A QUANTITATIVE COMPARISON OF LOCKPROTOCOLS FOR CENTRALIZED DATABASES

W. Kiessling, G. Landherr

Institute of Informatics, Technical University Munich
Arcisstrasse 21, D-8000 München 2, West-Germany

Abstract: To process transactions of several users concurrently and consistently in a shared database various lockprotocols have been developed recently. Though the need for a quantitative analysis of lockprotocols is rather obvious, this work has not yet been fully performed for centralized database systems. As database applications in the near future tend to have very high transaction rates, such quantitative investigation on the quality of lockprotocols will become increasingly valuable.

The paper evaluates three well-know lockprotocols by discrete-event simulation. We represent and discuss the results which we have gained in the simulation runs.

Contents

1. Introduction

To process transactions of several users concurrently and consistently in a shared database various lockprotocols have been developed recently. These lockprotocols hopefully increase the systems's efficiency by providing a high degree of concurrency among executing transactions. However, in raising the potential concurrency we also increase the involved synchronization effort to enforce the underlying lockprotocol. Therefore it is essential to make a tradeoff between potential efficiency gains by increased concurrency and the involved synchronization overhead when selecting a particular lockprotocol. In this paper we analyse three lockprotocols for centralized databases as presented in [EGLT 76], [BAY 76] and [BHR 80]. Though the need for a quantitative analysis of lockprotocols is rather obvious, this work has mostly been done only for distributed databases, where different optimization criteria are applied (see e.g. [BAD 80]).

For database applications in the near future high-performance database management systems are required, providing a transaction throughput in the order of about 100 transactions per second (as opposed to 1-15 transactions/sec for existent systems). Thus quantitative investigations concerning the quality of lockprotocols will become increasingly valuable also for centralized databases.

Most analysis techniques for distributed lockprotocols presented in the literature rely on analytical models. But as the subject of concurrency evaluation is rather complicated, often many simplifying assumptions are stated in order to get a still analytically tractable model. However, as we do not want to restrict our investigations by assumptions which are too restrictive, like preclaiming of locks ([PL 80], [MN 82]), we developed a discrete-event simulation model, which is more flexible and will presumably produce more reliable results.

Terms:

We consider a database consisting of a set of objects. The DB-system provides operations each of

which manipulates one or more objects. The execution of an operation on an object is called an action. Actions are uninterpreted, we only distinguish read-access and write-access.

In order to efficiently supervise the correctness of data, which are stored in a shared database and which are subject to instantaneous changes, the logical concept of a transaction has been developed.

A transaction [1] T is a finite sequence of actions, exhibiting an atomic behavior (often also called 'all-or-nothing' property): Either all actions of T are executed or T has no effect at all on the database state. In the first case, T is said to be successfully committed, in the later one, T is said to be backed out. In many practical applications efficiency requirements, such as high transaction rates and short response time, enforce that a set of transactions is processed cocurrently on a shared database.

Consistency deals with the correct processing of concurrent transactions. Various stages of correct concurrent behavior, termed consistency levels 0, 1, 2 and 3, have been identified in [GLPT 76]. In this paper we only deal with degree 3 consistency, sometimes also called serializability.

The concurrent execution of a set $\{T_1,T_2,\ldots,T_n\}$ of transactions is serializable, if $T_1,T_2,\ldots,T_n$ produces the same effect on the database state as some serial execution $\{T_{p(1)},T_{p(2)},\ldots,T_{p(n)}\}$, where $p: \{1,2,\ldots,n\} \rightarrow \{1,2,\ldots,n\}$ is a permutation.

To achieve consistency, the interleaved execution of concurrent transactions must be properly synchronized, i.e. the particular actions of these concurrent transactions must be scheduled in a way that guarantees the serializability-criterion. The usual method to achieve this is locking. A set of rules for requesting and releasing locks defines a lockprotocol.

In the sequel the term consistency will be used as a synonym for serializability. A transaction is termed well-formed, if it requests an appropriate lock on an object before it performs the intended action on this object.

