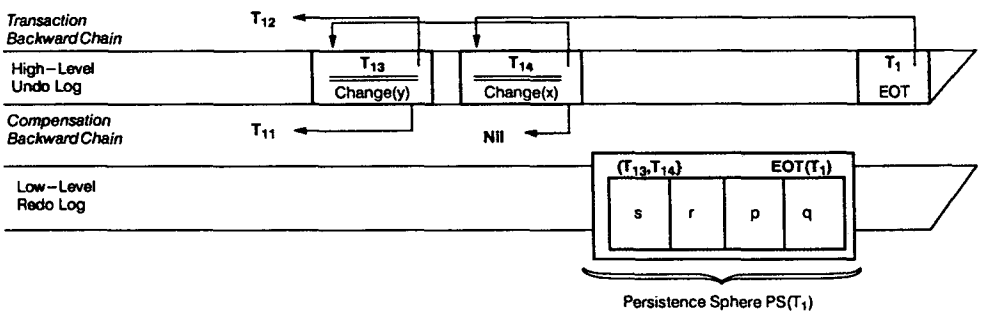


Figure 8. Log records written during the warmstart



redo logging must be in effect again during the undo phase, to ensure the atomicity of the executed compensating subtransactions. To keep track of the progress during the undo phase and to handle repeated warmstarts, L1 undo log records are written for the executed compensating subtransactions.

In our approach, compensating subtransactions and regular subtransactions are treated uniformly for simplicity. Thus, undoing a transaction is actually not distinguishable from performing forward recovery. For each compensating subtransaction, L0 redo log records are written, and an L1 undo log record is written that describes the inverse of the compensating subtransaction (i.e., the inverse of an inverse operation). At the end of undoing a transaction, an EOT log record is written as though the transaction were normally completed.

Figures 7 and 8 show the operations that are executed and the log records that are written during the warmstart for the scenario of Figures 1 and 4. Operations with a double overbar denote the inverses of inverse operations. If the system crashed once more just before the EOT(T1) log record is written (Figure 8), then the following warmstart would redo the subtransactions T11 through T14. Because the after-images of T13 and T14 did not include the EOT flag (because the rollback was not yet complete), our recovery manager would undo both the compensating subtransactions T14 and T13 and the regular subtransactions T12 and T11 by following the "transaction backward chain" of L1 log records.

An optimization of the described implementation would be to apply the technique of Mohan et al. (1992), which avoids undoing an undo operation (i.e., compensating a compensating subtransaction) by following an additional "compensation backward chain" between the L1 log records of a transaction. This technique guarantees that

repeated crashes do not cause increasingly longer warmstarts, and allows resolving top-level deadlocks by partially rolling back a transaction. Note, however, that page-level deadlocks can be handled more easily by rolling back and restarting one or more subtransactions (i.e., by exploiting the nested transaction structure).

Incorporating the optimization of Mohan et al. (1992) in our implementation would be fairly straightforward, accomplished by simply adding the compensation backward chain (Figure 8). The L1 log record of T13 would point to T11 (i.e., the predecessor of the subtransaction that is compensated by T13) and T14 would have a nil pointer because it compensates T11 and T11 is the transaction's first subtransaction. The processing of the L1 undo log would follow this additional compensation backward chain until a log record is encountered that corresponds to a subtransaction that is not among the "winner subtransactions" of the redo phase. As this log record is skipped, we also ignore its compensation backward pointer and follow the regular transaction backward chain.

3. Adding Intra-Transaction Parallelism

Advanced DBMS applications such as engineering or document management have a high potential for parallelism within a single transaction (Duppel et al., 1987; Haerder et al., 1989; DeWitt and Gray, 1992; Haerder et al., 1992). Such intra-transaction parallelism is a key technology for speeding up both retrieval and set-oriented update operations on complex objects. Similarly, applications that update large amounts of derived data and/or check complex integrity constraints can substantially benefit, too (Hudson and King, 1989) (Section 1). We have extended our implementation of multi-level transaction management also to parallel subtransactions of a single transaction. Implementing these extensions has been fairly straightforward. Multi-level transaction management, by its modular nature, is uniformly effective for subtransactions at the page level, regardless of whether two subtransactions belong to different transactions or to the same transaction. Thus, adding intra-transaction parallelism required only one additional component for scheduling the subtransactions within a transaction, and it required changes to the process architecture of DASDBS. In the following two subsections, these modifications are briefly discussed.

3.1 Scheduling of Subtransactions

The newly-implemented scheduling component requires that the programmer of a transaction program specify the precedence orders between the subtransactions of a transaction. Generally, two subtransactions have no precedence order if there is neither a control flow- nor a data flow-dependency between them and if they do not potentially conflict at the object level. However, subtransactions that are "independent" in the above sense are allowed to have potential conflicts at the page level. It is still reasonable to execute such subtransactions in parallel, because a

potential conflict does not necessarily mean that a lock conflict actually will occur. Even if there is a page-level lock conflict between two parallel subtransactions of the same transaction, it may still be beneficial, in terms of response time, to exploit the possible parallelism to the largest possible extent rather to serialize the subtransactions in advance. In the worst case, a page-level deadlock can involve two or more subtransactions of the same transaction. This is recognized by the lock manager and handled in the same way as a page-level deadlock between subtransactions that belong to different transactions. That is, one or more subtransactions are rolled back and (automatically) restarted; it is not necessary to abort the entire transaction. (This advantage would, of course, hold for any other nested transaction model as well; Moss, 1985, Haerder et al., 1992.) If the deadlock could be foreseen before the subtransactions start executing (i.e., if the probability of a deadlock is estimated to be high), the critical subtransactions should be serialized better in advance.

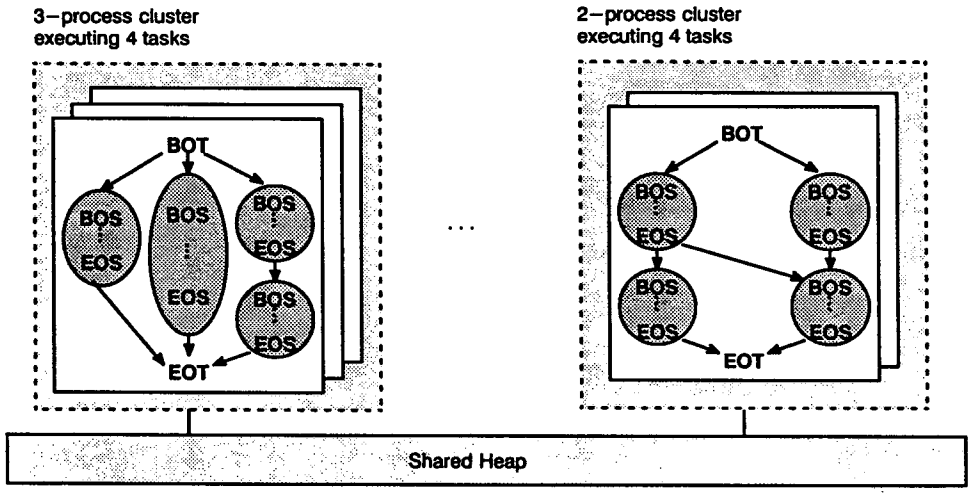
From the specification of the precedence orders between subtransactions, a Petri-net-like precedence graph is constructed. This graph is used for driving the parallel execution of the subtransactions. That is, a subtransaction is invoked by the scheduler when all its predecessors in the precedence graph are successfully completed. It is planned to enhance the scheduler so that it takes into account estimates about the resource consumption and the locking behavior of a subtransaction. The goal is to schedule eligible subtransactions so that the utilization of processors and disks is approximately balanced (cf. Pirahesh et al., 1990; Murphy and Shan, 1991). Furthermore, the scheduling of subtransactions should avoid data-contention bottlenecks and especially deadlocks that can be predicted in advance.

3.2 Process Architecture

DASDBS has a process-per-transaction architecture; that is, each transaction is executed in a separate process, with newly arriving transactions reusing existing processes. All global data structures (e.g., buffer frames, control blocks for buffer management, locking, logging, etc.) are allocated in shared memory. In the original implementation, each process had only one thread of control for sequentially executing all subtransactions of a transaction. This has been extended by spawning a light-weight process for each subtransaction that is to be executed. These light-weight processes are provided by the μSystem parallel programming library (Buhr and Stroobosscher, 1990) that we used in the implementation. Light-weight processes are called μtasks in the μSystem; we will refer to them simply as "tasks." Such tasks are executed within a "cluster" of one or more heavy-weight processes. The processes within a cluster are called "virtual processors" in the μSystem. All processes of all clusters share the same heap, for which the μSystem provides the memory management.

