INTEGRATED CONCURRENCY CONTROL AND RECOVERY MECHANISMS: DESIGN AND PERFORMANCE EVALUATION

by

Rakesh Agrawal
David J. DeWitt

Computer Sciences Technical Report #497

February 1983

Integrated Concurrency Control and Recovery Mechanisms: Design and Performance Evaluation

Rakesh Agrawal
David J. DeWitt

Computer Sciences Department
University of Wisconsin - Madison

This research was partially supported by the National Science Foundation under grant MCS82-01870 and the Department of Energy under contract #DE-AC02-81ER10920.

ABSTRACT

In spite of the wide variety of concurrency control and recovery mechanisms proposed during the past decade, the behavior and the performance of various concurrency control and recovery mechanisms remain largely not well understood. In addition, although concurrency control and recovery mechanisms are intimately related, the interaction between them has not been adequately explored. In this paper, we take a unified view of the problems associated with concurrency control and recovery for transaction-oriented multi-user centralized database management systems, and present several integrated mechanisms. We then develop analytical models to study the behavior and compare the performance of these integrated mechanisms and present the results of our performance evaluation.

1. Introduction

During the past decade, alternative concurrency control and recovery mechanisms have been the subject of intensive research activity [4, 5, 26, 48]. However, despite the wide variety of mechanisms proposed, there remains a lack of experimental and/or analytical evidence regarding the behavior of various concurrency control and recovery mechanisms and their influence on database system performance. In addition, although concurrency control and recovery mechanisms are intimately related, they have been treated primarily as two independent problems and very little research has been devoted to explore the interaction between the two mechanisms.

In this paper, we take a *unified* view of the problems associated with concurrency control and recovery for *centralized* database management systems, and present several integrated mechanisms. We then develop analytical models to study the behavior and compare the performance of these integrated mechanisms. Our approach for evaluating the performance of these mechanisms extends the approaches used previously. Earlier evaluation efforts have primarily used metrics such as transaction thruput (i.e. number of transactions completed per second) or average response time when comparing different concurrency control mechanisms (or different versions of the same mechanism) and have ignored the overhead[1] imposed on the transaction by the concurrency control and recovery mechanism. Our approach for evaluating concurrency control and recovery mechanisms incorporates both the effect of the mechanism on the conflict rate between transactions (that has a direct effect on the thruput rate) and the overhead associated with each mechanism on the execution of the transaction. We feel that this approach provides a more accurate evaluation of the performance of the alternative mechanisms than when only the transaction thruput rate is considered.

The organization of the rest of the paper is as follows. In Section 2, we present a review of related work. Summaries of the concurrency control and recovery

[1] By overhead we mean those instructions/operations (both CPU and I/O) that would not need to be executed by the transaction *if* the transaction was run *alone* on a computer with *perfect* software and hardware.

mechanisms that we evaluated are contained in Sections 3 and Section 4 respectively. We describe the cost model that we use for performance evaluation in Section 5. Integrated recovery and concurrency control mechanisms are presented in Section 6 along with the cost equations for each. In Section 8, we present the results of our performance evaluation using the database, mass storage device, and processor characteristics specified in Section 7. Section 9 contains our conclusions and suggestions for future research. A glossary of the notation used in the paper has been provided in Appendix A.

2. Summary of Related Research

In the past, the behavior of *locking* as a concurrency control mechanism has been investigated by a number of researchers using both simulation and analytical models. Through the use of simulation, in [42] the difference in performance between a system in which locks are released as soon after the shrink point [10] as possible and a system in which locks are held until the transaction completes was found to be insignificant. Alternate methods of choosing a victim for deadlock resolution were studied in [34]. The effects of locking granularity on database performance were examined in [38] and it was demonstrated that different settings for system and application parameters may favor different locking granularities. In [31] the probability that a lock request by a transaction either results in a deadlock or forces the transaction to wait (along with the expected average waiting time) are estimated.

Locking policies were analyzed in [36] using hierarchical analytical modeling and two queuing network models have been proposed in [25] to study the effect of locking granularity on database system performance. An analysis of the probability of waiting and deadlock has been presented in [15].

Finally, analytical models have been utilized to study *log-based recovery* systems in [8, 12, 13]. These models, however, only address the issue of selecting an optimum checkpoint interval.

In [11] the performance of two concurrency control algorithms [2, 47] for distributed database systems is compared using simulation. In [32] another two algorithms [3, 29] for distributed systems have been compared. An assessment of shadows vis-a-vis logs for recovery has been given in [17].

Gray et al. in [16] first pointed out the effect of recovery on two-phase locking. Reed proposed in [37] that concurrency control and recovery are really two aspects of the same problem and implemented what he called atomic actions within a unified mechanism.

The results that we will present in Section 8 are built upon several of these earlier efforts, in particular, those of Lin and Nolte [31]. Lin and Nolte have determined, through simulation, the probability of an access request by a transaction conflicting with another request as a function of a variety of parameters including the transaction size (the number of pages touched), the number of transactions running concurrently (i.e. the multiprogramming level), the size of the database, and the access pattern of the transaction. They have also determined for two-phase locking the probability of a lock request resulting in deadlock and the average waiting time of a blocked request as a function of these same parameters. We use these results as input to our performance evaluation models. We also use the results in [15] to estimate the probability of waiting and probability of deadlock for small transactions since this range of transaction sizes is not covered by results presented in [31].

3. Summary of Concurrency Control Mechanisms Evaluated

In this paper we consider three basic approaches[2] that we feel form the basis of most concurrency control algorithms:

- Locking
- Timestamp ordering
- Validation

[2] The interested reader is encouraged to examine [4, 5] for a complete exposition of all the approaches possible.

3.1. Locking

Locking synchronizes read and write operations by denying access to a certain portion of the database to a conflicting transaction[3]. Before accessing an object, a transaction is required to own a *non-conflicting* lock on the object. Two requests for a lock on an object conflict if (a) one is for a read lock and the other for a write lock, or (b) both are for write locks.

Eswaran et al. [10] have shown that for serializability[4], transactions should obtain locks in a *two-phase* manner. A transaction is said to be two-phase if it does not perform a lock action after the first unlock action. To avoid a cascade of backups if a transaction fails, it is required that the second phase be deferred to the transaction commit point [21].

3.1.1. Deadlock

Whenever a transaction waits for a lock request to be granted, it runs the risk of waiting forever in a deadlock. *Deadlock* has been shown to be equivalent to a *cycle* in a waits-for graph [9, 23]. There are three approaches to deadlock resolution: prevention, detection and avoidance.

Prevention is a cautious scheme that does not let a transaction wait if it *may* get into a deadlock. Timestamp-based preemptive *wound-wait* and non-preemptive *wait-die* schemes proposed in [40] are examples of deadlock prevention. In deadlock *detection*, deadlocks are detected by explicitly building the waits-for graph and examining it for cycles. Deadlock *avoidance* is a conservative technique that avoids transaction restarts altogether using hierarchal allocation [22].

[3] A transaction is a sequence of actions on a database that transforms a consistent database state into another consistent state [10].

[4] serializability is the sufficient condition for consistency [10, 6, 35].

3.2. Timestamp Ordering

In locking, the ordering of transactions in a serialization order is dynamically determined while transactions are executing based on interleaving of their requests. With timestamp ordering, a serialization order is selected *a priori* and transaction execution is forced to obey this order. We will describe a *basic* implementation of timestamp ordering as presented in [4].

For each object X, the largest timestamp of any read(X) and the write(X) is recorded. Let these be R-ts(X) and W-ts(X) respectively. First consider rw-synchronization. A read(X) with timestamp TS is denied if TS < W-ts(X); otherwise, the read is permitted and R-ts(X) is set to max {R-ts(X),TS}. For a write(X) with timestamp TS, the request is rejected if TS < R-ts(X); otherwise, the write proceeds and W-ts(X) is set to max {W-ts(X),TS}. For ww-synchronization, a write(X) with timestamp TS is rejected if TS < W-ts(X); otherwise, the write is allowed and W-ts(X) is set to TS. If a read or a write request of a transaction is denied, it is aborted and restarted with a new, and larger, timestamp.

Two variations of the basic algorithm: *multiversion* and *conservative* timestamp ordering have been described in [4]. Both attempt to reduce the number of restarts induced by the basic algorithm.

3.3. Validation

Unlike the locking or the timestamp ordering approach, algorithms based on validation allow a transaction to execute unhindered to its end. At the time of commit, the transaction is *validated* to determine whether or not to commit the transaction. The rationale for the validation approach is the *optimistic* assumption that only a few transactions conflict.

Kung and Robinson [27] have developed a timestamp-based approach to validation. As a transaction executes, information about the set of objects read, written and created by the transaction is collected and at the end, the transaction is validated using one of the three validation conditions. For the remainder of the paper we will use the term "optimistic" instead of more general term "validation" to emphasize that we are specifically considering Kung and Robinson's optimistic method in the integrated mechanisms presented below.

3.4. Basic Timestamp Ordering versus Locking

For *centralized* databases and database systems, we feel that the basic timestamp ordering algorithm is very similar to locking in its behavior but has the disadvantage of inducing a larger number of restarts.

In basic timestamp ordering, the serialization order is decided a priori, whereas the serialization order is dynamically decided in locking. Because of this, when compared to locking, basic timestamp ordering is more prone to transaction restarts. Assume, for example, that ts(T2) > ts(T1) and the following sequence of operations:

```
T2 : read(X)
T2 : commit
T1 : write(X)
```

Basic timestamp mechanism will abort T1 but locking will permit both T1 and T2 to commit. Gray has observed in [14] that transaction restarts are very expensive.

We will now investigate the similarity in basic timestamp ordering and locking mechanisms. With basic timestamp ordering, a transaction's read(X) [write(X)] is translated into 3 actions: (i) checking that the timestamp associated with the access request is not less than W-ts(X) [R-ts(X)], (ii) updating R-ts(X) [W-ts(X)], and (iii) executing read(X) [write(X)]. It is necessary that these three actions are executed in an atomic fashion. Consider, for example, the consistency assertion that X=Y, assume R-ts(X) = W-ts(X) = R-ts(Y) = W-ts(Y) = 0, ts(T1) = 1, ts(T2) = 2, and the following sequence of execution:

1. T1 : read(X)
2. T1 : write(X:=X+1)
3. T2 : read(X)
4. T2 : write(X:=2*X)
5. T1 : read(Y)
6. T1 : write(Y:=Y+1)
7. T2 : read(Y)
8. T2 : write(Y:=2*Y)

Assume serial execution up to step 5. At step 6, ts(T1) is checked to be greater than R-ts(Y), write(Y) is accepted, and W-ts(Y) is set equal to 1. However, before Y is updated, processing of read(Y) at step 7 begins. Since ts(T2) > W-ts(Y), the read is accepted, R-ts(Y) is updated, and read(Y) is carried out. Subsequently, the pending write(Y) of step 6 is completed. After execution of step 8, we will have an inconsistent database. Therefore, as in the case of locking[5], while an object is being accessed, other conflicting (read and write) accesses to the object must be blocked[6].

Furthermore, if an updated object is allowed to be accessed before the transaction that updated it completes, the problem of *triggered aborts* will occur. Assume, for example, that ts(T2) > ts(T1) and the following sequence of execution:

T1 : write(X)
T2 : read(X)
T2 : commit
T1 : abort

When T1 is aborted, T2 will also have to be aborted and any updates of T2 will have to be undone. Consequently, once a transaction begins updating an object, access to that object must be blocked until the transaction either commits or aborts. This is equivalent to putting a write lock on the object and keeping that lock set until the end of the transaction (as in the case of two-phase locking).

Finally, with locking, the entries for a transaction in the lock table may be removed as soon as the transaction completes. With basic timestamp ordering, however,

[5] However, once the transaction has finished reading an object, writes to that object may be allowed to proceed unlike the two-phase locking where the read locks must be kept until the shrink point.

[6] An alternative might be to recheck after executing read(X) [write(X)] that the timestamp associated with the request is still not less than W-ts(X) [R-ts(X)] and if the test fails, abort the transaction. This solution will further increase the number of restarts induced by the basic timestamp ordering.

timestamps corresponding to a transaction may have to be maintained even after the transaction has committed. Thus, the size of the timestamp table will be, in general, larger than the size of the lock table. Hence, granting an access request, adding and removing timestamps with basic timestamp ordering will not be less expensive than acquiring and releasing locks using the locking approach.

To summarize, the only situation where timestamp ordering may offer additional concurrency over locking is the one in which a write request on an object is allowed to proceed once another transaction has *finished* reading the object. However, with timestamp ordering, a larger percentage of transactions will have to be aborted and rerun. The result is likely to be less net concurrency. In view of the above arguments, we will not consider basic timestamp ordering further.

4. Summary of Recovery Mechanisms Evaluated

In this paper we consider four basic recovery mechanisms for transaction oriented database systems[7]:

- Log
- Shadows
- Differential Files
- Versions

4.1. Recovery using Logs

The log-based approach [18] relies upon a *redundant* representation of the database on an append-only *log*. In addition to updating a data object, every update operation also creates a log record that includes information such as the transaction identifier, the object identifier, and "before" and "after" values. The log records of a transaction are threaded together. To limit the amount of work at the time of system restart, *system checkpoints* are taken periodically in an action-consistent state. At system checkpoint, buffers are flushed and a checkpoint record containing a list of all

[7] For a different classification and some of the techniques that are not directly applicable to transaction-oriented database systems, see [48].

active transactions and pointers to their most recent log records is written to the log.

4.1.1. Commit Processing

Modification of the database follows the following *write-ahead-log* protocol:

(1) Before recording uncommitted updates of a transaction on stable storage, force its before-value log records to stable storage.
(2) Before committing updates of a transaction, force all its log records to stable storage.

4.1.2. Recovery Algorithm

The essential idea is to undo the effects of uncommitted transactions by reading log records for the transaction backwards and restoring the before-values. Similarly, the actions of committed transactions are redone by scanning log records for the transaction forward from the most recent checkpoint and reapplying the after-values.