In [EGLT 76] the notion of two phase transactions is introduced. A transaction is two phase, if it can be divided into a growing and a shrinking phase. During the growing phase the transaction may request locks but must not release locks. Inversely, during the shrinking phase it may release locks but not request locks. Well-formed two phase transactions are shown to preserve consistency.

If a transaction T_1 has to be backed out, it might be necessary to back out another transaction T_2, too, since T_2 has read a value of an object which has been generated by T_1. A cascade of backups may become necessary, being especially troublesome for already committed transactions. We avoid cascading by allowing only strict two phase transactions.

A transaction is strict two phase, if it is two phase and in addition its shrinking phase comprises only one indivisible action, i.e. the transaction must hold all its locks until its very end. Then all its locks are released with a single atomic action. Strict two phase transactions allow the isolated backup of an uncommitted transaction. This paper deals with three well-known lockprotocols for centralized database systems.[2] They all employ the concept of well-formed and strict two phase transactions.

2. Description of three Well-known Lockprotocols for a Centralized DBMS: (r,x)-, (r,a,x)- and (r,a,c)-Protocol

For concurrency control by locking, three basic lockprotocols are known in a centralized database:

(a) The (r,x)-protocol, presented in [EGLT 76].

(b) The (r,a,x)-protocol, presented in [BAY 76].

(c) The (r,a,c)-protocol, presented in [BHR 80].

In this paper we will give an overview about these synchronization methods; for a more precise description the interested reader is referred to the above stated references.

2.1. Lockmodes and Compatibilities

The (r,x)-protocol

This protocol disposes of two lockmodes which can be requested on an object, namely:

- the r-lock, if read access is intended
- the x-lock, if write access is intended.

The r-lock on a particular object protects it from concurrent updates, several r-locks may coexist on it. The x-lock provides exclusive access to an object. Thus, if a transaction T issues an x-lock request for an object which is already r-locked, this request has to be denied and T must wait. Likewise, r-locks and x-locks on an already x-locked object must not be granted and have to be deferred. From this semantics of r-locks and x-locks we can derive compatibilities between lockmodes.

Two lockmodes are said to be compatible, if they can coexist on the same object, otherwise they are incompatible.

---

[1] A formal model is presented in [GRA 80].

[2] The authors somewhat arbitrarily decided on this restricted set of locking protocols for the basic investigations. Our planned future work will cover a broader class of suggested concurrency control methods (see chapter 5).

	r	x
r	+	-
x	-	-

+ means compatible
- means incompatible

Fig. 1. Compatibility matrix for the (r,x)-protocol

This last protocol forms the starting point for the other two protocols, which both aim to reduce the inhibitions among concurrent transactions, originating from wait situations due to incompatible lock requests. In the (r,x)-protocol there are two transaction-states. A transaction is blocked, if it requests an incompatible lock. A transaction which is not blocked is called active.

The (r,a,x)-protocol

This protocol was encouraged by the following considerations concerning the processing of an isolated transaction backup: Since the effects of uncommitted transactions must be erasable in the event of failure, the following strategy obviously improves the efficiency of a backup procedure:

The new value of an object is prepared on a copy, leaving the valid value unaffected; this valid value is kept as a shadow for recovery purposes. Furthermore, read access to the shadow can be granted to concurrent transactions while the preparation of a new value on a copy is in progress.

The (r,a,x)-protocol employs the following lockmodes:

- the r-lock, if read access is intended
- the a-lock, if the preparation of a new value is intended
- the x-lock, if the prepared new value is ready for commit.

The compatibilities among these lockmodes are defined below:

	r	a	x
r	+	+	-
a	+	-	-
x	-	-	-

Fig. 2: Compatibility matrix for (r,a,x)-protocol

Only at the end of a transaction (EOT), when a transaction wants to commit all prepared new object values in one atomic action, read accesses to the respective shadow values are prohibited. This is achieved as follows:

Assume, transaction T holds a-locks on objects $O_1,O_2,\ldots,O_k$ and wants to get committed at EOT. When all r-locks on $O_1,O_2,\ldots,O_k$ from other transactions have been released, the a-locks of T on $O_1,O_2,\ldots,O_k$ are converted into x-locks. This conversion can cause certain difficulties which will be discussed later (chapter 2.2). Thereafter, in accordance to the strict two phase property, all locks held by T are atomically released, thereby committing T successfully. Additional to the transaction-states, blocked and active, a transaction T can also accept the state inactive in the (r,a,x)-protocol. Transaction T is inactive, if it is in its conversion phase, i.e. from the moment transaction T wants to get converted until all locks of T are released.