Because we have to deal with both inter- and intra-transaction parallelism, we generate a cluster of processes for each concurrently executing transaction, in accordance with the original process-per-transaction architecture. The new process

Figure 9. New process architecture of DASDBS



architecture is illustrated in Figure 9. The number of processes in a cluster is dynamically adjusted so that it is always equal to the number of simultaneously active tasks (i.e., parallel subtransactions within a transaction). Because of the high costs of process creation and destruction associated with this dynamic mechanism, we also support an alternative in which the number of processes in a cluster is set to the maximum number of tasks that can be simultaneously active. This number of processes is set already at the beginning of a transaction, and all processes are kept until the transaction completes.

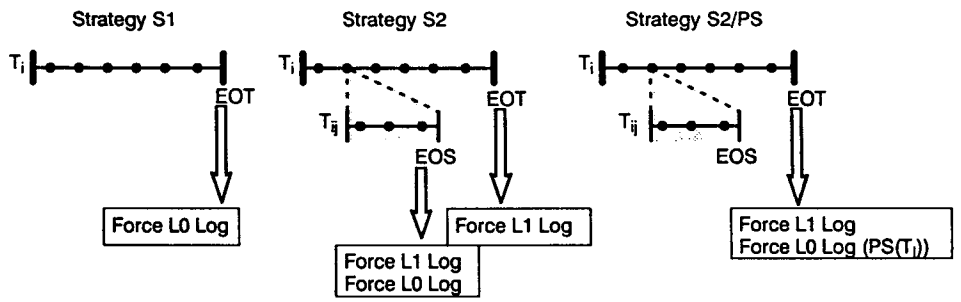
A major point of the described process architecture is that it also allows using fewer processes in a cluster than there are concurrently executing tasks. This option, which is provided by the μSystem for each cluster individually, is useful if some of the tasks are I/O-intensive so that the number of tasks that require processors is less than the number of executing tasks. In addition, the combination of inter- and intra-transaction parallelism may require limiting the total number of processes in the process clusters of the concurrently executing transactions. This sort of load control or throttling is essential for avoiding excessive context-switching (of heavy-weight processes) as well as other thrashing-like situations like excessive memory contention and data contention. The goal that we are pursuing in the long run is to adjust the number of processes in the clusters of the transactions to the current load situation dynamically and automatically.

4. Performance Evaluation

4.1 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

Figure 10. Log I/O costs of different recovery strategies



compared the following three strategies, all of which are implemented in DASDBS:

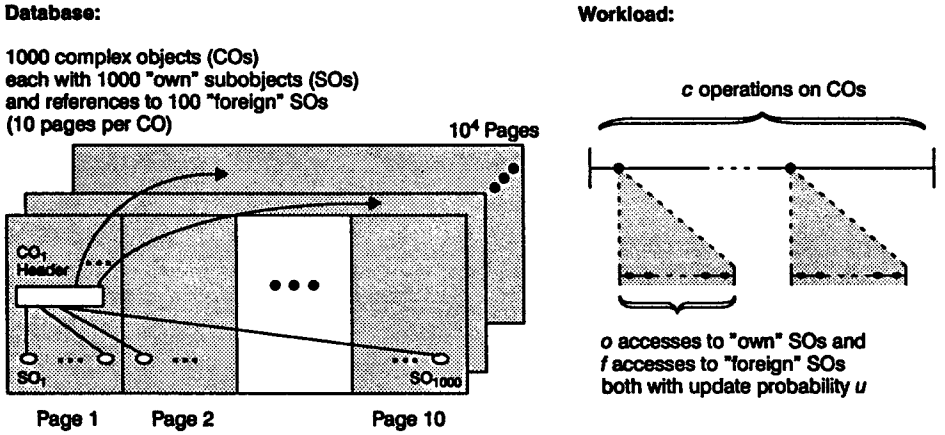
- *strategy S1,* page-oriented single-level transaction management, using strict two-phase locking on pages and the DB Cache method for recovery,
- *strategy S2,* two-level transaction management with log writes at each EOS, and
- *strategy S2/PS,* two-level transaction management with deferred log writes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

Because the logging overhead is 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 10.

Our performance evaluation is based on a synthetic benchmark which follows some ideas proposed in the complex-object benchmarks of Anderson et al. (1990) and DeWitt et al. (1990). The benchmark has the following characteristics, as illustrated in Figure 11.

- Our test database consists of 1,000 complex objects (COs) each of which consists of 1,000 "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 an 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 and 50-50 rules were implemented by applying a linear transformation to a normal distribution of random numbers. The 1,000 "own" SOs of a CO constitute a storage cluster of 10 contiguous pages, with a page 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 10,000 pages or 20 MBytes.

Figure 11. Database and workload of the performance experiments



- 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 Haerder (1987), 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 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). So our experimental setup models a closed queueing system (i.e., arrival rate equals throughput). In the experiments, the DMP was systematically varied for different runs.

4.2 Performance 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 (more than two times higher throughput) and 2.4 (more than two times shorter response time). Figures 12a and 12b 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. Detailed figures for this case are given in Figure 12f.

- *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 (Figure 12f, DMP 12). The observed conflict rate of 1.6% for strategy S1 at DMP 12 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% 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 12c shows the total lock wait time of all three strategies as a function of the DMP.

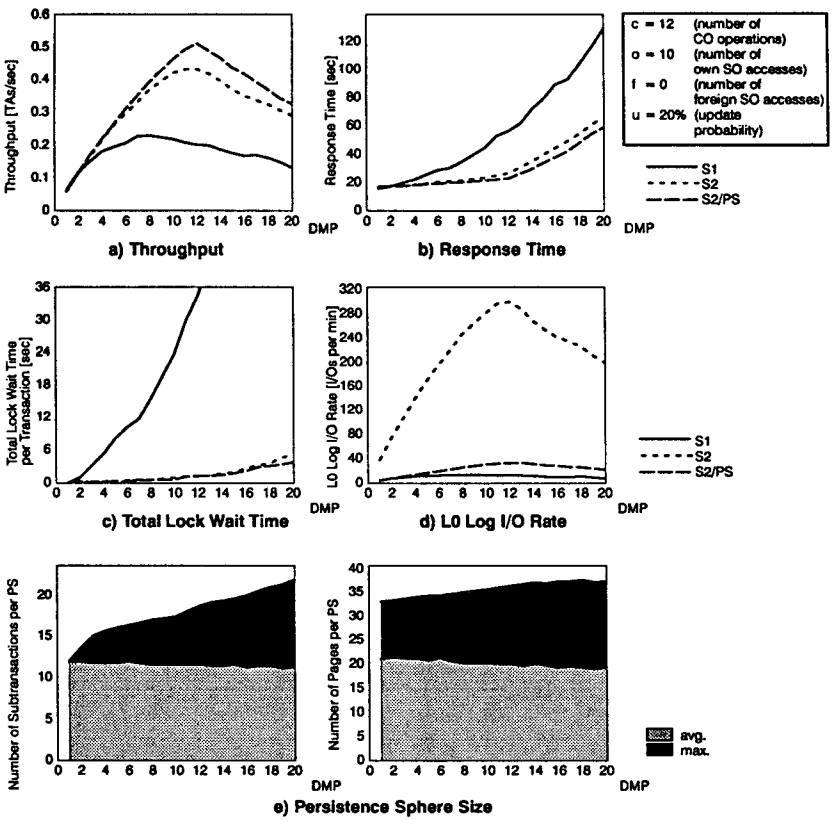
- *Log I/Os:* Because the simple two-level strategy S2 performed log I/Os for each update subtransaction, its log I/O rate was dramatically higher than that of strategy S1 (Figure 12d). This disadvantage of S2 was almost completely eliminated by strategy S2/PS. For example, at a DMP of 12, strategy S2/PS 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, which was in contrast to our expectations. In fact, the costs of log I/Os was our main concern in the design of the deferred logging approach of Section 2.2.2. However, 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 would 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 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 12e shows, this effect increases with the DMP. Note, 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 12a and 12b 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 (Graefe and Thakkar, 1992). 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). Because we implemented latches by spin locks (Graunke and Thakkar, 1990), latch contention actually led to wasted CPU cycles; and because the CPU utilization was almost 100% at DMP 12, increasing the DMP beyond 12 caused a significant decrease of performance.