4.2. Recovery using Shadows

The fundamental idea of shadows is not to do in-place updating but rather to keep two copies of the object being updated while the transaction is still active: the modified copy and a copy of the object as it was before the transaction began. This later copy is termed the *shadow* copy. When the transaction commits, the shadow copy is replaced by the updated copy. We will present a scheme based on the ideas in [33, 28].

For each relation, there is a shadow page-table, S-Map, that is maintained in stable storage. An *incremental* current page-table, C-Map, for each transaction is formed in the main memory as the transaction updates data pages. To update a page k, if k is already in C-Map then C-Map[k].PhysicalPage is used for updating. Otherwise, a free page j is obtained for the updated copy of k and an entry is added to C-map for k with C-Map[k].PhysicalPage = j.

4.2.1. Commit processing

At commit time, all the pages updated by a transaction are forced to the stable storage. Then, for all pages k that appear in a transaction's C-map, S-Map[k].PhysicalPage must be changed to C-Map[k].PhysicalPage. Since the system may fail when S-Map has been partially updated, S-Map is updated in two phases. First, C-map is written to a *commit list* on stable storage as transaction's *precommit* record. Once the precommit record of a transaction appears on the commit list, its effects cannot be undone. Next, S-Map is updated. Since system failure in the middle of writing of an S-Map block may garbage the block, the S-Map is updated *carefully*[8]. Finally, a *commit* record for the transaction is written to the commit list.

4.2.2. Recovery Algorithm

Recovery from a transaction abort is straight-forward. First, the C-map associated with the transaction is discarded. Next, the updated data pages are reclaimed. To recover from a system crash, the commit list is examined to determine those transactions for which a precommit record appears in the list but not the commit record. For all such transactions, S-Map is updated using the precommit record.

4.3. Recovery using Differential Files

With the differential file scheme proposed in [41], all logical files comprise of two physical files: a read-only *base* file and a read-write *differential* file. The base file remains unchanged until reorganization. All updates are confined to the differential file.

4.3.1. Hypothetical Data Bases

In [45], the notion of Hypothetical Data Bases (HDB's) was introduced and in [43], it was proposed that all databases (including the real ones) be treated as hypothetical. Each relation R = (B ∪ A) - D is considered a view [44] where B is the read-only base por-

[8] As explained in [28], careful updating requires either two physical writes for each write operation.

tion of R and A and D are append-only differential relations. Intuitively, additions to R go to A and deletions go to D. Operations on R are translated into operations on B, A and D.

4.3.2. Commit Processing

Assume that each transaction has been assigned a *unique* timestamp and that the tuples in the A and D files have been widened to have an extra field TS for such a timestamp. While a transaction is active, its updates go to its *local* A_l and D_l relations that are inaccessible to other transactions. When the transaction commits, A_l and D_l are appended to the *global* A_g and D_g relations and are forced to stable storage. Finally, the timestamp of the committing transaction is written to a *CommitList* relation.

4.3.3. Recovery Algorithm

If a transaction aborts, its A_l and D_l are simply discarded and its timestamp is not appended to the CommitList relation. To recover from system crash, instead of R, start using the following view:

```
Range of (b,a,d,x) is (B_g,A_g,d_g,CommitList)

Define View R-Crash ([ (b.all) ∪
     (a.all) Where s.TS = x.TS ] - [(d.all) Where d.TS = x.TS ])
```

4.4. Recovery using Versions

In the version-oriented approach [37, 46], an object is thought of as a sequence of unchangeable *versions* that are linked together through an *object header* to form a *history* of the object. Updating an object is considered as creating a new version, while reading an object is considered as selecting the proper version[9].

A *version* is a pair consisting of *value* and *time* attributes; the time attribute specifies its *range of validity*. The *start time* of a version is the time specified in the write request that created the version. The *end time* is initially the same as the start time, but it is extended by both read and write operations to the time specified in the

[9] By following the chain emanating from the object header.

request. When a new version gets created, the end time of the preceding version is frozen. To make versions immutable, only the start time is stored with versions. The end time of only the current version is kept in the associated object header which is mutable.

4.5. Shadows vs. Versions

The version approach is, in a certain sense, a "super shadow" mechanism. No doubt, versions offer more functionality[10], and, when coupled with multiversion timestamp ordering, may have the potential of allowing more transactions to run concurrently[11]. Unfortunately, they have a severe performance penalty. The major problem is that simply reading the current version of an object causes the corresponding object header to be updated. Thus every read operation will potentially require two disk accesses. Because of their expected poor performance, we will not consider versions further.

5. The Cost Model

To evaluate the performance of various concurrency control and recovery algorithms, our cost model incorporates both the impact that the concurrency control mechanism has on the probability that the transaction will run to completion without conflicting with another transaction *and* the extra burden imposed on the transaction by the algorithm. This burden is measured in terms of CPU and I/O resources consumed by the transaction (or by the system on behalf of the transaction) to execute the concurrency control and recovery algorithm.

When a transaction is started, there are three possible outcomes:

(1) the transaction runs to completion and commits (transaction *succeeds)*,

[10] It is possible to go back in time and answer questions such as "who did what when".

[11] Recently, in [30] it was found that the multiversion timestamp ordering performed only marginally better than the basic timestamp ordering.

(2) the transaction is aborted by the user or because of invalid input data (transaction *fails),*
(3) the transaction is aborted by the system and is restarted, perhaps many times, before it completes (transaction *succeeds after rerun(s)).*

In each of these three cases, the concurrency control and recovery mechanism adds *extra* but *varying* amount of burden on the transaction.

Let us examine the third case more closely. Assume that the transaction is restarted only once. The extra burden in this case consists of two parts:

[a] the burden from the time the transaction started to the time it was aborted by the system and its effects were undone,
[b] the burden during the final successful execution of the transaction from start to commit.

Note that the burden for case 3[b] is the same as for case 1. At first glance the burden for case 3[a] appears to be equal to the burden for case 2 (assuming that the transaction fails at the same point). However, the burden for case 3[a] must also include the execution cost of the transaction before it was aborted since this cost would not have been incurred if the transaction was run by itself. Another way of viewing this scenario is that transactions always succeed unless terminated by the user[12]. However, certain successful transactions get internally restarted before they succeed, creating extra burden.

The burden, BX, imposed on a transaction by the recovery and concurrency-control algorithm utilized can be modeled as:

$$BX = B_{setup} + p_{fail} * B_{fail} + p_{succ} * B_{succ} + p_{rerun} * B_{rerun}$$

where,

B_{setup}	is the initialization cost incurred irrespective of the ultimate fate of the transaction,
p_{fail}	is the probability that the transaction fails, i.e. is aborted by the user,
B_{fail}	is the cost incurred when a transaction fails,
p_{succ}	is the probability that the transaction ultimately succeeds,

[12] If a user restarts a transaction after aborting it, it is considered to be a new transaction.

B_{succ}	is the cost incurred when a transaction succeeds (e.g. for committing the transaction),
p_{rerun}	is the probability that the transaction is rerun[13],
B_{rerun}	is the cost incurred when a transaction is aborted by the system,

and,

$$p_{succ} + p_{fail} = 1.$$

We will develop cost equations for B_{setup}, B_{succ}, B_{fail} and B_{rerun} for various integrated concurrency control and recovery mechanisms in the following section. The value of p_{fail} will be based on Gray's estimates in [20]. Knowing p_{fail}, $p_{succ} = 1 - p_{fail}$.

When locking is used as the concurrency control mechanism, we assume that transactions that run into deadlock are rerun. We will take the value of p_{ddlk}, the probability that a lock request by a transaction will result in a deadlock, from Lin-Nolte's simulation study [31] and Gray et al.'s analysis of the probability of waiting and deadlock [15]. Knowing p_{ddlk}, the probability that a transaction will be restarted, p_{rerun}, is computed by assuming that all lock requests are independent and that a deadlock may be caused only at the time of a request for a write-lock[14].

For the optimistic method of concurrency control, we will assume that if an access to an object by a transaction conflicts with the objects accessed by another transaction, then the probability that the transaction will be restarted is 0.5. To see this, consider conflicting accesses to an object X by transactions T1 and T2 and two scenarios as shown in Figure 1. In the first case, T1 is aborted, while in the second case, T1 runs to completion. Values for $p_{conflict}$ are based again on earlier simulation and analytical analyses [31, 15]. We again assume accesses to be independent and compute p_{rerun} further assuming that only conflicting write accesses result in transaction aborts to be consistent with the assumptions made for locking.

[13] If a transaction is restarted more than once, it is modeled by suitably adjusting the value of p_{rerun}.

[14] The assumption that only write accesses may cause a a transaction to be aborted underestimates the probability that a transaction will be restarted. In Lin-Nolte's simulation [31] and in Gray et. al.'s analysis [15], it has been assumed that all locks are exclusive. Thus, in order to use their results we had to assume that for both the locking and optimistic mechanisms only a conflicting write access will cause a transaction abort. A sensitivity analysis that we performed showed that this assumption tends to favor the optimistic method.

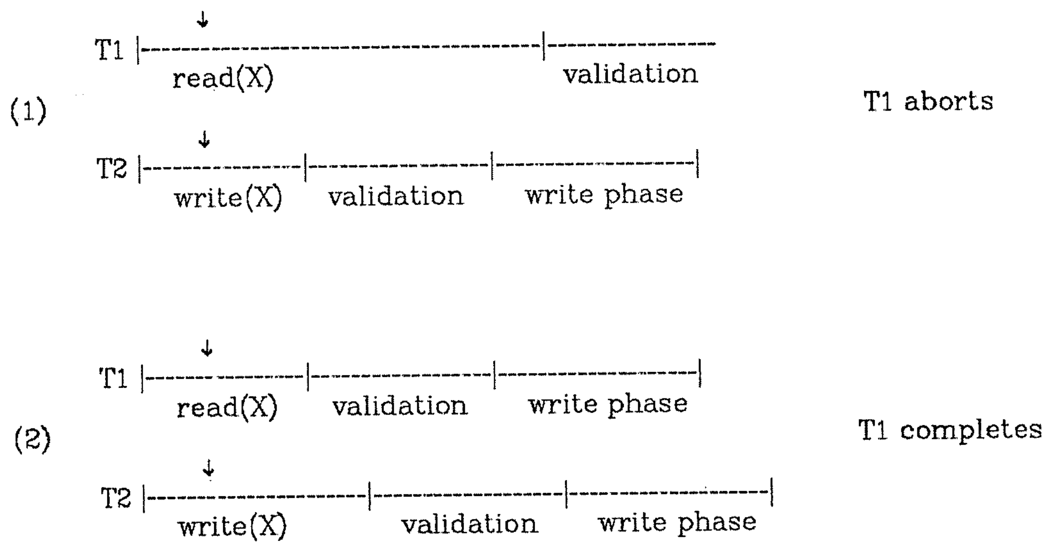

Figure 1. Transaction conflict with optimistic concurrency control

The Transaction Model

Any database system is characterized by a mix of read-only and the read/write transactions. We will model a transaction by the total number of database pages it touches, NP_t of which NP_u pages are updated (we assume that a transaction reads a page before updating it). Finally, for purposes of simplicity, we assume that each page is read by a transaction exactly once regardless of the number of records that must be accessed on the page[15].

6. Cost Equations

Before presenting integrated recovery and concurrency control mechanisms and their associated cost equations, we first specify the system parameters used in these

[15] Note that our model can be extended by including a parameter APPEND% to specify the fraction of update pages that were created by the transaction. These pages are not read before they are updated.

cost equations and state our assumptions about the concurrency control mechanisms. Assumptions about the recovery mechanisms will be described along with the cost equations for the integrated mechanisms. Appendix A contains a glossary of the notation used in the cost equations.

6.1. System Parameters

Table 1 shows the system parameters used in the development of cost equations for various recovery and concurrency control algorithms. The actual values of these parameters will depend on the physical characteristics of the stable storage device (assumed to be a disk) and the processing unit. Some parameters also depend on the characteristics of the database for which average values have been assumed. Before evaluating the performance of the alternative integrated concurrency control and recovery mechanisms in Section 8, values will be assigned to these parameters in Section 7.

$T_{l\text{-}io}$	time to read/write a disk page with disk seek
$T_{s\text{-}io}$	time to read/write a disk page without a seek
T_{page}	cpu time to process a page in memory
T_{rec}	cpu time to process a record in memory
DBSize	Size of the database
MPL	Level of multiprogramming

Table 1. System parameters

6.2. Assumptions About the Concurrency Control Mechanisms

6.2.1. Locking-Based Concurrency Control

1. The lock-acquisition discipline is "get only when needed".

2. The time to process a lock acquisition request is T_{al} and the time to process a lock

release request is T_{rl}[16]. The probability that a lock request will conflict is $p_{conflict}$ and p_{wait} is the probability that a request will be queued. T_{wait} is the wait time for a blocked request.

3. The granularity of locking is a page[17].

4. Transaction abort (either system initiated or user initiated) occurs when the transaction has read $NP_t/2$ pages and has updated $NP_u/2$ pages.

5. Deadlocks are resolved by checking for cycles in the waits-for-graph at each lock request that conflicts[18]. T_{ddlk} is the cpu time required for this test and p_{ddlk} is the probability that a cycle would be found. Thus, the probability that a lock request waits, $p_{wait} = p_{conflict} - p_{ddlk}$.

6.3. Optimistic Concurrency Control

1. The granularity of the elements in the various control sets (readset, writeset etc.) is a page.

2. The cost of creating various control sets is a function of NP_t[19]. When $NP_t = 1$, the cpu time to create the control sets is assumed to be T_{as}. We assume that a background process is responsible for deleting various control sets and we will not model this cost.

---

[16] Gray [19] asserts that the lock table can always be maintained in the main memory, and that this is the case in IMS and System R. Lin and Nolte [31] in their simulation of two-phase locking assumed the lock processing to be instantaneous. If, however, the lock table must be maintained on secondary storage, it can be modeled by choosing appropriate higher values of T_{al} and T_{rl}.

[17] A page may not necessarily be the best level of granularity [38], but we will assume it to be so uniformly for all the algorithms.