The (r,a,c)-protocol

This protocol succeeded in diminishing the delay caused by locking even further. It makes use of the fact that for recovery reason there are temporarily two versions of one object. The basic observation concerns the conversion procedure from a-locks to x-locks of the (r,a,x)-protocol, which delays this process although the new prepared values are readily available. The (r,a,c)-protocol handles the ending of an updating transaction T in the following way at EOT:

- All a-locks of T are converted into c-locks; a c-locked object now has two valid values, the shadow value and the new value. This conversion step is the actual commit point of a transaction (for complications introduced by this conversion step see chapter 2.2).
- The c-locks of T are released when the shadows are no longer needed for a consistent scheduling of concurrent transactions.

In [BHR 80] a method is presented to decide which value of a c-locked object has to be provided for read accesses in order to preserve consistency.

In summary, the lockmodes of the (r,a,c)-protocol are:

- the r-lock, if read access is desired
- the a-lock, if the preparation of a new value is intended
- the c-lock, if the prepared new value is committed; however, the shadow is still available for read accesses.

The main advantage of this (r,a,c)-protocol is the fact that read requests are never blocked by a concurrent update transaction, as can be seen from lock compatibilities. Similar to the (r,a,x)-protocol the (r,a,c)-protocol has also three different states of a transaction T. If T requests an incompatible lock it is blocked until the lock can be granted. If T has successfully converted its a-locks into c-locks, but not yet released its c-locks, it is called inactive.

Transactions which are neither blocked nor inactive are called active.

	r	a	c
r	+	+	+
a	+	-	-
c	+	-	-

Fig. 3: Compatibility matrix for the (r,a,c)-protocol

2.2. Specification of the Lockmanagers

In a well-structured DBMS there exists a special module called the lockmanager (LM) which is responsible for correctly executing the concurrency control method. For each actually locked object the LM maintains information describing the lock state of the particular objects; this lock information can be managed in a dynamic fashion. The interface operations of the particular LM correspond to the allowed lockrequests of the employed lockprotocol; the semantics of these lock/convert/unlock-operations is specified by the lock compatibilities. In addition, in a database system where locks can be requested dynamically (i.e. we don't restrict concurrency control to preclaiming of locks at transaction begin), a LM must be prepared for handling the well-known phenomena of deadlock and starvation.

The lockmanager $LM_{\{r,x\}}$:

The (r,x)-lockprotocol introduces wait relationships among concurrent transactions, issuing incompatible lockrequests for the same object. These wait relationships are retained in a directed graph G=(W,U) where

$$W = \{T_i \mid T_i \text{ is a transaction in the DB-system } (i=1,2,\ldots,n)\};$$

$$U = \{(T_i,T_j) \mid T_i \text{ waits for } T_j \ (i,j=1,2,\ldots,n;\ i\neq j)\};$$

This graph G(W,U), called the wait graph, is dynamically maintained by the $LM_{\{r,x\}}$, when transactions are blocked due to denied lockrequests or activated due to the granting of a so far denied lockrequest. In the former case the $LM_{\{r,x\}}$ has to check whether a deadlock situation has occurred (i.e. we test for deadlocks in a dynamic fashion and not periodically). Deadlocks are recognized by a cycle in the wait graph. If such a deadlock situation is detected, this deadlock must be broken by backing out one or several transactions; the victim transactions have to be restarted. This backup and restart procedure in the event of deadlocks is subject to another phenomenon known as livelock in which two or more transactions repeatedly cause each other to be backed out and restarted.[3)]

[3)] In our simulation livelocks actually occurred. Strategies for resolving livelocks are outlined in chapter 3.1.