Figure 12. Results of baseline experiment with disjoint complex objects



	TPUT [TAs per sec]	RT [sec]	#Lock Requests per min.		#Lock Waits per min.		Lock Conflict Probability [%]		#Deadlocks per min.		#Log I/Os per min.		#Pages per Log I/O	
			L0	L1	L0	L1	L0	L1	L0	L1	L0	L1	L0	L1
S1	0.20	56.1	2001	--	32.4	--	1.6	--	6.2	--	12.3	--	22.3	--
S2	0.44	26.6	4044	3372	7.1	3.4	.17	.10	2.2	0	299.6	334	2.04	0.71
S2/PS	0.51	23.0	4650	3884	8.3	4.8	.17	.12	2.3	0	33.2	33.6	19.9	3.66

f) Performance Comparison at DMP 12

- *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 13. These experiments essentially confirmed the observations discussed above. In interpreting the slope of the curves, 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.

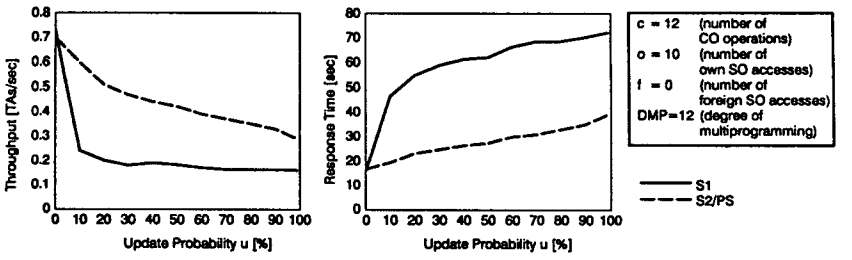
4.3 Complex Objects with Referentially Shared Subobjects.

In this section, we discuss the performance results for the case with accesses to foreign subobjects. We first discuss the performance when all subobjects accessed by a complex-object operation are foreign subobjects (i.e., subobjects 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 4.2. Strategy S2 is no longer considered here because it was always outperformed by S2/PS.

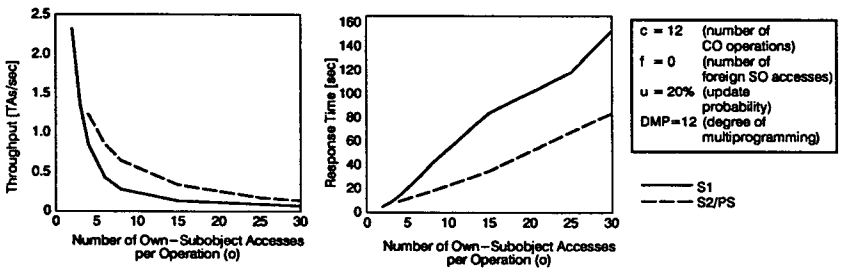
- *Overall performance and lock conflicts:* As shown in Figure 14, the performance difference of S1 and S2/PS became even bigger, compared to the case without foreign-subobject accesses. For example, at a DMP of 12, S2/PS achieved 16 times higher throughput and 10 times shorter response time than S1. This performance difference is mostly caused by data contention (Figures 14c and 14f). For strategy S1, both the total lock wait time and the conflict rate were substantially higher than in the experiment of Section 4.2. 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 4.2, 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, the first SO access within each complex-object operation (Section 4.2) 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) accesses. (The net effect is similar to preclaiming, even though no preclaiming is actually performed.) Destroying this locality led to the disastrous performance of strategy S1.

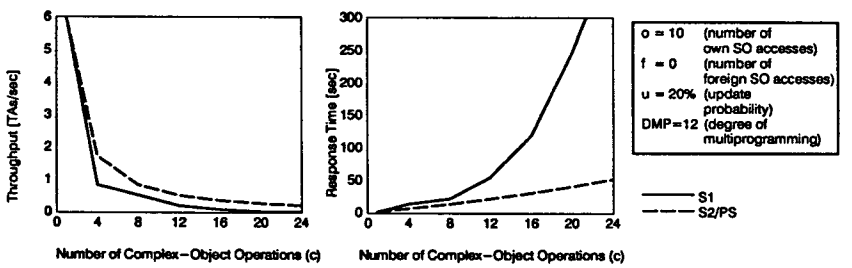
Figure 13. Sensitivity of baseline parameters with disjoint complex objects



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

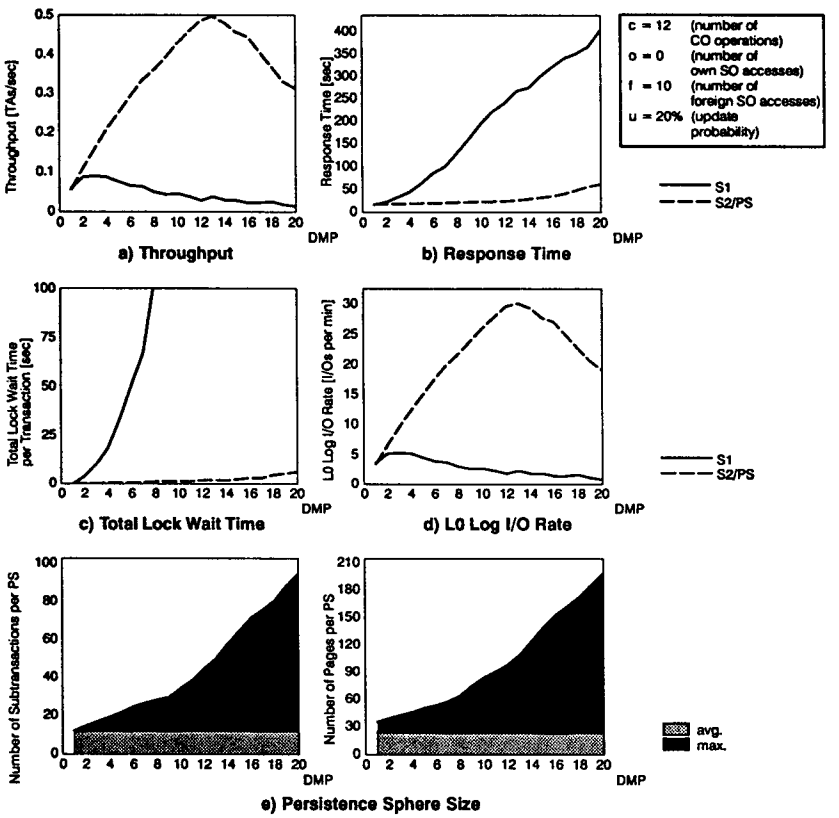


b) Throughput and Response Time with Varying Number of Own–Subobject Accesses (o)



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

Figure 14. Results of the experiment with foreign-subobject accesses (f = 10)



	TPUT (TAs per sec)	RT [sec]	#Lock Requests per min.		#Lock Waits per min.		Lock Conflict Probability [%]		#Deadlocks per min.		#Log I/Os per min.		#Pages per Log I/O	
			L0	L1	L0	L1	L0	L1	L0	L1	L0	L1	L0	L1
S1	.03	258	1176	--	42.5	--	3.6	--	14.6	--	1.73	--	23.7	--
S2/PS	.48	24	4279	3568	24.8	4.2	0.5	0.1	0.4	0	29.6	30.0	23.0	2.4

f) Performance Comparison at DMP 12

- *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 (Figure 14e). 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. This effect can be quite beneficial (Section 4.2), for it amounts to more batching of log I/Os (i.e., fewer 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 would cause (Gray et al., 1981).

  There are two ways to eliminate or alleviate the described effect (none of which is currently implemented in DASDBS). The first 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 mechanism may actually increase the total amount of work because it may write more pages, but 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 multi-disk log (cf. Seltzer and Stonebraker, 1990), 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 op-

eration are shown in Figure 15. These results essentially confirm the above observations. That is, with an increasing number of foreign-subobject accesses, transactions loose locality which leads to more conflicts with S1 and potentially larger persistence spheres with S2/PS.