[18] Deadlock prevention using one of the timestamp-based schemes proposed by Rosenkrantz et al. [40] can be modeled by assuming that half of the conflicting requests wait and the other half result in transaction aborts, that is, $p_{wait} = p_{rerun} = p_{conflict}/2$.

An alternative is to avoid the problem of deadlock altogether by assuming that all the locks needed for executing a transaction are requested at the initiation of the transaction as in [38, 36]. If any lock cannot be granted, the transaction releases all the locks and tries again. This can be modeled by choosing a larger value for T_{al}.

[19] A much finer analysis is possible where the size of various sets is estimated and accordingly the cost of creating various sets is determined. However, because of the coarse granularity chosen for the elements of these sets, they can be maintained in the main memory and the creation of sets would not contribute significantly to the total cost.

3. Reads and writes on a page with optimistic concurrency control first check the write set to determine whether the local copy exists of the corresponding page. Since the write set can be maintained in the main memory, we will assume the cost of this indirection to be negligible[20].

4. If a transaction is aborted by the user, it happens when the transaction has completed half of its read phase.

5. If a transaction fails to be validated, it is detected half way through the validation test.

6. The cost of validating a transaction is a function of the size of the transaction and the number of concurrently executing transactions, MPL. We will assume the cost of validation to be (MPL-1) * T_{valid}, where T_{valid} is the time to validate a transaction if only one other transaction executing concurrently with it.

6.4. Integrated Mechanisms

We will now sketch integrated recovery and concurrency control mechanisms and present cost equations for them.

6.4.1. Log+Locking

This is the well known scheme described in [18]. A transaction before accessing a data page acquires a lock on it and the database is updated using the "write-ahead log" protocol.

6.4.1.1. Assumptions

1. The number of log pages generated by a transaction is determined by the parameter Log% (Number of log pages = Log% * NP_u. NP_u is the number of pages updated by the transaction).

[20] If desired, the extra cost can be modeled by choosing a larger value of $T_{i/o}$ for reads and writes.

2. We postulate a function DFlush(X) that, given the total number of data pages X updated by a transaction at some time t, returns the number of pages that have migrated to disk at time t and are not present in the main memory. Similarly, the function LFlush(Y), where Y is the number of log pages generated by a transaction at time t, returns the number of log pages that have been written to the disk and are no longer available in the main memory at time t[21].

For the write-ahead protocol, it *must* be the case that for all time t

$$\text{LFlush}(Y) \geq \text{Log\%} * \text{DFlush}(X).$$

3. Writing a log page does not require a disk seek except when a complete cylinder has been filled with log pages. As specified in Section 7, we account for the cost of this seek by amortizing it across all write operations to the cylinder.

4. Since we assume that on average a transaction gets into deadlock after reading and processing $NP_t/2$ pages and updating $NP_u/2$ pages, the execution cost of an aborted transaction = cost of reading $NP_t/2$ pages + cost of processing $NP_t/2$ pages + cost of updating $NP_u/2$ pages = $(T_{l\text{-}io}+T_{page})*NP_t/2 + \text{DFlush}(NP_u/2)*T_{l\text{-}io}$.

6.4.1.2. Cost Equations

B_{setup} = cost of writing the tran-begin log record $\{ T_{s\text{-}io} \}$
+ cost of writing the commit/abort log record $\{ T_{s\text{-}io} \}$

B_{succ} = cost of acquiring locks[22] $\{ NP_t*T_{al} + NP_t*p_{conflict}*T_{ddlk} + NP_t*p_{wait}*T_{wait} \}$
+ cpu cost of creating log pages $\{ \text{ceil}(\text{Log\%} * NP_u) * T_{page} \}$
+ i/o cost of writing log pages $\{ \text{ceil}(\text{Log\%} * NP_u) * T_{s\text{-}io} \}$

[21] Sometimes, the flushing of data and log buffers is delayed as much as possible until the transaction commits or aborts in order to reduce the cost of undo processing. On the other hand, data and log buffers may be flushed as soon as they are created to increase parallelism and minimize commit time duration.
For the updated data pages, the first situation can be modeled by defining DFlush to be DFlush(X) = max {0,X-DBuff} where, DBuff is the number of data buffers allocated to the transaction. The second situation can be modeled by defining DFlush as DFlush(X) = X. The function LFlush may be defined analogously.

[22] Cost of acquiring locks = Cost of (requesting locks + deadlock detection + waiting for locks)

+ cost of releasing locks { $NP_t * T_{rl}$ }

B_{fail} = burden before the transaction abort
+ cost of undo processing

= cost of acquiring and releasing locks {$(T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl})*NP_t/2$}

+ cpu cost of creating log pages { $\text{ceil}(\text{Log\%} * NP_u/2) * T_{page}$ }

+ i/o cost of writing log pages { $\text{ceil}(\text{Log\%} * NP_u/2) * T_{s\text{-}io}$ }

+ i/o cost of reading flushed log pages for undo { $\text{LFlush}(\text{ceil}(\text{Log\%}*NP_u/2)) * T_{l\text{-}io}$ }

+ i/o cost of reading flushed data pages for undo { $\text{DFlush}(NP_u/2) * T_{l\text{-}io}$ }

+ cpu cost of undoing corrupted data pages { $\text{DFlush}(NP_u/2) * T_{page}$ }

+ i/o cost of writing undone pages { $\text{DFlush}(NP_u/2) * T_{l\text{-}io}$ }

B_{rerun} = B_{fail} + transaction execution cost before abort

= $B_{fail} + (T_{l\text{-}io}+T_{page})*NP_t/2 + \text{DFlush}(NP_u/2)*T_{l\text{-}io}$

6.4.1.3. Comments

Instead of incurring separate I/O's for writing the tran-begin and the commit/abort records, they can be written along with other log records for the transaction on the same page. In this case, we can assume that $B_{setup} = 0$.

6.4.2. Log+Optimistic

Transactions execute unhindered but instead of making separate local copies of updated objects during the read phase as required for concurrency control, the log records are used. However, although it is possible to derive the writeset and the createset of a transaction by examining its log records, it is more efficient to create them separately in main memory. During the write phase of the transaction, log pages are used to make the updates global while observing the write-ahead-log protocol.

6.4.2.1. Assumptions

Assumptions 1-3 of the log+locking mechanism are again assumed to hold. However, reading the log in order to make the local copies global will necessitate disk seeks as log pages from one transaction may not be physically adjacent on the disk.

Observe that the decision to abort a non-serializable transaction is taken after the completion of its read phase, i.e. after reading and processing NP_t pages, but no updated pages are written during the read phase (log records double up as local copies during the read phase). Hence, the execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t$.

6.4.2.2. Cost Equations

B_{setup} = cost of writing the tran-begin log record $\{ T_{s\text{-}io} \}$

+ cost of writing the commit/abort log record $\{ T_{s\text{-}io} \}$

B_{succ} = cost of creating control sets $\{ NP_t * T_{as} \}$

+ cpu cost of creating log pages $\{ \text{ceil}(\text{Log\%}*NP_u) * T_{page} \}$

+ i/o cost of writing log pages $\{ \text{ceil}(\text{Log\%}*NP_u) * T_{s\text{-}io} \}$

+ cost of validation test $\{ (MPL-1) * T_{valid} \}$

+ cost of making local copies global[23] $\{\text{LFlush}(\text{ceil}(\text{Log\%}*NP_u))*T_{l\text{-}io}+\text{DFlush}(NP_u)*T_{l\text{-}io}\}$

B_{fail} = burden before the transaction abort + cost of undo processing (= 0)[24]

= cost of creating the control sets $\{ NP_t/2 * T_{as} \}$

+ cpu cost of creating log pages $\{ \text{ceil}(\text{Log\%}*NP_u/2) * T_{page} \}$

+ i/o cost of writing log pages $\{ \text{LFlush}(\text{ceil}(\text{Log\%}*NP_u/2)) * T_{s\text{-}io} \}$

- cost of writing DFlush(NP_u/2) data pages[25] $\{ \text{DFlush}(NP_u/2) * T_{l\text{-}io} \}$

[23] The cost of making local copies global involves reading the log pages that have migrated to disk and the data pages to be updated that are no longer available in main memory. However, it would not include the cost of updating the data pages in the main memory and writing back the updated pages. These costs are not incurred during the read phase of the transaction and hence can be amortized during this phase.

[24] No undo processing is required as at this point all changes have been performed on the local copies.

B_{rerun} = cost of creating the control sets { $NP_t * T_{as}$ }
+ cpu cost of creating log pages { $ceil(Log\%*NP_u) * T_{page}$ }
+ i/o cost of writing log pages { $LFlush(ceil(Log\%*NP_u)) * T_{s\text{-}io}$ }
+ cost of the validation test { $(MPL-1) * T_{valid} / 2$ }
+ transaction execution cost before abort { $(T_{l\text{-}io}+T_{page})*NP_t$ }

6.4.2.3. Comments

As in the case of the log+lock algorithm, the tran-begin and the commit/abort records for a transaction can be written together with the other log records for the transaction.

6.4.3. Shadows+Locking

Before accessing a data page, the transaction locks that page. However, no explicit locking is needed to access page-table (both S-Map and C-map) entries. The protocol required is that a transaction accesses a page-table entry to get the physical address of a data page only if it has been granted a lock for that page. Thus, it is not possible for a transaction to access a page-table entry while it is being updated. Once a transaction completes, its write-lock on a page is released only after the corresponding entry in the page-table has been updated.

6.4.3.1. Assumptions

1. The size of the page-table is PtSize pages. For relations of reasonable size, PtSize will be large. Thus, the S-Map cannot reside in the main memory and must be paged from the secondary storage [17]. Consequently a data page I/O may also cause a page-table I/O. However, in general, accessing X data pages will not result in accessing X distinct pages of the S-Map since a number of page-table entries can be blocked into one

[25] Since we are developing formulas that express the overhead (burden) incurred, we must model savings provided by a mechanism as well as costs. Thus, since no pages are actually updated in the log+optimistic approach until the transaction is validated, $DFlush(NP_u/2)$ write operations are avoided when compared with a system that provides no recovery mechanism and does in-place updating.)

S-Map page. The number of S-Map pages that may have to be accessed will be determined by the function PtPages(X). For the random access of data pages, the number of S-Map pages required to be accessed is analogous to the number of pages accessed when randomly selecting records from a blocked file. We will use the Cardenas' expression [7] for this purpose[26], and define

$$\text{PtPages}(X) = \text{PtSize}(1 - (1 - 1/\text{PtSize})^{X}).$$

For sequential access of data pages,

$$\text{PtPages}(X) = 1 + X / \text{blocking-factor}^{27}.$$

2. The tran-begin and the incremental C-Map can be written on the same page as the pre-commit record on the commit list.

3. The function SFlush(X), where X is the number of S-Map pages read by the transaction at time t, returns the number of pages that are no longer available in the memory. SFlush(X) = max {0,X-SBuff} where SBuff is the number of buffers available to the transaction for reading the S-Map pages. The function DFlush(Y) which returns the number of updated pages that have migrated to the disk is defined analogously.

4. A shadow-based algorithm generates extra NP_u allocate-page and free-page requests for data pages when compared to an in-place updating algorithm. The cost of processing an allocate-page or a free-page request will be assumed to be T_{rec}.

5. Writing to the commit list does not require a disk seek.

6. The cost of creating an entry in the C-Map is T_{rec}.

7. As in the case of log+locking mechanism, the execution cost of an aborted transac-

[26] It has been shown that the Cardenas' expression gives the lower bound for the expected number of pages accessed and more accurate expressions are available in literature (see [49]). However, for large blocking factors (> 10) such as would be present in the S-Map, the error in Cardenas' approximation is practically negligible.

[27] One has been added to account for the fact that the desired page-table entries may not start at the beginning of a page-table page.

tion = $(T_{l\text{-}io}+T_{page})*NP_t/2$ + $DFlush(NP_u/2)*T_{l\text{-}io}$.

6.4.3.2. Cost Equations

B_{setup} = cost of writing commit/abort record = $T_{s\text{-}io}$

B_{succ} = cost of acquiring locks { $NP_t * (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait})$ }

+ i/o cost of reading S-Map pages for data reads { $PtPages(NP_t) * T_{l\text{-}io}$ }

+ cost of extra allocate-page requests { $NP_u * T_{rec}$ }

+ cpu cost of creating incremental C-Map { $NP_u * T_{rec}$ }

+ i/o cost of writing the pre-commit record { $T_{s\text{-}io}$ }

+ i/o cost of rereading flushed out S-Map pages { $SFlush(PtPages(NP_u)) * T_{l\text{-}io}$ }

+ cpu cost of updating S-Map entries { $NP_u * T_{rec}$ }

+ i/o cost of writing the updated S-Map pages { $PtPages(NP_u) * T_{l\text{-}io}$ }

+ cost of releasing locks { $NP_t * T_{rl}$ }

+ cost of extra free-page requests { $NP_u * T_{rec}$ }

B_{fail} = burden before the transaction abort + cost of undo processing

= cost of acquiring and releasing locks {$(T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl})*NP_t/2$}

+ i/o cost of reading S-Map pages for data reads { $PtPages(NP_t/2) * T_{l\text{-}io}$ }

+ cost of extra allocate-page requests { $NP_u/2 * T_{rec}$ }

+ cpu cost of creating incremental C-Map { $NP_u/2 * T_{rec}$ }

+ cost of extra free-page requests { $NP_u/2 * T_{rec}$ }

B_{rerun} = B_{fail} + transaction execution cost before abort

= $B_{fail} + (T_{l\text{-}io}+T_{page})*NP_t/2 + DFlush(NP_u/2)*T_{l\text{-}io}$

6.4.3.3. Comments

1. The writing of commit/abort can be piggybacked with the pre-commit record of the next transaction at the expense of increasing somewhat the response time of the transaction.

2. It is possible to avoid writing the abort record when a transaction is aborted by the user. However, the disadvantage is that at the time of the recovery from system crash, it would not be possible to distinguish the user-aborted transactions from those that were active at the time of crash, and they may get restarted.