Though the $LM_{\{r,x\}}$ recovers from deadlock, and handles livelock, too, it is possible that a transaction T is prevented from ever finishing. This situation may happen,if an x-lock request of T is blocked on an object O and there are always r-lock requests for O from different transactions granted. This is the well-known problem of starvation.

To avoid starvation for the (r,x)-protocol, the $LM_{\{r,x\}}$ pursues the following strategy:

With each locked object the $LM_{\{r,x\}}$ associates a waiting queue of pending (i.e. waiting or not yet served) lockrequests. This queue is served in a first-in-first-out manner. Thus, in case an x-lock request of T is blocked on an object O and T' issues an r-lock request on O subsequently, this lockrequest is not granted immediately, instead it is queued behind the x-lock request of T (and the wait graph is maintained accordingly). This strategy ensures that starvation cannot occur, however, this has been achieved only by restricting the compatibilities as specified in Fig. 1. A strategy for selecting a backup victim to resolve a deadlock will be given in a summary later, because this strategy is identical for all three lockmanagers $LM_{\{r,a,x\}}$ and $LM_{\{r,a,c\}}$, in order to make things comparable.

The lockmanager $LM_{\{r,a,x\}}$:

The (r,a,x)-protocol likewise introduces wait relationships among conflicting transactions.

These relationships are retained again in a (extended) wait graph G(W,U). The manipulation of G(W,U) by the $LM_{\{r,a,x\}}$ is as follows:

- Wait relationships due to blocked r-lock or a-lock requests are inserted and removed as usual.
- Within the conversion phase:
  Suppose, T holds an a-lock on O and wants to convert this lock into an x-lock, further there exist r-locks of $T_1,T_2,\ldots,T_k$ on O. Then the conversion can only be granted when $T_1,T_2,\ldots,T_k$ have released their r-locks on O. Therefore wait relationships from T to $T_1,T_2,\ldots,T_k$ must be inserted into G(W,U). If this leads to a cycle in G(W,U), then backup victims have to be selected.

The problem of starvation cannot arise when a-locks are requested as can be seen from the compatibility matrix in Fig. 2. However, the problem has not vanished at all, instead it has been shifted to the conversion of a-locks into x-locks. Our $LM_{\{r,a,x\}}$ employs the following strategy for coping with starvation of conversion for a transaction T:

Having inserted the respective wait relationships and having tested that no cycle exists in G(W,U), we delay further r-lock requests on objects a-

locked by transaction T.

This delay can conceptionally be achieved by placing a "delay mark" on the respective objects. Practically the same effect is gained by immediately converting T's a-locks into x-locks, thereby slightly changing the compatibilities as specified in Fig. 2. The x-locks are finally released when all r-locks on the respective shadow values have disappeared.

It should be recalled that we perform the conversion of all a-locks held by a transaction T at the very end of T in a single step. Theoretically, a-locks could also be converted separately into x-locks without violating consistency. However, our solution has the advantage that the $LM_{\{r,a,x\}}$ has to execute the costly cycle detection algorithm only once due to conversion of T.

On the other hand inconsistencies are detected at the latest possible moment.

The lockmanager $LM_{\{r,a,c\}}$.

The synchronization requirements of the (r,a,c)-protocol necessitate the $LM_{\{r,a,c\}}$ again to maintain a graph, which is now called a dependence graph. Arcs in this graph represent wait relationships defined as usual or the so-called follow relationships being defined as:

T_i follows T_j if there is an object O such that T_i reads the new value of O while T_j holds a c-lock on O or T_i holds a c-lock on O while T_j reads the shadow value of O.

For a precise description of the maintenance of this dependence graph by the $LM_{\{r,a,c\}}$ the reader is referred to [BHR 80].

However it remains to emphasize two facts:

- The conversion of a-locks into c-locks is done as last active action at the very end of a transaction, only one cycle detection process is performed. This means again that inconsistencies are detected at the latest possible moment (just before transaction commit).
- Starvation of conversion is not possible due to the lock compatibilities (it either fails or can be done immediately).

Finally we want to summarize the requirements which we impose on our three lockmanagers:

(1) Each lockmanager reacts in the described way when a transaction T requests a lock or announces its end (transactions are strict two phase).