4.4 CPU Overhead

In this section, we discuss the additional CPU costs that are incurred by our multi-level recovery algorithm. For this purpose, we reran a number of experiments using the UNIX profiling tool gprof. We restricted ourselves to the case of disjoint complex objects, i.e., the workload parameter setting of the base experiment of Section 4.2: $c = 12$ complex-object operations per transaction, $o = 10$ own-subobject accesses per complex-object operation, no foreign-subobject access ($f = 0$), and update probability $u = 20\%$. Figure 16 shows the CPU time per transaction for the DMP values 1 (i.e., single-user mode), 4, 8, and 12, comparing the strategies S1 and S2/PS. The total CPU time is broken down into the following components:

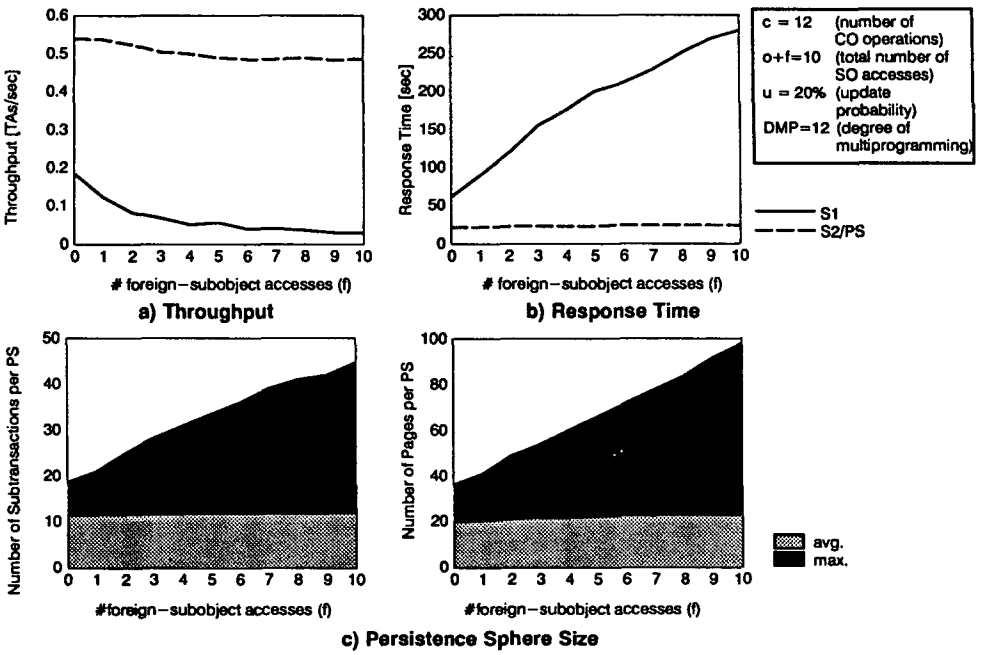
- *OM:* object management, which includes the management of complex records and object buffers and the query processing (see Schek et al., 1990, for these components of DASDBS),
- *PM:* page management, which includes the buffer manager, free place administration, and I/O services,
- *TM-1:* the object-level transaction management, which includes the L1 lock and log management and the transaction bookkeeping, and
- *TM-0:* the page-level transaction management, which includes the L0 lock and log management, the management of persistence spheres, and the subtransaction bookkeeping.

The breakdown of the CPU costs is shown in Figure 16. The total figures show that the two-level transaction management incurs an overhead of up to about 14%. This overhead is mostly caused by the object-level locking and logging. Note, however, that our experiments are based on a university prototype which has a large potential for code fine-tuning. In addition to the overhead at level L1, there is also a noticeable overhead at the page level L0. The total increase of CPU time in the page management and the page-level transaction management, for S2/PS versus S1, is almost 50%, but note that the absolute page-level CPU time of S2/PS constitutes less than 10% of a transaction's total CPU time.

The page-level CPU overhead of S2/PS can be attributed to the following factors, ordered by descending fraction of costs:

- releasing and re-requesting page locks within a transaction (included in TM-0), which is by far the largest factor within TM-0,

Figure 15. Sensitivity of number of foreign-subobject accesses (f)



- additional page copying and before-image management in cases where the same page is modified by multiple subtransactions of the same transaction (included in PM, since this is integrated into the buffer manager),
- wasted CPU cycles due to latch waits (included in both TM-0 and PM), and
- bookkeeping for subtransactions and persistence spheres (included in TM-0).

Note that the strategy S1 suffered substantially fewer latch waits (not explicitly shown in Figure 16), since the data-contention bottleneck for this strategy led to a large fraction of blocked transactions which in turn reduced the contention for latches.

We also measured the CPU costs for other experiments, including a scenario in which data contention was not a performance-limiting factor. These measurements confirmed that the CPU overhead of our multi-level method is acceptable. In all cases, the overhead of S2/PS was on the order of 10%, mostly due to the logging and locking at the object level L1. This is a modest price for the benefit of increased concurrency whenever data contention is of concern. Note that similar costs would inevitably arise with every kind of object-level concurrency control and recovery.

Figure 16. CPU costs of S1 and S2/PS

	DMP 1		DMP 4		DMP 8		DMP 12	
	S1	S2/PS	S1	S2/PS	S1	S2/PS	S1	S2/PS
OM	10.65	10.65	10.65	10.65	10.65	10.65	10.65	10.65
PM	0.67	0.76	0.70	0.76	0.74	0.83	0.68	0.84
TM−1	−−	0.89	−−	0.93	−−	1.01	−−	1.19
TM−0	0.03	0.06	0.03	0.09	0.03	0.16	0.03	0.25
Total	11.35	12.36	11.38	12.43	11.42	12.65	11.36	12.93

Costs in CPU seconds for the baseline experiment with disjoint complex objects.

4.5 Preliminary Performance Results for Intra-Transaction Parallelism

In a final series of experiments, we studied the impact of intra-transaction parallelism on multi-level transaction management. We concentrated on evaluating the strategy S2/PS because it always outperformed S2. Note that intra-transaction parallelism requires some form of subtransactions and is therefore not feasible with strategy S1 as it was implemented. In our benchmark, we assumed that all subtransactions of a transaction can indeed be executed in parallel (i.e., there is no precedence order between the complex-object operations of a transaction). In the experiments, the effective degree of intra-transaction parallelism (DIP) was varied between 1 and 6. For example, with a DIP of 6, the first through sixth subtransaction of a transaction were executed in parallel, and subsequently the seventh through twelfth subtransaction were in parallel. We varied the DMP orthogonally to the DIP, to investigate how inter- and intra-transaction parallelism affect each other. Some preliminary results are discussed in the following.

- *Overall performance:* Figures 17 and 18 show the performance results with and without foreign-subobject accesses, respectively. In the following, we concentrate on discussing the more interesting case with foreign-subobject accesses. The performance impact of the DIP was highly dependent on the DMP. With a low DMP, a relatively high DIP reduces the transaction response time and improves throughput; with a high DMP, however, the potential benefits of intra-transaction parallelism clearly are outweighed by the additional costs. The main bottleneck was the CPU capacity, as we had only 12 processors available but generated DMP * DIP processes with a CPU-intensive workload.

- *Lock conflicts and latch conflicts:* As Figure 18c shows, the contention for locks, especially page locks at level L0, increased drastically with increasing

values of the product DMP * DIP. For example, at DMP 12 and DIP 4, about 25% of a transaction's response time was spent waiting for a lock. This observation is remarkable as the same workload under the same strategy S2/PS showed almost no data contention in the previous experiments without intra-transaction parallelism even at a high DMP (Figure 14). The phenomenon has two explanations. First, the execution time of a transaction increases considerably with the product DMP * DIP, and therefore the potential for data contention increases. Second, intra-transaction parallelism increases the number of concurrently active subtransactions and hence the data contention at level L0. Thus, if the product DMP * DIP is not properly controlled, then short-term page locks become a performance-critical factor even though they are released at EOS.