6.4.4. Shadows+Optimistic Algorithm

When shadows are used as a recovery mechanism, there are always two copies of each data page being updated by a transaction: the updated copy and the unmodified (shadow) copy on disk. When shadows are combined with an optimistic concurrency algorithm, the updated copy of each data page being modified can also be used as the local copy for concurrency control purposes.

For purposes of concurrency control, as a transaction executes, it creates various control sets (readset, writeset etc.). There is, however, no need to create a C-Map as required by recovery mechanism since the writeset (which normally contains only the page numbers of the updated pages) can be augmented to include the disk addresses of the modified pages along with the page numbers. With this approach, the write phase in which local copies are made global requires simply updating the S-map entries using the writeset to point to new disk addresses.

Note that it is not required to keep S-Map or C-Map page numbers accessed by a transaction in its control sets. If the updates to S-Map by a transaction have been partially applied and meanwhile another transaction reads the not yet updated S-Map entries, that transaction will not be validated.

6.4.4.1. Assumptions

We assume that assumptions 1-5 of the shadows+locking mechanism are valid for this mechanism also. However, since the decision to abort a non-serializable transaction is taken after the completion of its read phase, the execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t + DFlush(NP_u)*T_{l\text{-}io}$

6.4.4.2. Cost Equations

B_{setup} = cost of writing commit/abort record = $T_{s\text{-}io}$

B_{succ} = cost of creating the control sets { $NP_t * T_{as}$ }
+ i/o cost of reading S-Map pages for data reads { $PtPages(NP_t) * T_{l\text{-}io}$ }
+ cost of extra allocate-page requests { $NP_u * T_{rec}$ }
+ cost of validation test { $(MPL-1) * T_{valid}$ }
+ i/o cost of writing the pre-commit record { $T_{s\text{-}io}$ }
+ i/o cost of reading flushed out S-Map pages { $SFlush(PtPages(NP_u)) * T_{l\text{-}io}$ }
+ cpu cost of updating S-Map entries { $NP_u * T_{rec}$ }
+ i/o cost of writing the updated S-Map pages { $PtPages(NP_u) * T_{l\text{-}io}$ }
+ cost of extra free-page requests { $NP_u * T_{rec}$ }

B_{fail} = burden before the transaction abort + cost of undo processing
= cost of creating the control sets { $NP_t/2 * T_{as}$ }
+ i/o cost of reading S-Map pages for data reads { $PtPages(NP_t/2) * T_{l\text{-}io}$ }
+ cost of extra allocate-page requests { $NP_u/2 * T_{rec}$ }
+ cost of extra free-page requests { $NP_u/2 * T_{rec}$ }

B_{rerun} = burden before the transaction abort + cost of undo processing
+ transaction execution cost before abort
= cost of creating the control sets { $NP_t * T_{as}$ }
+ i/o cost of reading S-Map pages for data reads { $PtPages(NP_t) * T_{l\text{-}io}$ }
+ cost of extra allocate-page requests { $NP_u * T_{rec}$ }
+ cost of validation test { $(MPL-1) * T_{valid} / 2$ }
+ cost of extra free-page requests { $NP_u * T_{rec}$ }
+ transaction execution cost { $(T_{l\text{-}io}+T_{page})*NP_t/2 + DFlush(NP_u/2)*T_{l\text{-}io}$ }

6.4.4.3. Comments

As in the case of the shadow+lock algorithm, the commit/abort record may be piggybacked with the pre-commit record of the next transaction. Also, the writing of the abort record in the case of a user-initiated transaction abort may be avoided.

6.4.5. Differential File+Lock Algorithm

As the transaction executes, it locks the pages of the global base relation, B_g, the and differential relations, A_g and D_g[28], and creates the local differential relations A_l and D_l. Once the transaction commits, A_l and D_l are appended to A_g and D_g.

6.4.5.1. Assumptions

1. The sizes of the differential relations, A_g and D_g, are Size% of the size of the base relation, B_g. We will assume A_g and D_g to be of equal size.

2. Accessing NP_t pages of R needs Xtra% extra page accesses. Xtra% is a function of the size of A_g and D_g. We assume that the transaction is executed when half of A_g and half of D_g have been created. Thus, Size%/2 extra pages from A_g and D_g will be read.

3. Processing of pages in the memory also incurs extra cpu overhead[29]. We will assume the extra cpu overhead to be CpuOH% of the total cpu time consumed if the transaction was run alone without any provision for recovery.

4. The number of A_l and D_l pages generated by a completed transaction is Comprs% of NP_u. Thus, a transaction writes ceil(Comprs%*NP_u) pages to both A_l and D_l. However, in-place updating would incur the cost of writing NP_u pages. Therefore, the net cost is the cost of writing (2*ceil(Comprs%*NP_u) - NP_u) pages.

[28] Recall that $R = (B \cup A) - D$

[29] For example, since $R = (B \cup A) - D$, a retrieve on R will be translated into a retrieve on B, A, and D followed first by a union of tuples retrieved from B and A and then by a set-difference of the result and tuples retrieved from D.

5. DFlush(X) is the function that returns the number of A_l and D_l pages that migrate to disk[30].

6. Writing to the commit list does not require a disk seek.

7. The execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t/2$. Note that unlike an in-place updating mechanism like log+locking, with the differential file approach the transaction does not incur the cost of writing updated pages. The cost of writing A and D pages is considered to be recovery burden associated with the differential file approach.

6.4.5.2. Cost Equations

$B_{setup} = 0$[31]

B_{succ} = cost of acquiring locks $\{ (1+Size\%)*NP_t * (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}) \}$

+ cost of extra data page reads $\{ Size\%*NP_t * T_{l\text{-}io} \}$

+ extra cpu cost of processing data pages $\{ CpuOH\% * (NP_t*T_{page}) \}$

+ cost of writing flushed A_l and D_l pages $\{ 2 * DFlush(ceil(Comprs\%*NP_u)) * T_{l\text{-}io} \}$

+ cost of rereading flushed A_l and D_l pages $\{ 2 * DFlush(ceil(Comprs\%*NP_u)) * T_{l\text{-}io} \}$

+ net cost of writing A_g and D_g pages[32] $\{ (2*ceil(Comprs\%*NP_u) - NP_u) * T_{l\text{-}io} \}$

+ cost of extra allocate-page requests $\{ 2 * ceil(Comprs\%*NP_u) * T_{rec} \}$

+ cost of releasing locks $\{ (1+Size\%)NP_t * T_{rl} \}$

+ cost of writing the tran-id to commit list $\{ T_{s\text{-}io} \}$

B_{fail} = burden before the transaction abort + cost of undo processing

= cost of acquiring and releasing locks
$\{ (1+Size\%)*NP_t/2 * (T_{al}+p_{conflict}*T_{ddlk}+p_{wait}*T_{wait}+T_{rl}) \}$

+ cost of extra data page reads $\{ Size\%*NP_t/2 * T_{l\text{-}io} \}$

---

[30] DFlush(X) = max {0,DBuff-X} where DBuff is the number of buffers available.

[31] Only the tran-id of committed transactions is written to commit list and this cost is included in B_{succ}.

[32] See assumption 3 above.

+ extra cpu cost of processing data pages { CpuOH% * $(NP_t/2*T_{page})$ }

+ net cost of writing flushed A_l and D_l pages
{ (2 * DFlush(ceil(Comprs%*NP_u/2)) - DFlush(NP_u/2)) * $T_{l\text{-}io}$ }

B_{rerun}[33] = B_{fail} + cost of writing DFlush(NP_u/2) pages { DFlush(NP_u/2) * $T_{l\text{-}io}$ }
+ transaction execution cost before abort { $(T_{l\text{-}io}+T_{page})*NP_t/2$ }

6.4.6. Differential File+Optimistic Algorithm

First observe that the A_l and D_l can also be used for the local copies of modified records for concurrency control purposes. As the transaction executes, it creates control sets and if it is validated, it appends A_l and D_l to global A_g and D_g.

6.4.6.1. Assumptions

The same assumptions stated for differential file+locking are assumed to hold. As in the case of differential file+locking mechanism, the execution cost of an aborted transaction = $(T_{l\text{-}io}+T_{page})*NP_t$. The difference is that the decision to abort the transaction is taken after reading and writing NP_t pages, instead of $NP_t/2$ pages as in the case of differential file+locking mechanism.

6.4.6.2. Cost Equations

$B_{setup} = 0$

B_{succ} = cost of creating the control sets { (1+Size%)*NP_t * T_{as} }
+ cost of extra data page reads { Size%*NP_t * $T_{l\text{-}io}$ }
+ extra cpu cost of processing data pages { CpuOH% * (NP_t*T_{page}) }
+ cost of writing flushed A_l and D_l pages { 2 * DFlush(ceil(Comprs%*NP_u)) * $T_{l\text{-}io}$ }
+ cost of validation test { (MPL-1) * T_{valid} }
+ cost of re-reading flushed A_l and D_l pages { 2 * DFlush(ceil(Comprs%*NP_u)) * $T_{l\text{-}io}$ }

[33] The execution cost before abort does not incur the cost of writing DFlush(NP_u/2) pages as is the case in an in-place updating algorithm. However, since the cost of writing DFlush(NP_u/2) pages was subtracted from B_{fail}, this cost will be added here in order to make the formula correct.

+ net cost of writing A_g and D_g pages { $(2*\text{ceil}(\text{Comprs}\%*NP_u) - NP_u) * T_{l\text{-}io}$ }

+ cost of extra allocate-page requests { $2*\text{ceil}(\text{Comprs}\%*NP_u) * T_{rec}$ }

+ cost of writing the tran-id to commit list { $T_{s\text{-}io}$ }

B_{fail} = burden before the transaction abort + cost of undo processing (=0)

= cost of creating the control sets { $(1+\text{Size}\%)*NP_t/2 * T_{as}$ }

+ cost of extra data page reads { $\text{Size}\%*NP_t/2 * T_{l\text{-}io}$ }

+ extra cpu cost of processing data pages { $\text{CpuOH}\% * (NP_t/2*T_{page})$ }

+ net cost of writing flushed A_l and D_l pages
{ $2 * \text{DFlush}(\text{ceil}(\text{Comprs}\%*NP_u/2)) - \text{DFlush}(NP_u/2)) * T_{l\text{-}io}$ }

B_{rerun} = burden before the transaction abort + cost of undo processing (=0)
+ transaction execution cost before abort

= cost of creating the control sets { $(1+\text{Size}\%)*NP_t * T_{as}$ }

+ cost of extra data page reads { $\text{Size}\%*NP_t * T_{l\text{-}io}$ }

+ extra cpu cost of processing data pages { $\text{CpuOH}\% * (NP_t*T_{page})$ }

+ cost of writing flushed A_l and D_l pages { $2 * \text{DFlush}(\text{ceil}(\text{Comprs}\%*NP_u)) * T_{l\text{-}io}$ }

+ cost of validation test { $(MPL-1) * T_{valid} / 2$ }

+ transaction execution cost { $(T_{l\text{-}io}+T_{page})*NP_t/2$ }

7. Database, Mass Storage Device and Processor Specifications

In this section, we specify the characteristics of the database, the mass storage device, and the processor employed in our evaluation.

The mass storage device is modeled after the IBM 3350 disk drive [24] whose characteristics are shown in Table 2. We assume that to access a random block on the disk on the average the heads must be moved half way across the disk. Thus,

$$T_{l\text{-}io} = \text{Average seek time} + \text{Latency} + \text{Transfer time} = 37.525 \text{ ms.}$$

When an I/O operation is performed on an append-on file such as a log, seek operations are only occasionally necessary. For algorithms that utilize shadows or differential files for recovery, a seek operation is only needed when the current cylinder has been

Parameter	Value
No. of recording surfaces	30
No. of cylinders	555
No. of blocks per track	4
Block size	4096 bytes
Revolution time	16.7 ms.
Time to move head N cylinders	10 + 0.072*N ms.
Average seek time	25 ms.

Table 2. Disk drive Specifications

completely filled. To simplify our costs expressions, we have amortized the cost of these occasional seek operations across every write operation. Thus,

$T_{s\text{-}io}$ = Latency + Transfer time + 1/120*(time to move heads to the adjacent cylinder)[34]

= 12.61 ms.

When the append-only file is used to hold a recovery log, the disk heads must be moved from the end of the log file to perform transaction undo. In this case,

$T_{s\text{-}io}$ = Latency + Transfer time + 1/120*(time to move heads to the adjacent cylinder)

+ $(p_{fail} + p_{rerun})$ * Average Seek Time

= 12.61 + $(p_{fail} + p_{rerun})$ * 25.0 ms.

We have assumed that a 1 MIP processor is used to execute transactions, that 500 instructions are required to process a record (T_{rec} = 0.5 ms), and that 5000 instructions are required to process a page of approximately 10 records[35] (T_{page} = 5.0 ms.).

The size of the database, DBSize, has been assumed to be 100 million bytes. We have evaluated the performance of the integrated concurrency and recovery algorithms under three different workloads: small (TS), medium (TM), and large (TL). Their sizes and some associated characteristics are shown in Table 3.

Twait has been calculated using the formula:

$$T_{wait} = T_{wait}Fctr * (T_{l\text{-}io} + T_{page})$$

in which TwaitFctr=0.83 for the TS workload, 1.96 for TM, and 2.94 for TL. These

[34] There are 4 blocks per track and 30 tracks per cylinder

[35] A record can be, for example, either a database record or a log record.

Parameter	TS	TM	TL
NPt - number of pages touched	2	50	250
NPu - number of pages updated	1	15	50
MPL - multiprogramming level	15	10	7
Pfail - probability of transaction failure	0.05	0.05	0.05
Pconflict - probability of transaction conflict	0.0012	0.0065	0.01
Pddlk - probability of deadlock	1.92e-7	5.6e-6	3.0e-5
Prerun(locking) - probability of rerun	1.92e-7	8.4e-5	0.0015
Prerun(optimistic) - probability of rerun	0.0006	0.04766	0.22169
Twait - wait time for a blocked request	0.83 ms.	1.96 ms.	2.94 ms.
Tvalid - time to validate a transaction	0.1 ms.	0.5 ms.	2.0 ms.