(2) The respective graph is maintained in the correct way. The method of selecting a backup victim to break a cycle in the graph is chosen identically for all three protocols: Simply backup and restart that transaction which caused the cycle. Strategies for coping with livelock are discussed in chapter 3.1.

3. The Simulation Model

3.1. The Transaction Processing Model

The processing of transactions essentially is simulated by two functional modules, the transaction manager (TM) and the lockmanager (LM).

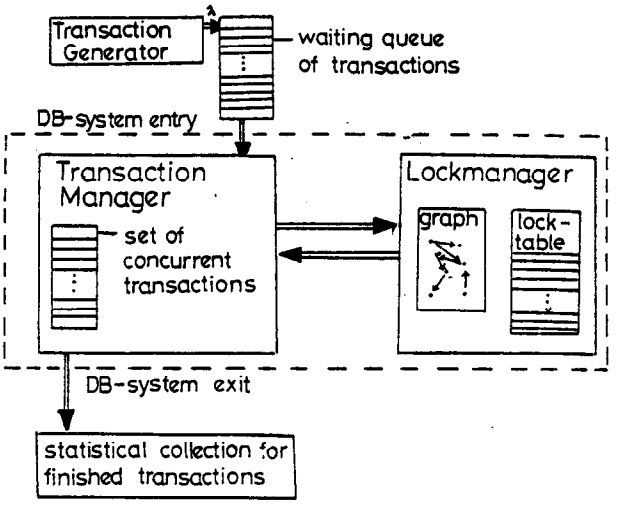

Fig. 3.1.: The transaction processing model

The actual LM is one of the previously described lockmanagers $LM_{\{r,x\}}$, $LM_{\{r,a,x\}}$, $LM_{\{r,a,c\}}$ depending on the employed lockprotocol. In fact, the LMs are not simulated, instead they are implemented in their full functionality. The TM is identical for all three lockprotocols.

The task of the transaction manager TM is as follows:
The TM supervises the execution of transactions which work concurrently in the database system. The maximum concurrency-degree is limited by the number n_{max}, i.e. at most n_{max} transactions are allowed to be in the database system at a time; of course n_{max} can be varied in different simulation runs. The TM is in charge of the coordination of executing actions of concurrent transactions. Therefore, it retains also the states (active, blocked, inactive) of the particular transactions. To enable the execution of an action, the TM sends the appropriate lockrequest, which is required by the employed lockprotocol, to the LM. The LM decides this lockrequest in the specified way and returns an answer to the TM, whether this request is granted (then the transaction T in question remains active) or T is blocked or backed up.

For well-formed transactions a granted lock is the prerequisite for a read or write operation on the locked object. However, as our intention is to investigate the potential concurrency inherent in the particular lockprotocols, we do not

model those parts of the DB-system (access system, storage subsystem, ...) which actually perform the desired operation.

Since we are only interested in the lockrequest pattern of concurrent transactions, we consider an action to be executed when the concerning lockrequest has been granted by the LM. In this way we gain two advantages:

- Simplicity[4] of the transaction processing model.
- Only relevant aspects are included in our model; thus the results will not be impaired by potential bottlenecks araising at other server modules of DB-system and secondly, the results can be interpreted more accurately.

To simulate the concurrent execution of transactions, the TM serves the particular actions in the following interleaved way: Active transactions in the concurrent working set are served in a cyclic manner, one lockrequest at a time.

Surely there are more refined (and realistic[5]) scheduling strategies, but, mindful of the variety of other simulation parameters discussed later, we restricted ourselves to this straightforward scheduling strategy.

During transaction execution the TM further collects statistic data of interest (see chapter 3.3). When a transaction T has finished, i.e. T has released all its locks at EOT, T exits the DB-system. The statistic results of T are kept in a statistics module for global evaluation of the simulation run.

Another task of the TM consists in the handling of livelocks. The TM is the proper place to detect a threatening livelock because the TM has available statistic data such as a backup count for each individual transaction.