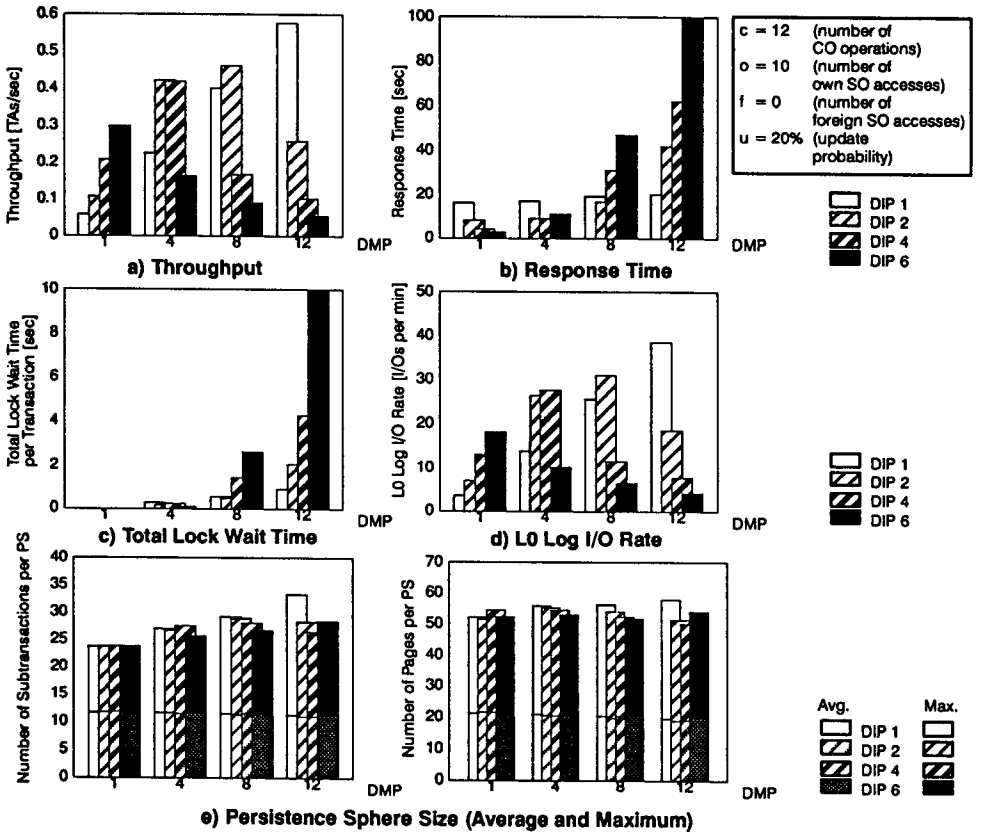
Finally, the contention for internal latches became a severe performance problem at high values of DMP * DIP (Section 4.1). Even though this problem could be alleviated by tuning the code within the critical sections (which may include redesigning some of the buffer manager's and the lock manager's internal data structures), it cannot be completely eliminated if the number of concurrently active subtransactions is unrestricted.

These problems clearly show the need for load control for inter- and intra-transaction parallelism. We are pursuing an approach that dynamically adjusts the DMP and the DIP of the admitted transactions to the current load in terms of lock and latch contention as well as resource contention (cf. Carey et al., 1990; Moenkeberg and Weikum, 1991, 1992; Thomasian, in press).

- *Log I/Os:* As far as the log I/O rate is concerned, the results with intra-transaction parallelism were no different from the results of Sections 4.2 and 4.3. That is, the number of log I/Os per time interval (Figure 18d) was approximately proportional to the achieved transaction throughput. We observed an interesting effect concerning the maximum size of persistence spheres at different DMP and DIP values. Persistence spheres became larger with increasing DMP for all DIP values (Figure 18e). The gradient of this increase, however, was smaller for high DIP values than for low ones. This may indicate that intra-transaction parallelism is beneficial for keeping persistence spheres small and thus making the execution time of the commit processing more predictable.

  As an explanation of this phenomenon we offer the following hypothesis: The probability that two persistence spheres are merged increases with the product of the number of concurrently active subtransactions (i.e., DMP * DIP) and the average time between a subtransaction's EOS and the EOT of its transaction, or actually, with the integral of the number of completed but not yet forced subtransactions over time. The reason for this relationship is that a subtransaction is eligible for joining a persistence sphere only after its

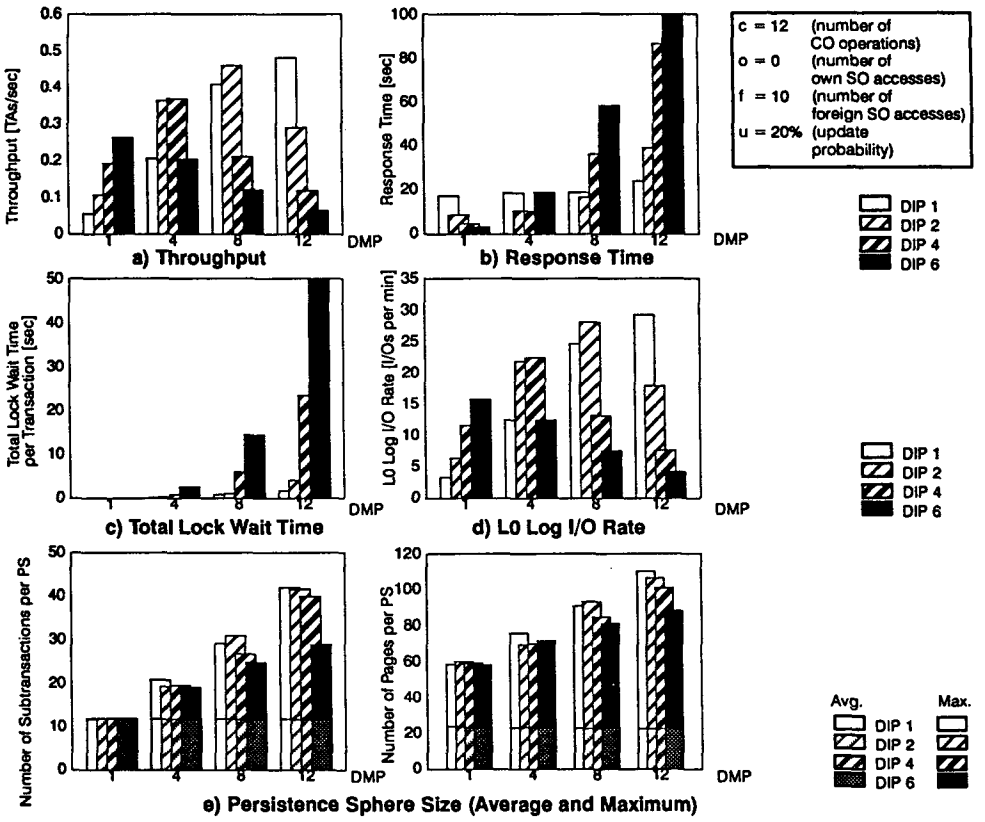
Figure 17. Inter- and intra-transaction parallelism for disjoint complex objects



EOS, and is forced to the log file at EOT at the latest.

When we compare, for example, DMP = 12 and DIP = 1 with DMP = 2 and DIP = 6, the average time for which a subtransaction may join another transaction's persistence sphere can be estimated in a simplified way as follows. In the first case, the first subtransaction of a transaction consisting of 12 subtransactions stays for $\frac{11}{12}$ of the transaction's response time (RT_1) between EOS and EOT, the second subtransaction for $\frac{10}{12}$ of RT1, and so on. This calculation yields an average of $((\frac{11}{12} + ... + \frac{1}{12})$ * $RT_1)/12 = \frac{11}{24}$ RT_1 for the time interval during which a subtransaction may join a persistence sphere. In the second case, the first through sixth subtransaction of a transaction stay for $\frac{1}{2}$ of the transaction's response time (RT_2) in the state between EOS and EOT; the seventh through twelfth subtransaction spend virtually no time between EOS and EOT if we assume ideal scheduling. This yields

Figure 18. Inter- and intra-transaction parallelism with foreign-subobject accesses



an average of $((6 * \frac{1}{2}) * RT_2)/12 = \frac{RT_2}{4}$ for the critical time interval. Of course, this strawman calculation disregards lock wait time and scheduling effects. Nevertheless, we believe that it can be considered as an argument that intra-transaction parallelism is indeed beneficial for keeping (the variance of) persistence spheres small.