Table 3. Transaction sizes and database characteristics

numbers and the probability figures in Table 3 are based on the results presented in [15, 20, 31].

T_{valid} is based on the assumption that a transaction can be validated against a concurrent transaction in $O(NP_t + NP_u)$ time plus the time for a procedure call. T_{al} (the time to process a lock request), T_{rl} (time to release a lock request), and T_{as} (time to construct the control sets for the optimistic concurrency control algorithm) have been assumed to be 0.5 ms. A value of 0.5ms. has also been used to represent determine whether granting a lock request will result in deadlock (T_{ddlk}) based on the results described in [1].

8. Evaluation

In this section we compare the performance of the different integrated concurrency control and recovery mechanisms by computing the *burden ratio* for each mechanism. The burden ratio is defined to be the ratio of BX (the extra burden imposed on the transaction by the concurrency control and recover mechanism) to the execution time of the transaction if run without any concurrency control or recovery mechanism:

$$\text{Burden Ratio} = \frac{BX}{\text{Execution Time of the Transaction}}$$

$$= \frac{B_{setup} + P_{fail}*B_{fail} + P_{succ}*B_{succ} + P_{rerun}*B_{rerun}}{NP_t*T_{l-io} + NP_t*T_{page} + NP_u*T_{l-io}}$$

We first compare the relative performance of locking and optimistic concurrency control for each of the recovery mechanisms. Then the performance of the three finalists are compared.

8.1. Logging

The relative performance of the log+locking and log+optimistic mechanisms is shown in Figure 2 and Table (i) in Appendix B. We assumed that DBuff, the number of data buffers allocated to the transaction equals 10, LBuff, the number of buffers available to collect log records for the transaction, equals 1, and Log%, the fraction of each updated page that must be recorded in a log record, equals 0.1.

Based on Figure 2 and the data in Table (i), we can make the following observations about the performance of these two mechanisms:

1. The operation of "making local copies global" in the optimistic concurrency control algorithm is very expensive since NP_u-DBuff data pages[36] that need to be updated will have migrated to disk before the write phase begins and will have to be reread during the write phase.

2. Backing up a user aborted transaction is more expensive with locking due to the cost of undo processing (reading back those updated pages that have migrated to the disk, undoing the changes and then rewriting them, and the cost of acquiring and releasing locks). In the case of the optimistic method, B_{fail} can actually have a negative value for large transactions as only LFlush(Log%*NP_u/2) data pages are written to the disk instead of DFlush(NP_u/2) pages[36] and no undo processing or validation cost is incurred.

3. As the average transaction size increases, the number of transaction restarts increases faster for the optimistic mechanism than for a lock-based mechanism that uses deadlock prevention. Hence, the value of B_{rerun} increases faster for the optimistic method.

[36] Recall that for TL, NPu=50, DBuff=10, LFlush(Log%*NPu/2)=2 and DFlush(NPu/2)=15.

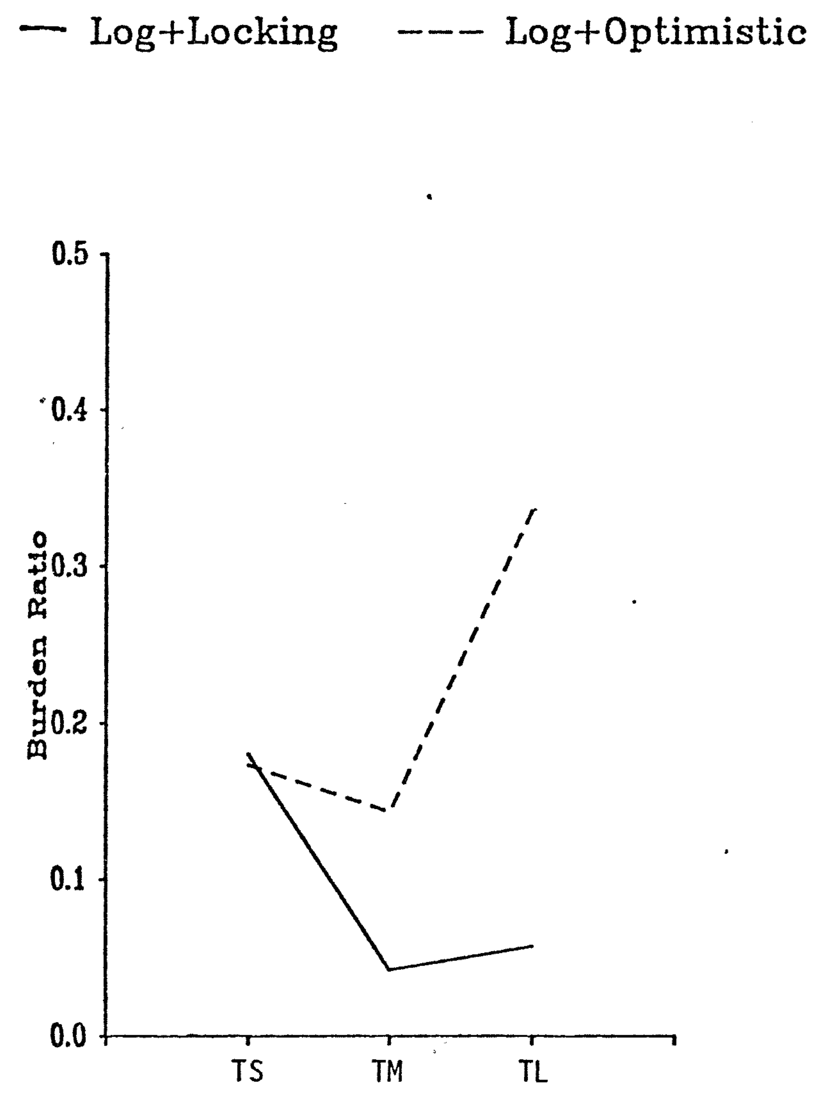

Figure 2. Performance of Log+Locking and Log+Optimistic

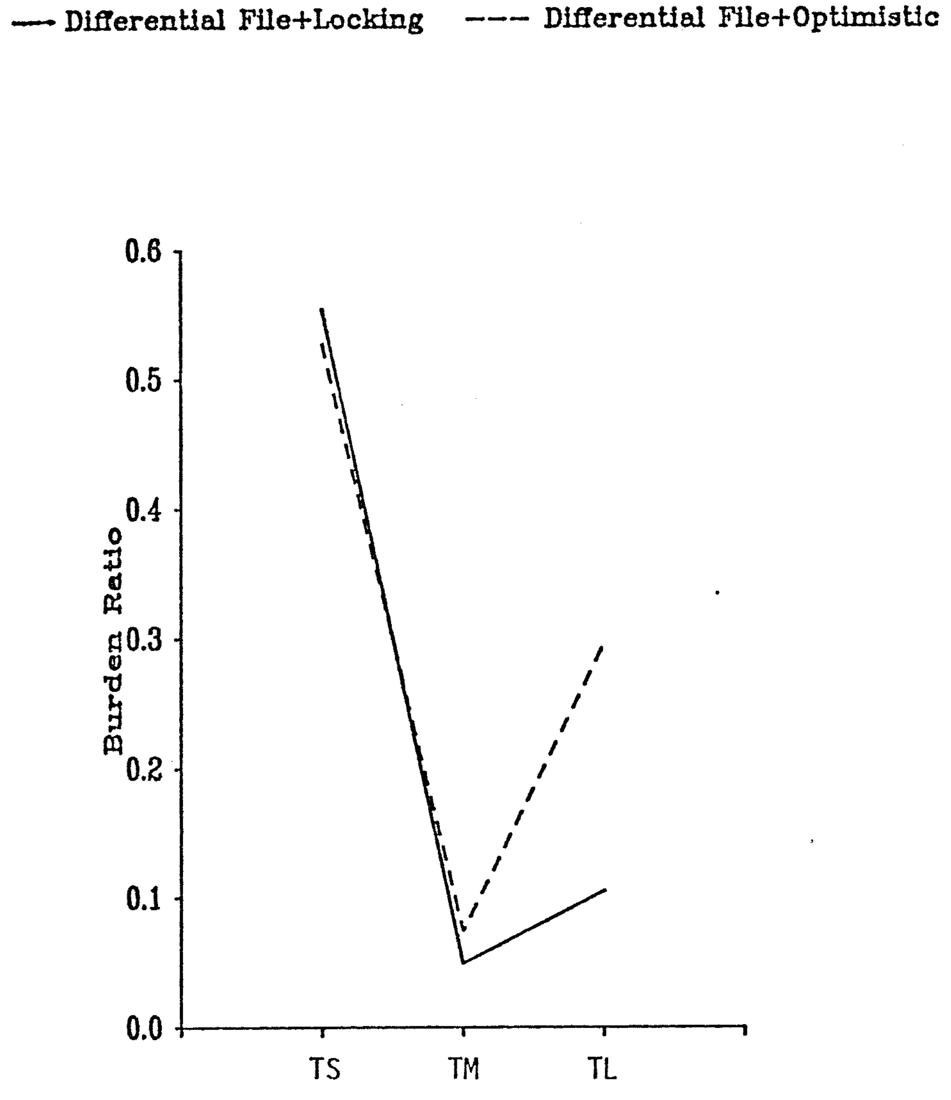

Figure 3. Performance of Differential File+Locking and Differential File+Optimistic

4. For a successful transaction, with an increase in the transaction size the cpu burden becomes a larger fraction of the total burden in the log+locking combination as the cost of transaction waits becomes very significant. For the optimistic method, the validation cost does not increase significantly with the transaction size but the I/O burden increases significantly due to the high cost of making the local copies global.

5. The dip in the burden ratio for TM in Figure 2 is due to the *blocking* effect while writing the log. TS, although it updates only 1 data page, writes 1 log page. On the other hand, TM, although it updates 15 pages, writes only 2 pages[37]. The increased cost of undo processing and waiting increases the burden ratio for TL in the case of locking. In the case of the optimistic method, the increased cost of making local copies global coupled with the high cost of transaction restarts result in a higher burden ratio.

6. Although the total cost of validation is less than the cost of lock management, the log+locking combination outperforms the log+optimistic combination because of the high cost of making local copies global and the the higher restart rate associated with the log+optimistic mechanism. Only in the case of small transactions does the log+optimistic combination perform marginally better. In this case the buffer space available to the transaction, DBuff, is large enough to hold all the pages updated by the transaction until the transaction is validated and hence the cost of making local copies global is not significant. In addition, for small transactions the value of B_{rerun} is quite low.

8.2. Differential Files

The performance of the differential file+locking and differential file+optimistic mechanisms is shown in Figure 3 and Table (ii) in Appendix B. We assumed that Size%, the relative size of the A and D files compared to the B file, equals 10%, that, CpuOH%, the extra cpu overhead equals 100%, (implying that, for example, T_{rec} is 1.0 ms. instead

[37] ceil(0.1*1)=1 and ceil(0.1*15)=2.

of 0.5ms), and that Comprs% equals 0.1 (i.e. a transaction writes ceil(Comprs%*NP_u) pages to both A_l and D_l). One page-sized buffer was allocated for the base file and five each for the A and D files. We make following observations based on Figure 3 and Table (ii):

1. There is a considerable burden in accessing Size% extra data pages and extra CpuOH% processing. However, for the values assumed for Size% and CpuOH%, this burden is more than compensated by the savings that result from not writing the updated data pages as in an in-place updating algorithm.

2. Writing to the A and D files is akin to writing to the log and hence the performance characteristics of the differential file approach appears similar to that of the log approach. In particular, because of blocking effect while writing to the A and D files, TM performs better than TS. The burden ratio for TL becomes higher than for TM because of comparatively less savings in not writing the updated data pages[38]. In addition, the cost of waiting for the lock+differential file mechanism and the cost of transaction restarts in the case of optimistic+differential file mechanism increases considerably from the TM workload to the TL workload.

3. Overall, the differential+locking mechanism performs better than differential+optimistic mechanism for medium and large transactions due to the larger number of transaction restarts with optimistic method. Only for small transactions, where there are not many transaction restarts, does the differential+optimistic method performs marginally better.

8.3. Concurrency Control with Shadows for Recovery

The comparative performance of shadow+locking and shadow+optimistic mechanisms is shown in Figure 4 and Table (iii) in Appendix B. We assumed that DBuff=1 and SBuff=10. Each entry in the page table is assumed to take 4 bytes and thus the size of

[38] For the values assumed, TM updates 30% of the pages read while TL updates only 20% of the pages read.