Our heuristic livelock avoidance strategy works as follows:
If a transaction's backup count exceeds a certain limit, the TM removes this particular transaction from the concurrent working set and puts it into the waiting queue before the system entry (see Fig. 3.1.) and, in exchange, takes a different waiting transaction (if there is one) into its concurrent working set. In this way, the composition of the concurrent working set is changed reducing the probability of livelocks, because it is likely that different objects are involved in lockrequests (in different modes and different sequences).
The transaction generator (see Fig. 3.1.) produces transactions to be processed with a certain arrival rate and puts them into the waiting queue in front of the system entry to the TM. If the concurrency-degree within the TM sinks below the maximum value n_{max}, the TM removes transactions from the waiting queue and includes them into its concurrent working set, if there are some transactions in the waiting queue.

3.2. Generation of a Synthetic Transaction Workload

A transaction T_i is simply modelled by a sequence of pairs of the form (object, action mode), the action mode is either 'Read' or 'Write':

$$T_i = [(O_{i_1}, am_{i_1}), \ldots, (O_{i_{l_i}}, am_{i_{l_i}})],$$

$$am_{i_j} \in \{\text{'Read'}, \text{'Write'}\},$$

l_i is called the transaction length of T_i.

T_i is called a reader, if $am_{i_j} = \text{'read'}$, $1 \le j \le l_i$.

T_i is called a pure writer, if $am_{i_j} = \text{'write'}$, $1 \le j \le l_i$.

A transaction which is neither a reader nor a pure writer is called a writer.

The problem to be solved now is, in which way do we obtain a realistic transaction profile and transaction mix to run our simulation. Because trace data from existing database systems were not available, we had to construct synthetic transactions. Again, as nowhere in the literature characteristic transaction profiles are described, we were forced to construct a probabilistic transaction generator with a variety of parameters such that many practical applications are covered by this generation mechanism.

These parameters are:

(PAR 1) Number of lockable objects $(O_1, O_2, \ldots, O_{maxobj})$.
This corresponds to the notion of lock granularity.

(PAR 2) Transaction length ℓ.
This parameter is a random variable with a specific statistical distribution, which itself is a parameter of the transaction generator. The expectation of ℓ, termed avg-ℓ, indicates what percentage of the database is locked on the average by a transaction.
Obviously in addition we must take care that $1 \le$ maxobj holds for every generated transaction T.

(PAR 3) Ratio between readers and writers.
This parameter is essential to produce different and realistic transaction mixes.

(PAR 4) Ratio between 'Read'- and 'Write'-actions within a writer.
Due to the incompatibilities among the various lockmodes, this parameter is likely to influence concurrency of accesses very heavily.

[4] This means also limitation of the programming effort.

[5] Our solution assumes equal-priority transactions and equal processing time for each action.

(PAR 5) Distribution of accesses to the database objects.

With this parameter we can model clustered or dispersed access to the DB-objects, it governs the selection of objects to be locked by a transaction. As a side condition we make sure that no duplicate objects are selected for a particular transaction.

3.3. Evaluation of the Simulation Model

In the literature the judgement of the quality of lockprotocols often is given by simple informal arguments like 'protocol 1 is better than protocol 2 because its lockcompatibilities are less restrictive'. Due to this lack of precise measures on concurrency and efficiency of lockprotocols we had to develop our own evaluation model for a quantitative comparison of different lockprotocols.

In general two contrary aspects of lockprotocols have to be analysed in order to judge their qualities:

(1) How much concurrency and efficiency is enabled by a lockprotocol?

(2) How much synchronization overhead is necessary to realize a lockprotocol?

The purpose of this paper is to give an answer to question (1), question (2) which is related to the efficiency of a lockmanager, will be investigated in a forthcoming paper.

A measure for the potential concurrency enabled by a lockprotocol is the number of simultaneously active transactions in the DB-system. Obviously, this measure is heavily influenced by the lockcompatibilities. Instead of measuring this quantity directly, we determined the number of blocking situations which indirectly relates to the same property.

A measure for the efficiency of a lockprotocol is the number of transaction backups and, in more detail, the number of backed up actions. These quantities shall give information about how much work gets lost due to the concurrent scheduling of transactions and whether a transaction is backed up early or late during its course of life.

Further quantities related to the stated measures are given in [LAN 82].