5. Comparison with Related Work

Multi-level transaction management methods are implemented in the commercial database system SQL/DS, which is essentially System R (Gray et al., 1981), Synapse (Ong, 1984), and Informix-Turbo (Curtis, 1988). 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 (von Bueltzingsloewen et al., 1988). 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 (Iochpe, 1989). However, it is not 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 (Mohan and Pirahesh, 1991; Mohan et al., 1992; Mohan and Levine, 1992). Even though the two methods were independently developed with very different design objectives, they have quite a few properties in common, as discussed in the following.

1. Both methods perform redo at the page level (i.e., "physical redo" in the terms of Mohan et al., 1992), thus minimizing the redo costs during a warmstart.

2. 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.

3. As an unavoidable consequence of properties (1) and (2), 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 (Mohan et al., 1992).

4. To keep track of the modifications 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; Mohan et al., 1992).

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 (Mohan et al., 1992), 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, the fourth point would be actually 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, however, evolved from a theoretically well-founded but relatively puristic framework, aiming at high modularity and generality that would allow handling arbitrarily complex high-level operations.

A difference between ARIES and our method is the amount of redo processing during a warmstart. The difference has only minor practical relevance, but it provides insight into the different behavior of the two methods. Persistence spheres, as used in our method, are the minimal sets of redo log records that must be written 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 redoes all updates up to the point of the crash. The enhanced version called ARIES/RRH (Mohan and Pirahesh, 1991) avoids some of this redo work by determining, 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; Mohan and Pirahesh, 1991) 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 was followed by a winner transaction that modified the critical page or if the dirty page was written back into the database before the crash occured.

Another closely related recovery method is the MLR method by Lomet (1992). This method aims to combine the industrial-strength properties of ARIES with the modular structure of our multi-level approach. MLR essentially takes the original multi-level recovery method of Weikum (1991) as a conceptual starting point, and then adds a number of optimizations. In particular, MLR uses entry logging, merges the high-level undo log and the low-level log into a single log file, and is able to combine the writes of a high-level undo log record and multiple redo log records into a single atomic event to ensure the atomicity of subtransactions. These optimizations are similar to some of the ARIES features. In fact, ARIES would ensure the atomicity of multi-page updates also by writing several log records to a single log file in an atomic manner (even though this is not explicitly discussed in the ARIES papers). Several similar techniques have been used in Tandem's commercial database systems (Gray and Reuter, 1993), but were not published in the academic community.

6. Further Performance Improvements

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 are briefly discussed in the following.

- *"Light Weight" Subtransactions:* For particular types of high-level operations, the resulting reference pattern at the page level may have specific properties (e.g., pages are accessed in a specific order), so that it may be possible to guarantee deadlock-freedom between the corresponding subtransactions. Because the conflict rate at the page level is usually low, we may also simplify the queue management at virtually no risk of starvation. Under these conditions, it would be feasible to implement the page-level concurrency control between the eligible subtransaction types by latches rather than full-fledged locks. This would substantially reduce the CPU costs of multi-level concurrency control. Of course, the ultimate goal of such an approach would be to automatically generate the necessary latching protocol, based on the analysis of the possible page reference patterns of the particular types of subtransactions.

- *Multi-Granularity Locking:* Another approach to reducing the CPU costs of multi-level concurrency control on complex objects is to incorporate multi-granularity locking at the object level. Unfortunately, while this is relatively simple and actually implemented in our system for the case of disjoint complex objects, it seems that the case of complex objects with referentially shared subobjects has not yet been completely solved (Garza and Kim, 1988; Herrmann et al., 1990; Haerder et al., 1992).

- *Organization of Log Buffers and Log Files:* For subtransactions for which the L0 log write can be deferred until EOT, it is not necessary to write the L1 undo log records before the L0 after-images because the L0 write is atomic. Thus, the transaction's L1 log records could actually be discarded from the L1 log buffer after the successful L0 log write I/O. This would save log I/Os at the expense of having to change the L1 log buffer organization from a sequential ring buffer to a heap-like organization with direct addressing of log records. Note that the selective writing of after-images, which minimizes the amount of L0 redo log records (Section 5), is also based on the fact that after-images are kept in the regular page buffer pool with directly addressable buffer frames rather than in a separate sequentially organized log buffer. An orthogonal way of further reducing the amount of log I/Os could be to write the L1 undo log records also into the L0 log file, i.e., to combine the two logs into a single physical file. It seems that this could be done without major changes to the organization of the L0 log file so that the efficient log compaction technique of Elhardt and Bayer (1984) would still be applicable. The merging of the two logs would result in less (but slightly larger) set-oriented I/Os.

- *Log File Partitioning:* Even though our measurements did not show a log I/O bottleneck, the dramatically increasing speed of CPUs (due to RISC

processors and/or multiprocessor systems) may eventually lead to a situation in which the transaction throughput is limited by the bandwidth of the (L1 or L0 or combined) log disk. Such a bottleneck could only be eliminated by partitioning the log file(s) and distributing the partitions across multiple disks. This could be done transparently to the DBMS, by using RAIDs as a high-speed log device (cf. Seltzer and Stonebraker, 1990), or by explicitly dealing with multiple log partitions. The latter approach has the potential advantage that, during a warmstart, the partitions of the log could be processed in parallel and independently (King et al., 1991). Unlike previous approaches to parallel logging (Agrawal, 1985), our method can indeed achieve this advantage by partitioning the L1 log by transaction numbers and the L0 log by subtransaction numbers or page numbers.

Partitioning the L0 log by subtransaction numbers is only feasible with after-image logging. In this case, we can use timestamps in the headers of after-images and apply the Thomas write rule (Bernstein et al., 1987) to ensure that an after-image will not overwrite a more recent after-image of the same page during the parallel processing of the L0 log partitions.

Partitioning the L0 log by page numbers, on the other hand, leads to the problem that the after-images of a persistence sphere may be distributed across multiple partitions yet have to be written atomically. Rather than employing a full-fledged two-phase commit for this case, a cheaper solution could be to include in the header of each after-image an identification and the cardinality of the persistence sphere to which the after-image belongs. Then, during the parallel redo phase, a determination can be made whether a persistence sphere is complete or if the distributed log I/O failed on one of the partitions.

7. Conclusions

7.1 Major Lessons

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

- It allows the exploitation of the semantics of high-level operations to enhance concurrency.
- Our algorithms can direct complex high-level operations on arbitrarily complex objects. In particular, our method 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 also supports parallelism within a transaction.

The presented performance evaluation basically confirmed the expected benefits in terms of high concurrency and low log I/O costs. In addition, we obtained the following, more specific insights into the impact of multi-level transaction management on various performance factors.

- *Log I/O:* For the class of complex-object databases that we modeled and with computing resources that are comparable to our benchmark platform, log I/O is not a bottleneck in multi-level transaction management. We believe that this observation holds for a fairly large spectrum of object-oriented database workloads. This situation may change with dramatically increasing CPU speed and only gradually improving disk performance. However, the presented deferred logging approach aims to minimize log I/O costs and is scalable in the sense that log files can be distributed across multiple disks and processed largely independently (Section 6). Therefore, it is unlikely that log I/O will cause performance problems in the near future. Note that this observation holds for both transaction throughput and response time. The additional latency that is incurred by the writing of persistence spheres seems to have a minor impact on response time.

- *Data contention:* Data contention is likely to cause performance problems in complex-object applications. Thus, some form of multi-level concurrency control that is able to deal with semantic high-level operations and fine-grained data access is absolutely necessary. Because of the complex nature of such high-level operations, the lower-level concurrency control cannot be implemented by simple page latching, especially if high-level operations could be user-defined and have unpredictable page access patterns (as would be the case in an extensible database system). The inevitable consequence is that the CPU costs of multi-level locking and logging are higher than, for example, the costs of record locking in a relational database system (see below). Another consequence that should be recalled is that conventional page-level logging and recovery methods do not work correctly in combination with concurrency control methods beyond page locking.

- *CPU costs:* The additional CPU costs of the multi-level transaction management are fairly low, but are nevertheless noticeable. For the complex-object workload of our performance evaluation, the CPU costs of a transaction were increased by up to 14% under multi-level transaction management. To a large extent, this reflects a lack of code fine-tuning of our prototype. However, even with improved code, the CPU costs of the multi-level transaction

management method are sensitive to the number of subtransactions into which a transaction is decomposed. Particularly in situations where several subsequent subtransactions access essentially the same set of pages, the CPU costs of releasing and re-requesting page locks are a noticeable factor.