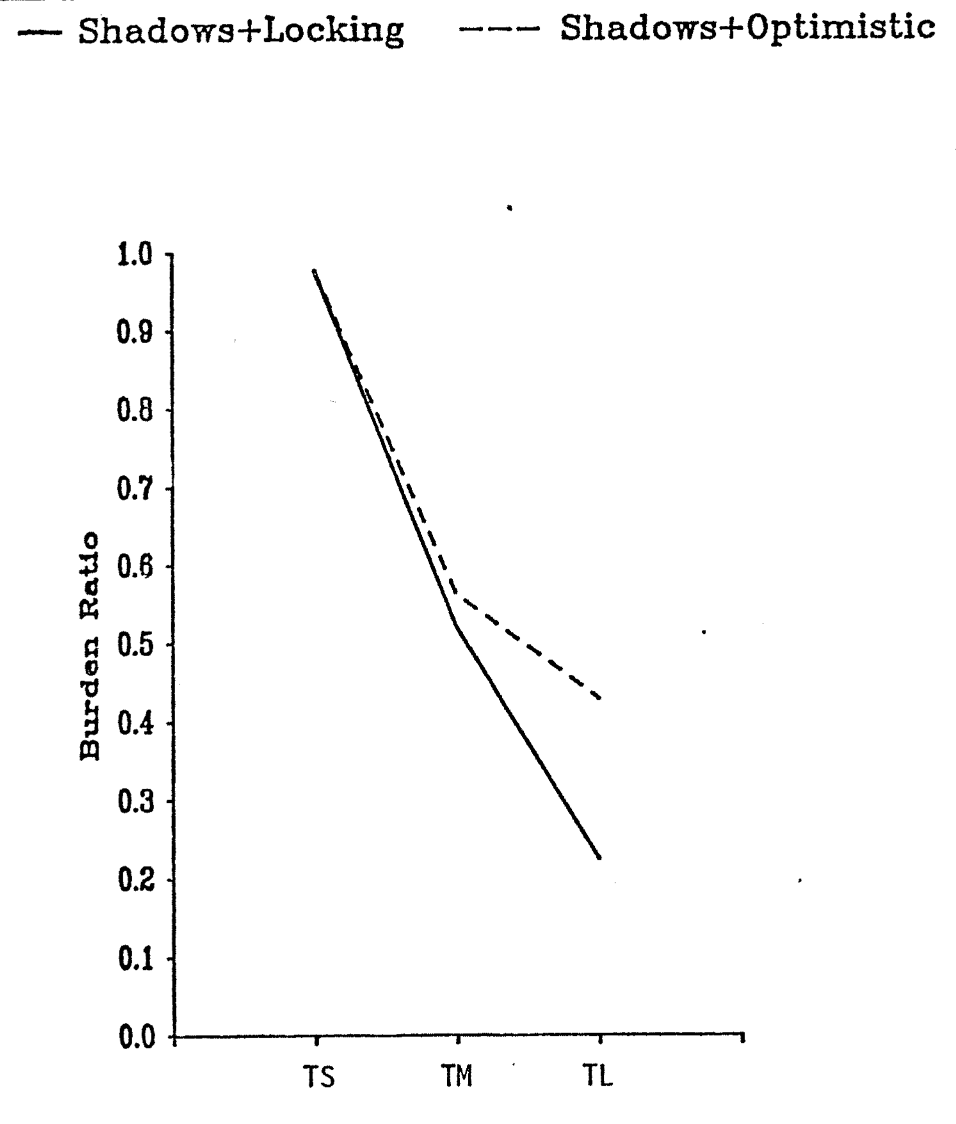

Figure 4. Performance of Shadows+Locking and Shadows+Optimistic (Random)

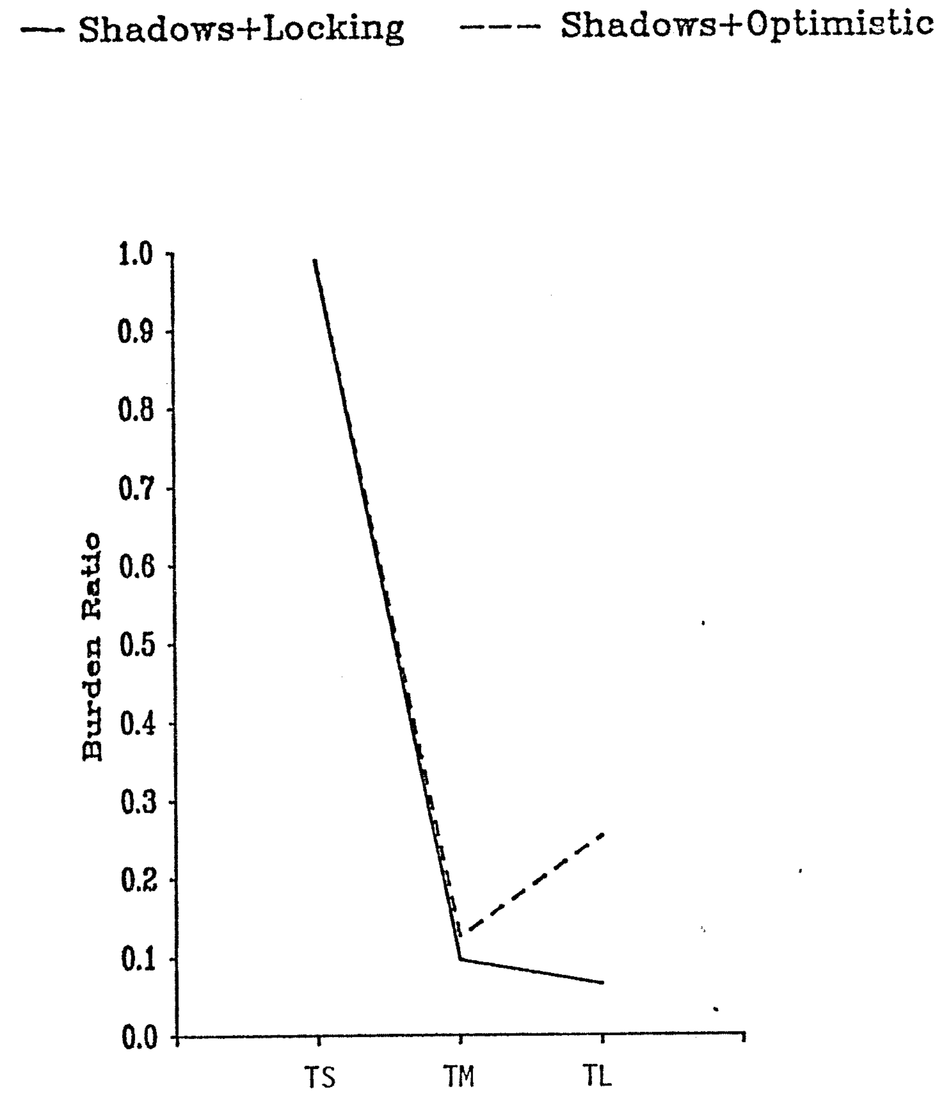

Figure 5. Performance of Shadows+Locking and Shadows+Optimistic (Sequential)

S-Map is 25 pages[39]. We make following observations based on Figure 4 and Table (iii):

1. The cost of reading and updating S-Map (the shadow page table map) constitutes the major portion of total burden.

2. The proportion of the cost of reading S-Map pages reduces with an increase in transaction size since more page-table entries can be found on the same S-Map page (see Table 4). The cost of updating S-Map increases for larger transactions because at the time of updating S-Map, PtPages(NP_u)-SBuff of S-Map pages are reread. However, the reduction in the cost of reading S-Map pages is much higher than the increase in the cost of updating the S-Map. This is reason why in Figure 4, the burden ratio reduces with an increase in transaction size.

3. Overall, the performance of shadow+locking and shadow+optimistic mechanisms are very similar since the cost of reading and updating S-Map (which is the dominant factor in the total burden) is independent of the concurrency control mechanism. For large transactions, the optimistic approach performs somewhat poorer because of the high cost of transaction restarts.

4. We also considered the case of sequential accesses to the database pages for TM and TL workloads. The performance results for this case are shown in Figure 5 and Table (iv). The performance improves considerably because of large reduction in the cost of reading and updating the S-Map. The relative behavior of the locking and

No. of Data Pages Accessed (N)	No. of Page Table Pages Accessed: PtPages(N)
1	1.0
2	1.96
15	14.48
50	21.75
250	25.00

Table 4. No. of accesses to the S-Map

[39] DBSize = 100 million bytes ≃ 25000 pages and No. of S-Map entries per page ≃ 1000.

optimistic methods is similar to that of the random access case.

However, as pointed out in [17], a consequence of using shadows is that logically adjacent pages may not be physically adjacent. Thus, although accesses may be logically sequential, getting the next page may involve disk seek. [33] suggests a page allocation strategy that maintains physical clustering of logically adjacent pages within a cylinder. We have assumed that the shadow mechanism employs such a scheme and we have not assigned any extra cost for potential disk seeks during sequential accesses.

8.4. Some General observations

1. Relatively speaking, deadlock detection is so inexpensive (see the tables in Appendix B) that, in any locking scheme, it should be preferred over deadlock prevention that induces a relatively larger number of transaction aborts.

2. There are many more transaction restarts with the optimistic approach than with the locking approach using deadlock detection. This fact is reflected in the higher values of B_{rerun} in the total burden for the optimistic combination in all three recovery mechanisms. This factor is mainly responsible for the poorer performance of the optimistic combinations for medium and large transactions[40].

3. In the case of the optimistic method, the decision to abort a nonserializable transaction is made only after the transaction has run to completion. In the case of locking, since deadlock detection is performed whenever a lock request conflicts, if a transaction is to be aborted, it will be discovered relatively earlier. Thus, transaction restarts are more expensive with the optimistic approach since the transaction will have run to completion before a conflict is detected. This observation is verified by the tables in Appendix B.

[40] Recently using simulation, Robinson [39] has also found that unless the number of transaction restarts is low, locking uniformly outperforms the optimistic method of concurrency control.

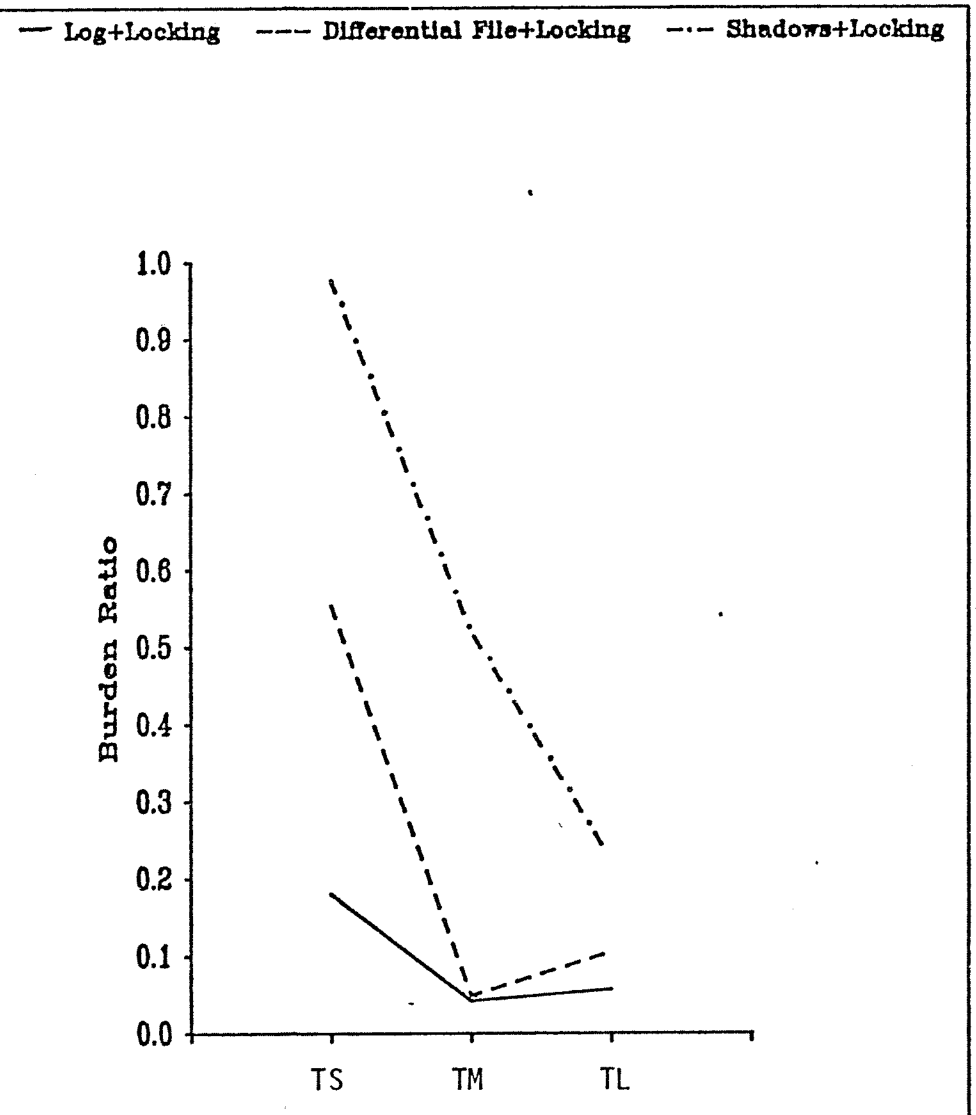

Figure 6. Performance of Integrated Recovery and Concurrency Control Mechanisms

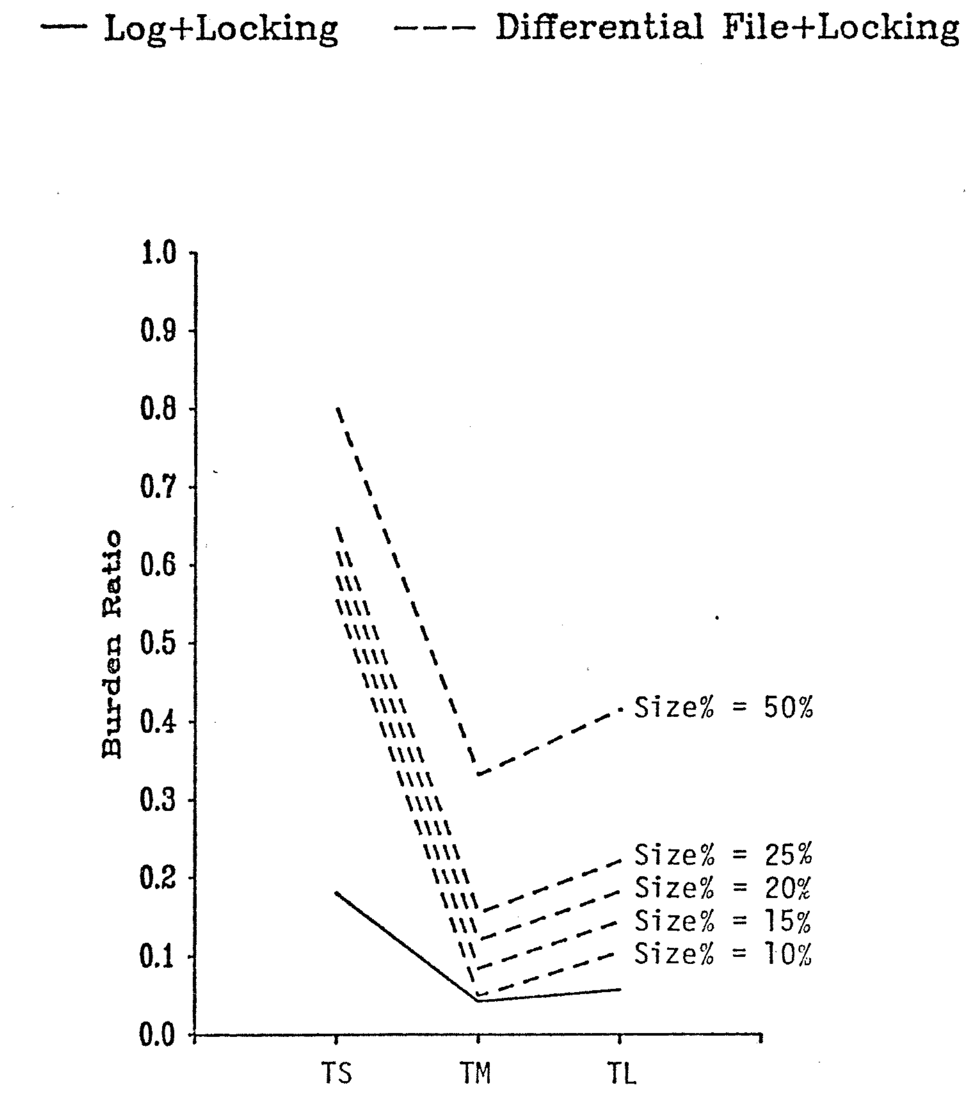

Figure 7. Effect of Size% on the Performance of Differential File+Locking mechanism