Summarizing, the measured quantities by which we want to evaluate a simulation run are

(1) the number of blocking situations

(2) the number of backed out transactions

(3) the number of actions of a backed out transaction which have to be re-processed.

Note, that it is essential to measure both quantities (2) and (3) separately. There may be situations where the number of observed backups for a protocol P1 is greater than that for a protocol P2 (processing the same workload), but the number of actions to be re-processed for P1 is smaller than that for P2.

4. Simulation Results

Let us shortly mention some implementation aspects. The functional modules of the simulation model are programmed in sequential PASCAL. To realize the dynamic simulation process we used the event-oriented simulation language SIMPAS [BRY 80], which is an extension of ordinary PASCAL. In our model we have two events, the transaction generator filling the transaction waiting queue and the TM emptying the same queue. The unit of simulation time was chosen virtually as a certain number of processed actions.
The unexpanded source code (which includes all three LMs) amounts to about 2000 lines of code.

The flexibility of a synthetic transaction workload, with its wide spectrum of possible parameter combinations, made it necessary to restrict the experiments to a few, but hopefully significant and/or realistic cases. Again we were faced with the problem that no reliable informations about transaction profiles and mixes in real systems are available. Also the problem of an appropriate lockgranularity (which is related to our parameter maxobj) is not satisfactorily answered in literature (see [RS 79]).

The following simulation parameters are fixed for all performed experiments:

- The arrival process of newly generated transactions is modelled as a Poisson process, the arrival rate λ is chosen so high that always n_{max} transactions are in the concurrent working set of the TM (at simulation start n_{max} transactions are generated at a time).
- PAR 1 is set to maxobj = 100.
  This can be considered as 100 lockable objects in the DB-system, or alternatively, these 100 objects can be regarded as the highly active part of the DB-system (catalog data, access path data, ...) where the conflict rate is most significant.
- The transaction length is uniformly distributed in the interval [5,15]. Therefore, the average transaction length avg-ℓ is 10. Thus every transaction locks approximately 10% of the DB.
- For PAR 5, objects are uniformly selected from the objects $O_1, O_2, \ldots, O_{maxobj}$ over the entire simulation run.
- The simulation run is stopped after 300 transactions have finished.
  This value has been gained by several test runs indicating the 300 is a sufficient run length in order to produce statistically reliable evaluation results.

The following experimental results reflect measured quantities, which are averages over three

simulation runs with identical input parameters, but different initializations of the employed random number generators.

Basically we performed three different test series with the above fixed parameters and with changing parameters PAR 3, PAR 4 and n_{max}.

Experiment 1

- PAR 3: only writers are generated (with different percentage of 'read'-actions)
- PAR 4 varies
- Concurrency: n_{max} = 5; (see fig. 4.1.)

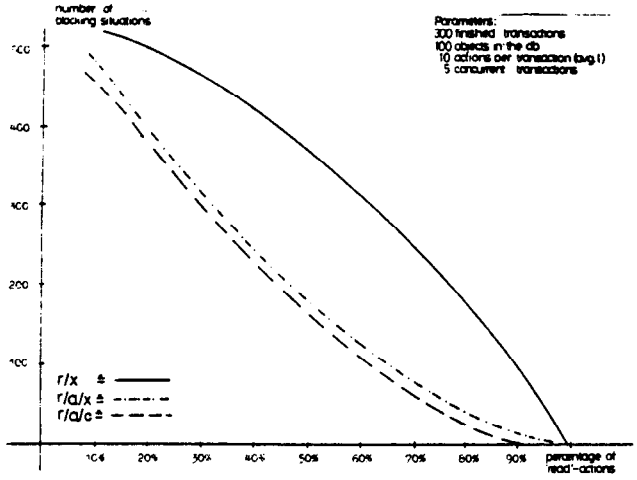

Fig. 4.1.: Relationship between the number of blocking situations and the percentage of 'read'-actions (within writers)

Interpretation of results:
At first we can observe the expected high dependence of the number of blocking situations on the percentage of 'read'-actions within the transactions.