A general recommendation, based on these findings, could be the following. Multi-level transaction management is well-suited for complex-object applications with relatively long transactions and potential data-contention problems. For applications with very short transactions, the gain in concurrency may not be worth the additional CPU overhead, so that simple page-level transaction management or conventional record-level transaction management (without support for multi-page object accesses) could be a better choice. Note, however, that this would not allow exploiting the semantics of high-level operations that access more than one page. Finally, for applications with virtually no data-contention problems, simple page locking and logging are, of course, sufficient.

7.2 Future Work

For applications that do not have data-contention problems, multi-level transaction management incurs unnecessary overhead. For such applications, standard page locking and logging are sufficient, at lower CPU costs and reduced code complexity. However, many applications may face occasional data-contention problems (e.g., due to load peaks or under a specific mix of transactions). In this case, it would be desirable to switch dynamically from page-oriented single-level transaction management to multi-level transaction management. This idea is similar to the de-escalation technique that is used to switch from coarse-grained (e.g., page) locking to fine-grained (e.g., record) locking (Joshi, 1991). Unfortunately, semantic locking for arbitrary high-level operations cannot be easily incorporated into the de-escalation approach or other forms of multigranularity locking (Muth et al., 1993).

The implemented prototype system will serve as a testbed for further studies, especially on the tuning problems that arise with the coexistence of inter- and intra-transaction parallelism. This coexistence leads to more contention for resources (i.e., processors, memory, I/O bandwidth, locks, latches), compared to a conventional database system with inter-transaction parallelism alone. Therefore, the decision on how much intra-transaction parallelism should be exploited in an individual transaction is dependent on the overall system load. Our long-term goal is to develop load control (i.e., transaction and subtransaction admission) and scheduling strategies that adjust the degree of inter-transaction parallelism and the degrees of intra-transaction parallelism of the individual transactions to the current load dynamically and automatically.

This and further tuning problems are being addressed as part of the COMFORT project at ETH Zurich (Weikum et al., 1993). The ultimate goal of COMFORT is to automate tuning decisions for transaction processing in parallel database systems, thus simplifying the tricky job of system administrators and human tuning experts.

Acknowledgments

The method for deferred log writes and the concept of persistence spheres were designed jointly with Peter Broessler and Peter Muth. Their contribution is gratefully acknowledged. We are grateful to Arnie Rosenthal for helpful discussions on the definition of persistence spheres, and we would like to thank the anonymous referees for their very constructive comments. Finally, we would like to thank the UBILAB of the Union Bank of Switzerland (Schweizerische Bankgesellschaft) and especially Rudolf Marty for supporting our work.

References

Agrawal, R. A Parallel logging algorithm for multiprocessor database machines. *Fourth International Workshop on Database Machines,* Grand Bahama Island, 1985.

Anderson, T.L., Berre, A.H., Mallison, M., Porter, H., and Schneider, B. The hypermodel benchmark. *Second International Conference on Extending Data Base Technology,* Venice, 1990.

Badrinath, B.R. and Ramamritham, K. Performance evaluation of semantics-based multilevel concurrency control protocols. *ACM SIGMOD International Conference on the Management of Data,* Atlantic City, 1990.

Beeri, C., Schek, H.-J., and Weikum, G. Multi-level transaction management, theoretical art or practical need? *First International Conference on Extending Database Technology,* Venice, 1988.

Beeri, C., Bernstein, P.A., and Goodman, N. A model for concurrency in nested transactions systems. *Journal of the ACM,* 36(1):230-269, 1989.

Bernstein, P.A., Hadzilacos, V., and Goodman, N. *Concurrency Control and Recovery in Database Systems,* Reading, Massachusetts: Addison-Wesley, 1987.

Broessler, P. and Freisleben, B. Transactions on persistent objects. *International Workshop on Persistent Object Systems,* Newcastle, Australia, 1989.

Buhr, P.A. and Stroobosscher, R.A. The μSystem: Providing light-weight concurrency on shared-memory multiprocessor computers running UNIX. *Software–Practice and Experience,* 20(9):929-964, 1990.

Carey, M.J., Krishnamurthi, S., and Livny, M. Load control for locking: The "half-and-half" approach. *ACM International Symposium on Principles of Database Systems,* Nashville, Tennessee, 1990.

Cart, M. and Ferrie, J. Integrating concurrency control into an object-oriented database system. *Second International Conference on Extending Database Technology,* Venice, 1990.

Copeland, G. and Keller, T. A comparison of high-availability media recovery techniques. *ACM SIGMOD International Conference on the Management of Data,* Portland, 1989.

Crus, R.A. Data recovery in IBM database 2. *IBM Systems Journal,* 23(2):178-188, 1984.

Curtis, R.B. Informix-Turbo. *IEEE COMPCON,* San Francisco, 1988.

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

DeWitt, D.J. and Gray, J. Parallel database systems: The future of high performance database systems. *Communications of the ACM,* 35(6):85-98, 1992.

Duppel, N., Peinl, P., Reuter, A., Schiele, G., and Zeller, H. *Progress report #2 of PROSPECT.* Department of Computer Science, University of Stuttgart, 1987.

Elhardt, K. and Bayer, R. A database cache for high performance and fast restart in database systems. *ACM Transactions on Database Systems,* 9(4):503-525, 1984.

Fekete, A., Lynch, N., Merritt, M., and Weihl, W. Commutativity-based locking for nested transactions, Technical Report MIT/LCS/TM-370, MIT, Cambridge, Mass., 1988.

Garcia-Molina, H. Using semantic knowledge for transaction processing in a distributed database. *ACM Transactions on Database Systems,* 8(2):186-213, 1983.

Garcia-Molina, H. and Salem, K. Sagas. *ACM SIGMOD International Conference on the Management of Data,* San Francisco, 1987.

Garza, J. and Kim, W. Transaction management in an object-oriented database system. *ACM SIGMOD International Conference on the Management of Data,* Chicago, 1988.

Gawlick, D. and Kinkade, D. Varieties of concurrency control in IMS/VS fast path. *IEEE Database Engineering,* 8(2):3-10, 1985.

Gibson, G.A. *Redundant Disk Arrays: Reliable, Parallel Secondary Storage,* New York: ACM Press, 1992.

Graefe, G. and Thakkar, S.S. Tuning a parallel database system on a shared-memory multiprocessor. *Software: Practice and Experience,* 22(7):495 ff, 1992.

Graunke, G. and Thakkar, S. Synchronization algorithms for shared-memory multiprocessors. *IEEE Computer,* 23(6):60-69, 1990.

Gray, J. Notes on Database Operating Systems. In: Bayer, R., Graham, R., and Seegmueller, G., eds., *Operating Systems: An Advance Course,* Berlin: Springer, 1978.

Gray, J., McJones, P., Blasgen, M., Lindsay, B., Lorie, R., Price, T., Putzolu, F., and Traiger, I. The recovery manager of the system R database manager. *ACM Computing Surveys,* 13(2):223-242, 1981.

Gray, J. and Reuter, A. *Transaction Processing: Concepts and Techniques,* City?: Morgan Kaufmann, 1993.

Hadzilacos, T. and Hadzilacos, V. Transaction synchronization in object bases. *ACM International Symposium on Principles of Database Systems,* Austin, Texas, 1988.

Haerder, T. and Reuter, A. Principles of transaction-oriented database recovery. *ACM Computing Surveys,* 15(4):287-317, 1983.

Haerder, T. On selected performance issues of database systems. *Fourth German Conference on Performance Modeling of Computing Systems,* Erlangen, Germany, 1987.

Haerder, T., Schoening, H., and Sikeler, A. Parallel query evaluation: A new approach to complex object processing. *IEEE Data Engineering,* 12(1):23-29, 1989.

Haerder, T., Profit, M., and Schoening, H. Supporting parallelism in engineering databases by nested transactions. Technical Report 34/92, SFB 124, Department of Computer Science, University of Kaiserslautern, Germany, 1992.

Hasse, C. and Weikum, G. A performance evaluation of multi-level transaction management. *International Conference on Very Large Data Bases,* Barcelona, Spain, 1991.