8.5. Comparing the Finalists

The relative performance of the log+locking, differential+locking, and shadow+locking mechanisms is shown in Figure 6. We conclude with the following observations:

1. For small transactions, log+locking is the clear winner but for medium and large transactions, differential file+locking also appears promising. As recovery mechanisms, the log and the differential file approach have many similarities. Both do not suffer from the one level of indirection found in the shadow mechanism. The A and D files in the differential file approach are in certain sense after-value and before-value logs. However, in the log approach a transaction, besides writing its log records, also writes to the stable storage the updated data pages at the same time[41]. On the other hand, while the differential file approach must also write pages of the A and D files (that are like log pages) to the stable storage, the actual updating of the data pages in the base relation (that is, merging of pages of the A and D files with the pages of B) can be deferred until a slack time.

The disadvantage of the differential file approach is the cost of accessing Size%*N extra pages in order to access N data pages and the cpu processing overhead of CpuOH%. A sensitivity analysis we have performed indicates the its performance is very critically dependent upon the values of these two factors. Figure 7 and Figure 8 show the performance of the differential+locking mechanism for larger values of Size% and CpuOH% respectively and the performance degrades considerably for larger values of these parameters. We propose to investigate whether it is possible to achieve Size%=10% and CpuOH%=100%. In addition, the assumption that that the differential A and D files can be merged with the main file in slack time is crucial to the performance of this approach.

[41] Writing of updated data pages may not be deferred to some slack time as the associated data buffers may have to be reallocated to another transaction.

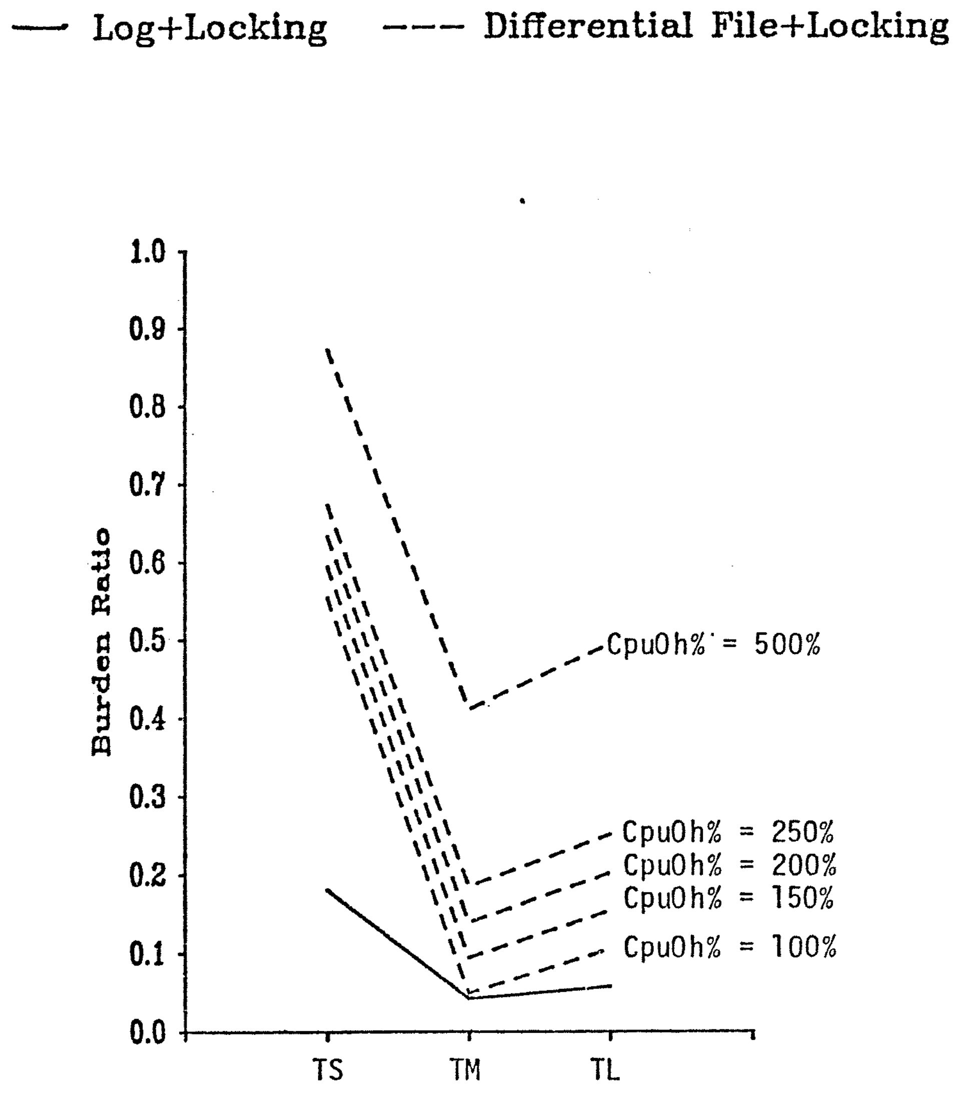

Figure 8. Effect of CpuOh% on the Performance of Differential File+Locking mechanism

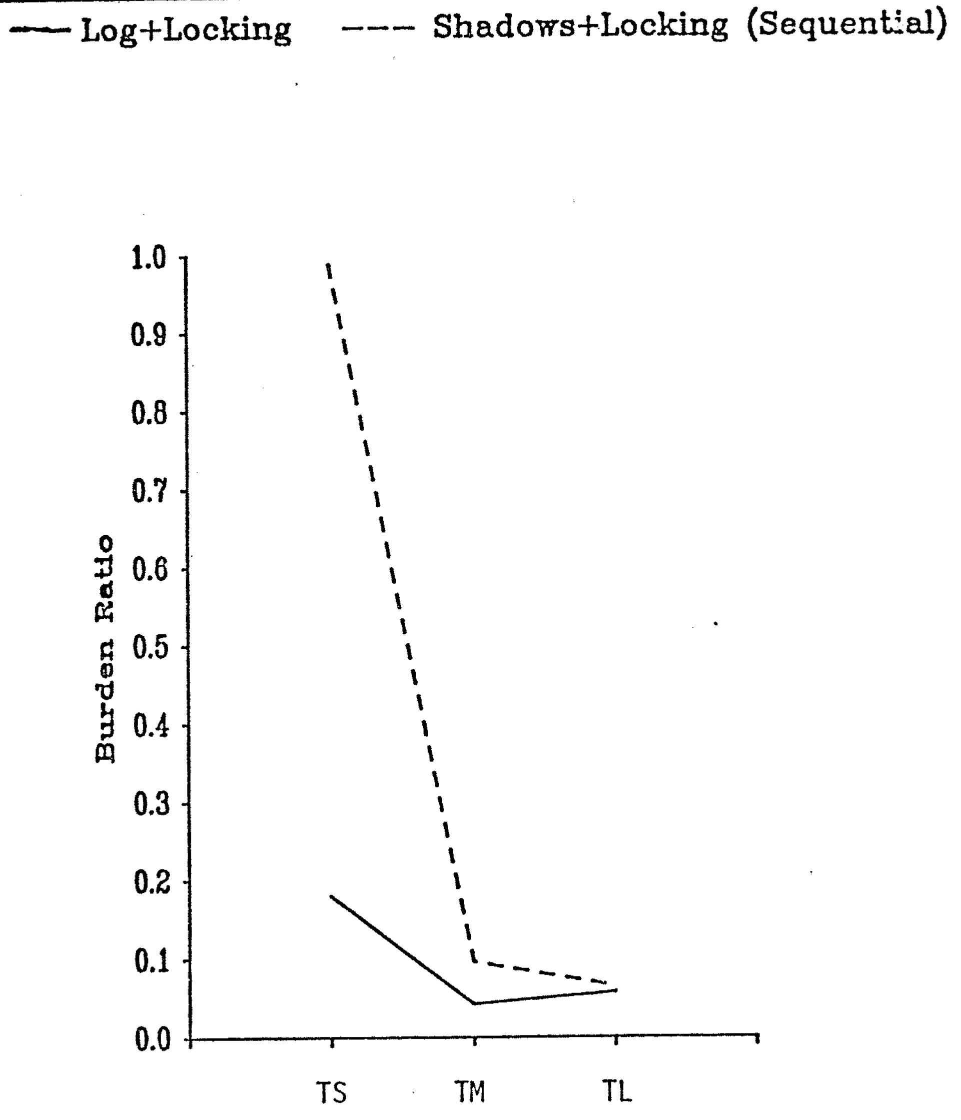

Figure 9. Performance of Log+Locking and Shadows+Locking(Sequential)

2. Figure 9 compares the performance of log+locking with shadow+locking when a sequential access pattern has been assumed for medium and large transactions. Only in case of large transactions, does their performance become comparable.

9. Conclusions

The choice of the "best" integrated concurrency control and recovery mechanism seems to depend on the database environment. If there are only small transactions or there is mix of transactions of varying sizes, log+locking emerges as the most appropriate mechanism. If there are only large transactions with only sequential access pattern, the shadow+locking mechanism is a possible alternative. In an environment of medium and large sized transactions, the differential+locking is a viable alternative to the log+locking mechanism.

The optimistic method of concurrency control should only be considered in an environment where transactions are small with very a low probability of conflict. Even in a low conflict situation, if transactions are large and in-place updating is required, the cost of making local copies global will make the optimistic algorithm an expensive mechanism. Thus, the optimistic method can be attractive only in combination with a recovery mechanism that requires that all updates be collected in some scratch area and applied to the main copy only after a transaction has completed. Thus, recovery and concurrency control mechanisms may share the data structures and the cost of making local copies global.

The major disadvantage of shadows as a recovery mechanism is the cost of indirection through the page table. This mechanism can become attractive only if the page table can always be maintained in the main memory or with an architecture that avoids this indirection.

To summarize, we have presented six integrated mechanisms that perform the tasks of both concurrency control and recovery for centralized database systems. In particular, we have shown what data structures may be shared between the recovery and the concurrency control algorithms in a unified mechanism. We have also extended the shadow and differential file mechanisms for use in a multi-transaction environment. Finally, we have presented a new approach for evaluating the performance of recovery and concurrency control mechanisms. Although the analytical models that we have developed are simple, unlike other approaches that generate one final number for comparison, our approach helps in isolating the costs of various components of a mechanism. Thus in addition to saying that a particular mechanism is expensive, one may answer *why* the mechanism is expensive and *where* efforts should be concentrated to improve the mechanism. We would like to encourage other researchers to use this approach to evaluate other concurrency control and recovery algorithms or our algorithms under a different set of assumptions.