Helland, P., Sammer, H., Lyon, J., Carr, R., Garrett, P., and Reuter, A. Group commit timers and high volume transaction systems. *Second International Workshop on High Performance Transaction Systems,* Pacific Grove, California, 1987.

Herrmann, U., Dadam, P., Kuespert, K., Roman, E.A., and Schlageter, G. A lock technique for disjoint and non-disjoint complex objects. *Second International Conference on Extending Database Technology,* Venice, Italy, 1990.

Hudson, S.E. and King, R. Cactis: A self-adaptive, concurrent implementation of an object-oriented database management system. *ACM Transactions on Database Systems,* 14(3):291-321, 1989.

Iochpe, C. Database recovery in the design environment: Requirements analysis and performance evaluation. Ph.D. Thesis, Department of Computer Science, University of Karlsruhe, Germany, 1989.

Joshi, A.M. Adaptive locking strategies in a multi-node data sharing environment. *International Conference on Very Large Data Bases,* Barcelona, Spain, 1991.

King, R.P., Halim, N., Garcia-Molina, H., and Polyzois, C.A. Management of a remote backup copy for disaster recovery. *ACM Transactions on Database Systems,* 16(2)338-368, 1991.

Korth, H.F., Levy, E., and Silberschatz, A. Compensating transactions: A new recovery paradigm. *International Conference on Very Large Data Bases,* Brisbane, Australia, 1990.

Lindsay, B., Selinger, P.G., Galteri, C., Gray, J.N., Lorie, R.A., Price, T.G., Putzolu, F., and Wade, B.W. Notes on Distributed Databases. IBM Research Report RJ2571, San Jose, California, 1979.

Lomet, D.B. MLR: A recovery method for multi-level systems. *ACM SIGMOD International Conference on the Management of Data,* San Diego, California, 1992.

Martin, B.E. Modeling concurrent activities with nested objects. *International Conference on Distributed Computing Systems,* Berlin, 1987.

Moenkeberg, A. and Weikum, G. Conflict-driven load control for the avoidance of data-contention thrashing. *IEEE International Conference on Data Engineering,* Kobe, Japan, 1991.

Moenkeberg, A. and Weikum, G. Performance analysis of an adaptive and robust load control method for the avoidance of data-contention thrashing. *International Conference on Very Large Data Bases,* Vancouver, Canada, 1992.

Mohan, C. and Pirahesh, H. ARIES-RRH: Restricted repeating of history in the ARIES transaction recovery method. *IEEE International Conference on Data Engineering,* Kobe, Japan, 1991.

Mohan, C., Haderle, D., Lindsay, B., Pirahesh, H., and Schwarz, P. ARIES: A transaction recovery method supporting fine-granularity locking and partial rollbacks using write-ahead logging. *ACM Transactions on Database Systems,* 17(1):94-162, 1992.

Mohan, C. and Levine, F. ARIES/IM: An efficient and high concurrency index management method using write-ahead logging. *ACM SIGMOD International Conference on the Management of Data,* San Diego, California, 1992.

Moss, J.E.B. *Nested Transactions: An Approach to Reliable Distributed Computing,* Cambridge, Massachusetts: MIT Press, 1985.

Moss, J.E.B., Griffeth, N.D., and Graham, M.H. Abstraction in recovery management. *ACM SIGMOD International Conference on the Management of Data,* Washington, DC, 1986.

Moss, J.E.B., Leban, B., and Chrysanthis, P.K. Finer grained concurrency for the database cache. *IEEE International Conference on Data Engineering,* Los Angeles, 1987.

Murphy, M.C., and Shan, M.-C. Execution plan balancing. *IEEE International Conference on Data Engineering,* Kobe, Japan, 1991.

Muth, P. and Rakow, T. Atomic commitment for integrated database systems. *IEEE International Conference on Data Engineering,* Kobe, Japan, 1991.

Muth, P., Rakow, T., Weikum, G., Broessler, P., and Hasse, C. Semantic concurrency control in object-oriented database systems. *IEEE International Conference on Data Engineering,* Vienna, 1993.

O'Neil, P.E. The escrow transactional method. *ACM Transactions on Database Systems,* 11(4):405-430, 1986.

Ong, K.S. Synapse approach to database recovery. *International Symposium on Principles of Database Systems,* Waterloo, Canada, 1984.

Patterson, D.A., Gibson, G., and Katz, R.H. A case for redundant arrays of inexpensive disks (RAID). *ACM SIGMOD International Conference on the Management of Data,* Chicago, 1988.

Pirahesh, H., Mohan, C., Cheng, J., Liu, T.S., and Selinger, P. Parallelism in relational database systems: Architectural issues and design approaches. *Second International Symposium on Databases in Parallel and Distributed Systems,* Dublin, Ireland, 1990.

Rakow, T.C., Gu, J., and Neuhold, E.J. Serializability in object-oriented database systems. *IEEE International Conference on Data Engineering,* Los Angeles, 1990.

Schek, H.-J., Paul, H.-B., Scholl, M.H., and Weikum, G. The DASDBS project: Objectives, experiences, and future prospects. *IEEE Transactions on Knowledge and Data Engineering,* 2(1):25-43, 1990.

Schwarz, P.M. and Spector, A.Z. Synchronizing shared abstract types. *ACM Transactions on Computer Systems,* 2(3):223-251, 1984.

Seltzer, M. and Stonebraker, M. Transaction support in read optimized and write optimized file systems. *International Conference on Very Large Data Bases,* Brisbane, Australia, 1990.

Shasha, D. What good are concurrent search structure algorithms for databases anyway? *IEEE Database Engineering,* 8(2):84-90, 1985.

Shasha, D. and Goodman, N. Concurrent search structure algorithms. *ACM Transactions on Database Systems,* 13(1):53-90, 1988.

Shrivastava, S.K., Dixon, G.N., and Parrington, G.D. An overview of the arjuna distributed programming system. *IEEE Software,* 8(1):66-73, 1991.

Skarra, A.H. and Zdonik, S.B. Concurrency control and object-oriented databases. In: Kim, W. and Lochovsky, F.H. eds., *Object-Oriented Concepts, Databases, and Applications.* New York: ACM Press, 1989.

Thomasian, A. Two-phase locking performance and its thrashing behavior. To appear in: *ACM Transactions on Database Systems,* in press.

von Bueltzingsloewen, G., Iochpe, C., Liedtke, R.-P., Dittrich, K.R., and Lockemann, P.C. Two-level transaction management in a multiprocessor database machine. *Third International Conference on Data and Knowledge Bases,* Jerusalem, 1988.

Weihl, W.E. Commutativity-based concurrency control for abstract data types. *IEEE Transactions on Computers,* 37(12):1488-1505, 1988.

Weihl, W.E. The impact of recovery on concurrency control. *ACM International Symposium on Principles of Database Systems,* Philadelphia, 1989.

Weikum, G. A theoretical foundation of multi-level concurrency control. *ACM International Symposium on Principles of Database Systems,* Cambridge, Massachussetts, 1986.

Weikum, G. Enhancing concurrency in layered systems. *Second International Workshop on High Performance Transaction Systems,* Pacific Grove, California, 1987.

Weikum, G., Hasse, C., Broessler, P., and Muth, P. Multi-level recovery. *ACM International Symposium on Principles of Database Systems,* Nashville, Tennessee, 1990.

Weikum, G. Principles and realization strategies of multilevel transaction management. *ACM Transactions on Database Systems,* 16(1):132-180, 1991.

Weikum, G. and Schek, H.-J. Architectural issues of transaction management in layered systems. *International Conference on Very Large Data Bases,* Singapore, 1984.

Weikum, G. and Schek, H.-J. Multi-level transactions and open-nested transactions. *IEEE Data Engineering,* 14(1):60-64, 1991.

Weikum, G. and Schek, H.-J. Concepts and applications of multilevel transactions and open-nested transactions. In: Elmagarmid, A.K., ed., *Database Transaction Models for Advanced Applications,* San Mateo, California: Morgan Kaufmann, 1992.

Weikum, G., Hasse, C., Moenkeberg, A., Rys, M., and Zabback, P. The COMFORT project (Project Synopsis). *Second International Conference on Parallel and Distributed Information Systems,* San Diego, California, 1993.