Appendix A

Notation

BX	Total extra cost in running a transaction because of recovery & concurrency control
B_{fail}	Extra cost incurred when a transaction is aborted by the user
B_{rerun}	Extra cost incurred when a transaction is aborted by the system
B_{setup}	Fixed extra cost irrespective of the ultimate fate of the transaction
B_{succ}	Extra cost incurred when a transaction succeeds
DBSize	Size of the database
DBuff	Number of Data buffers allocated to a transaction
DFlush	The function that returns the number of updated data pages that have been flushed to the disc at some time, given the total number of updated pages
Comprs%	The number of differential file pages generated by a transaction is Comprs% of the data pages updated by it.
CpuOH%	With differential files, extra cpu time required to process a transaction is CpuOH% of the cpu time consumed if the transaction was run alone without any provision for recovery
LBuff	Number of buffers available to a transaction to collect log records
Log%	The number of log pages generated by a transaction is log% of the data pages updated by it.
LFlush	The function that returns the number of log pages that have been flushed to the disc at some time, given the total number of log pages
MPL	Level of multiprogramming
NP_t	Total number of pages accessed by a transaction
NP_u	Number of pages updated by a transaction
$P_{conflict}$	Probability that an access request of a transaction would conflict with that of another transaction
P_{ddlk}	Probability that a lock request of a transaction would result in a deadlock

p_{fail}	Probability that a transaction would be aborted by the user
p_{rerun}	Probability that a transaction would be aborted by the system
p_{succ}	Probability that a transaction would complete
p_{wait}	Probability that a lock request of a transaction would be blocked
PtPages	The function that determines the number of page-table pages that would be accessed to access certain number of data pages
SBuff	Number of buffers available to a transaction to get pagetable pages
SFlush	The function that returns the number of page-table pages that are no longer available in the memory, given the total number of page-table pages read by the transaction
Size%	The size of the differential files is Size% of the number of pages in the base file
T_{al}	Time to process a grant-lock request
T_{as}	Time to create control sets (read, write, active etc.) in optimistic method, if $NP_t = 1$
$T_{l\text{-}io}$	Time to read/write a disk page with disk seek
T_{page}	Cpu time to process a page in memory
T_{rec}	Cpu time to process a record in memory
T_{rl}	Time to process a release-lock request
$T_{s\text{-}io}$	Time to read/write a disk page without a seek
T_{valid}	Time to validate a transaction in optimistic method, if there is only one concurrent transaction
T_{wait}	Wait time for a blocked lock request

Appendix B

PERFORMANCE DATA (Unit: milli seconds)

		Total Burden	BSucc component	BFail component	BRerun component	Total I/O	Total cpu	Locking			Set creation	Validation	Make local global
								total	wait	ddlk			
T	Lock	22.1	19.9	2.2	0.0	15.05	7.05	2.0	0.1	0.0			
S	Opt	21.2	20.2	0.95	0.05	13.9	7.3				1.0	1.3	0.0
T	Lock	113.4	109.2	4.1	0.1	28.3	85.1	75.3	26.4	0.2			
M	Opt	385.5	280.0	1.7	103.8	333.4	52.1				25.6	4.4	343.6
T	Lock	715.7	624.5	80.0	11.2	137.2	578.5	549.2	304.1	1.2			
L	Opt	4200.9	1814.6	-22.3	2408.6	3731.4	469.5				149.6	12.7	1568.6

Table (i). log+locking & log+optimistic

		Total Burden	BSucc component	BFail component	BRerun component	Total I/O	Total cpu	Locking			Set creation	Validation	extra size% I/O	extra CpuOh	I/O savings
								total	wait	ddlk					
T	Lock	67.9	67.4	0.5	0.0	55.0	12.9	2.2	0.1	0.0			7.5	9.8	35.7
S	Opt	64.6	64.0	0.5	0.1	51.5	13.1				1.1	1.3	7.4	9.8	35.7
T	Lock	131.4	118.2	13.1	0.1	-197.1	328.5	82.8	29.0	0.2			190.5	243.8	534.7
M	Opt	199.6	64.4	11.6	123.6	-102.4	302.0				28.0	4.5	194.4	255.7	534.7
T	Lock	1311.7	1257.7	43.9	10.1	-517.8	1829.5	604.1	344.5	1.3			970.3	1219.7	1808.7
L	Opt	3713.6	808.0	31.9	2873.7	1758.6	1955.0				163.2	14.1	1137.6	1495.9	1808.7

Table (ii). differential+locking & differential+optimistic

		Total Burden	BSucc component	BFail component	BRerun component	Total I/O	Total cpu	Locking			Set creation	Validation	Smap read	Smap update
								total	wait	ddlk				
T	Lock	119.8	117.8	2.0	0.0	115.8	4.0	2.0	0.1	0.0			71.8	36.1
S	Opt	119.7	117.7	1.9	0.1	115.9	3.8				1.0	1.3	71.8	36.1
T	Lock	1397.8	1365.1	32.5	0.2	1293.4	104.4	75.3	26.4	0.2			805.5	486.5
M	Opt	1511.0	1312.7	31.0	167.3	1446.7	64.3				25.6	4.4	844.4	486.5
T	Lock	2806.4	2732.6	62.6	11.2	2159.3	647.1	549.2	304.1	1.2			939.2	1227.0
L	Opt	5367.8	2304.3	51.0	3012.5	4844.8	523.0				149.6	12.7	1145.8	1227.0

Table (iii). shadow+locking & shadow+optimistic (random)

		Total Burden	BSucc component	BFail component	BRerun component	Total I/O	Total cpu	Locking			Set creation	Validation	Smap read	Smap update
								total	wait	ddlk				
T	Lock	119.8	117.8	2.0	0.0	115.8	4.0	2.0	0.1	0.0			71.8	36.1
S	Opt	119.7	117.7	1.9	0.1	115.9	3.8				1.0	1.3	71.8	36.1
T	Lock	259.3	252.9	6.3	0.1	154.9	104.4	75.3	26.4	0.2			75.1	78.4
M	Opt	337.1	200.4	4.7	132.0	272.8	64.3				25.6	4.4	78.7	78.4
T	Lock	810.4	780.8	19.7	9.9	163.3	647.1	549.2	304.1	1.2			75.2	95.1
L	Opt	3181.8	352.5	8.1	2821.2	2658.8	523.0				149.6	12.7	91.7	95.1

Table (iv). shadow+locking & shadow+optimistic (sequential)

REFERENCES

[1] R. Agrawal, M. Carey, and D.J. DeWitt, "Deadlock Detection Is Cheap," Electronics Research Lab. Mem. No. UCB/ERL M83/5, Univ. California, Berkeley (Jan. 1983).

[2] P.A. Alsberg, G.G. Belford, J.D. Day, and E. Grapa, "Multi-Copy Resiliency Techniques," R Center for Advanced Computation Doc. 202, Univ. Illinois, Urbana-Champaign, Illinois (May 1976).

[3] P.A. Bernstein, "The Concurrency Control Mechanism of SDD-1: A System for Distributed Databases (The General Case)," Tech. Rep. CCA-77-09, Computer Corp. America, Cambridge, Massachusetts (Dec. 1977).

[4] P.A. Bernstein and N. Goodman, "Concurrency Control in Distributed Database Systems," *ACM Computing Surveys 13*, 2, pp. 185-221 (June 1981).

[5] P.A. Bernstein and N. Goodman, "A Sophisticate's Introduction to Distributed Database Concurrency Control," *Proc. 8th Int'l Conf on Very Large Data Bases*, (Sept. 1982).

[6] P.A. Bernstein, D.W. Shipman, and W.S. Wong, "Formal Aspects of Serializability in Database Concurrency Control," *IEEE Trans. Software Eng. SE-5*, 3, (May 1979).

[7] A.F. Cardenas, "Analysis and Performance of Inverted Database Structures," *Commun. ACM 18*, 5, pp. 253-263 (May 1975).

[8] K.M. Chandy, "A Survey of Analytic Models of Rollback and Recovery Strategies," *IEEE Computer 8*, 5, pp. 40-47 (May 1975).

[9] E.G. Coffman, M.J. Elphic, and A. Shoshani, "System deadlocks," *ACM Computing Surveys 3*, 2, pp. 67-78 (June 1971).

[10] K.P. Eswaran, J.N. Gray, R.A. Lorie, and I.L. Traiger, "The Notions of Consistency and Predicate Locks in a Database System," *Commun. ACM 19*, 11, pp. 624-633 (Nov. 1976).

[11] H. Garcia-Molina, "Performance of Update Algorithms for Replicated Data in a Distributed Database," Stan-CS-79-744, Computer Sciences Dept., Stanford Univ. (June 1979) Ph.D. Dissertation.

[12] E. Gelenbe and D. Derochette, "Performance of Rollback Recovery Systems Under Intermittent Failures," *Commun. ACM 21*, 6, pp. 493-499 (June 1978).

[13] E. Gelenbe, "On the Optimum Checkpoint Interval," *J. ACM 26*, 2, pp. 259-270 (April 1979).

[14] J.N. Gray, "A Discussion of Distributed Systems," Invited Lecture at the Congresso Annuale of Associazione Italiana per il Calcolo Automatico, Bari, Italy (Aug. 1979).

[15] J.N. Gray, P. Homan, H. Korth, and R. Obermarck, "A Straw Man Analysis of the Probability of Waiting and Deadlock in a Database System," Rep. RJ3066, IBM Research Lab., San Jose, California (Feb. 1981).

[16] J.N. Gray, R.A. Lorie, G.F. Putzolu, and I.L. Traiger, "Granularity of Locks and Degrees of Consistency in a Shared Database," pp. 365-394 in *Modelling in Data Base Management Systems*, ed. G.M. Nijssen,North-Holland, Amsterdam (1976).

[17] J.N. Gray, P.R. McJones, B.G. Lindsay, M.W. Blasgen, R.A. Lorie, T.G. Price, F Putzolu, and I.L. Traiger, "The Recovery Manager of the System R Database Manager," *ACM Computing Surveys 13*, 2, pp. 223-242 (June 1981).

[18] J.N. Gray, "Notes on Database Operating Systems," in *Lecture Notes in Computer Science 60, Advanced Course on Operating Systems*, ed. G. Seegmuller,Springer Verlag, New York (1978).

[19] J.N. Gray, Personal communication to D.J. DeWitt. (April 1982).

[20] J.N. Gray, "The Transaction Concept: Virtues and Limitations," *Proc. 7th Int'l Conf. on Very Large Data Bases*, pp. 144-154 (Sept. 1981).

[21] J.N. Gray, "A Transaction Model," pp. 282-298 in *Lecture Notes in Computer Science 85, Automata, Languages and Programming*, ed. J. van Leeuwen,Springer Verlag, New York (1980).

[22] P.B. Hansen, *Operating System Principles*, Prentice-Hall, Englewood Cliffs, N.J. (1973).

[23] R.C. Holt, "Some Deadlock Properties of Computer Systems," *ACM Computing Surveys 4*, 3, pp. 179-196 (Sept. 1972).

[24] IBM, "Reference Manual for IBM 3350 Direct Access Storage," GA26-1638-2, File No. S370-07, IBM General Products Division, San Jose, California (April 1977).

[25] K.B. Irani and H.L. Lin, "Queueing Network Models for Concurrent Transaction Processing in a Database System," *Proc. ACM-SIGMOD 1979 Int'l Conf. on Management of Data*, pp. 134-142 (May 1979).

[26] W.H. Kohler, "A Survey of Techniques for Synchronization and Recovery in Decentralized Computer Systems," *ACM Computing Surveys 13*, 2, pp. 149-183 (June 1981).

[27] H.T. Kung and J.T. Robinson, "On Optimistic Methods for Concurrency Control," *ACM Trans. Database Syst. 6*, 2, pp. 213-226 (June 1981).

[28] B. Lampson and H. Sturgis, "Crash Recovery in a Distributed Data Storage System," Computer Science Lab., Xerox PARC (1979) to appear in Commun. ACM.

[29] W.T.K. Lin, "Concurrency Control in a Multiple Copy Distributed Data Base System," *Proc. 4th Berkeley Workshop on Distributed Data Management and Computer Networks*, (Aug. 1979).

[30] W.T.K. Lin and J. Nolte, "Basic Timestamp, Multiple Version Timestamp, and Two Phase Locking," Computer Corp. America, Cambridge, Massachusetts (Jan. 1983).

[31] W.T.K. Lin and J. Nolte, "Performance of Two Phase Locking," *Proc. 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pp. 131-160 (Feb. 1982).

[32] W.T.K. Lin, "Performance Evaluation of Two Concurrency Control Mechanisms in a Distributed DBMS," *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, pp. 84-92 (April 1981).

[33] R.A. Lorie, "Physical Integrity in a Large Segmented Database," *ACM Trans. Database Syst. 2*, 1, pp. 91-104 (March 1977).

[34] R. Munz and G. Krenz, "Concurrency in Database Systems - A Simulation Study," *Proc. ACM-SIGMOD 1977 Int'l Conf. on Management of Data*, pp. 111-120 (Aug. 1977).

[35] C.H. Papadimitriou, "Serializability of Concurrent Updates," *J. ACM 26*, 4, pp. 631-653 (Oct. 1979).

[36] D. Potier and Ph. Leblanc, "Analysis of Locking Policies in Data Base Management Systems," *Commun. ACM 23*, 10, pp. 584-593 (Oct. 1980).

[37] D.P. Reed, "Naming and Synchronization in a Decentralized Computer System," Lab. for Computer Science MIT/LCS/TR-205, Massachusetts Institute of Technology, Cambridge, Massachusetts (Sept. 1978) Ph.D. Dissertation.

[38] D.R. Ries, *The Effects of Concurrency Control on Database Management System Performance*, Computer Sciences Dept., Univ. California, Berkeley (April 1979) Ph.D. Dissertation.

[39] J.T. Robinson, "Design of Concurrency Controls for Transaction Processing Systems," CMU-CS-82-114, Computer Science Dept., Carnegie-Mellon Univ., Pittsburgh, Pennsylvania (April 1982) Ph.D. Dissertation.

[40] D.J. Rosenkrantz, R.E. Stearns, and P.M. Lewis, "System Level Concurrency Control for Distributed Database Systems," *ACM Trans. Database Syst. 3*, 2, pp. 178-198 (June 1978).

[41] D.G. Severance and G.M. Lohman, "Differential Files: Their Application to the Maintenance of Large Databases," *ACM Trans. Database Syst. 1*, 3, pp. 256-267 (Sept. 1976).

[42] J.F. Spitzer, "Performance Prototyping of Data Management Applications," *Proc. ACM 76 Annual Conf.*, pp. 287-297 (Oct. 1976).

[43] M.R. Stonebraker, "Hypothetical Data Bases as Views," *Proc. ACM-SIGMOD 1981 Int'l Conf. on Management of Data*, pp. 224-229 (May 1981).

[44] M.R. Stonebraker, "Implementation of Integrity Constraints and Views by Query Modification," *Proc. ACM-SIGMOD 1975 Int'l Conf. on Management of Data*, pp. 65-78 (June 1975).

[45] M.R. Stonebraker and K. Keller, "Embedding Expert Knowledge and Hypothetical Data Bases into a Data Base System," *Proc. ACM-SIGMOD 1980 Int'l Conf. on Management of Data*, pp. 58-66 (May 1980).

[46] L. Svobodova, "A Reliable Object-Oriented Data Repository for a Distributed Computer System," *Proc. ACM-SIGOPS 8th Symp. on Operating Systems Principles*, pp. 47-58 (Dec. 1981).

[47] R.H. Thomas, "A Solution to the Update Problem for Multiple Copy Databases Which Uses Distributed Control," BBN Rep. 3340, Bolt, Beranek and Newman Inc. (July 1978).

[48] J.S.M. Verhofstad, "Recovery Techniques for Database Systems," *ACM Computing Surveys 10*, 2, pp. 167-195 (June 1978).

[49] S.B. Yao, "Approximating Block Accesses in Database Organizations," *Commun. ACM 20*, 4, pp. 260-261 (April 1977).