Another point, which should be noted, is that

- the (r,a,x)- and (r,a,c)-protocol both involve significantly fewer blocking situations than the (r,x)-protocol,
- the difference between (r,a,x)- and (r,a,c)-protocol is relatively small.

With a concurrency degree of five simultaneous processed transactions there are relatively few backed up transactions. Therefore to compare $LM_{\{r,x\}}$, $LM_{\{r,a,x\}}$ and $LM_{\{r,a,c\}}$ also with respect to the number of backed up transactions n_{max} is doubled in the next experiment.

Experiment 2

- PAR 3: only writers are generated
- PAR 4 varies
- Concurrency: n_{max} = 10.

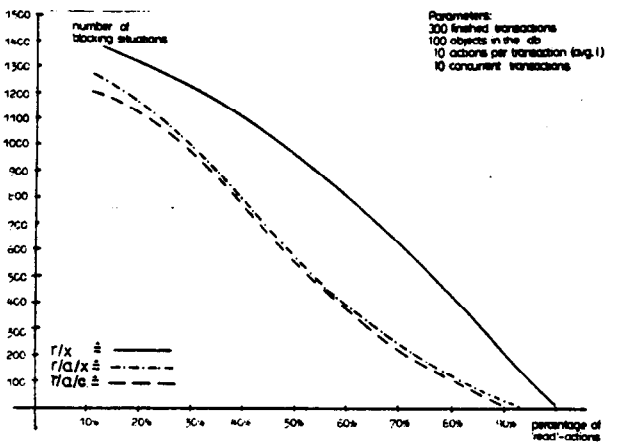

Fig. 4.2.: Relationship between the number of blocking situations and the percentage of 'read'-actions

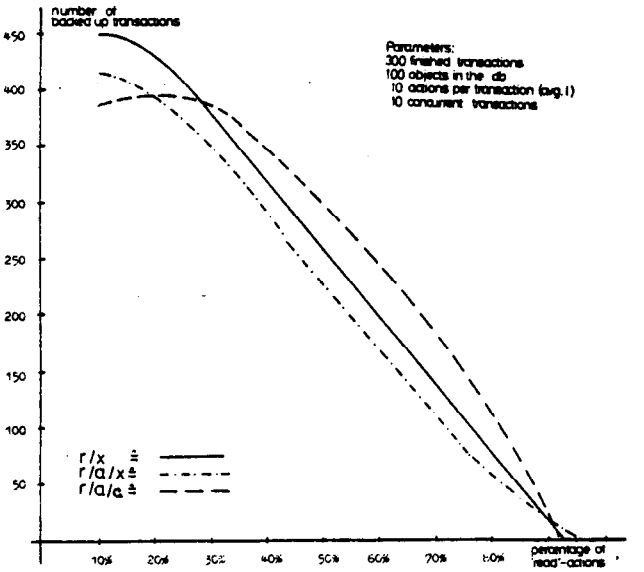

Fig. 4.3.: Relationship between the number of backed up transactions and the percentage of 'read'-actions

Interpretation of results:
Fig. 4.2. which shows the number of blocking situations depending on the percentage of 'read'-actions within the transactions is very similar to the results of the corresponding Fig. 4.1.

If we also consider the number of backed up transactions (Fig. 4.3.) it can be seen that the (r,a,x)-protocol backs up fewer transactions than the (r,x)-protocol. To our surprise the (r,a,c)-protocol causes by far the most backups on the whole.

On the other hand, if we look at the number of actions which have to be reprocessed because of the backed up transactions, we observe (Fig.4.4.) that the application of the (r,a,x)- or (r,a,c)-

protocol require more actions to be reprocessed than the (r,x)-protocol. The results in Fig. 4.3. and 4.4. were unexpected, because our intuition (and that of the designers of the (r,a,c)- protocol) was that the improved lock compatibilities of the (r,a,c)-protocol would lead to a reduction of the conflict rate among concurrent transactions and this should result in a reduced number of backups. But this feeling proved incorrect for special workloads as demonstrated above.

In summary it is now obvious that the (r,x)-protocol backs up the transactions earlier during their course of life opposite to the (r,a,x)- and (r,a,c)-protocols which resolve most conflicts at the very end of a transaction.

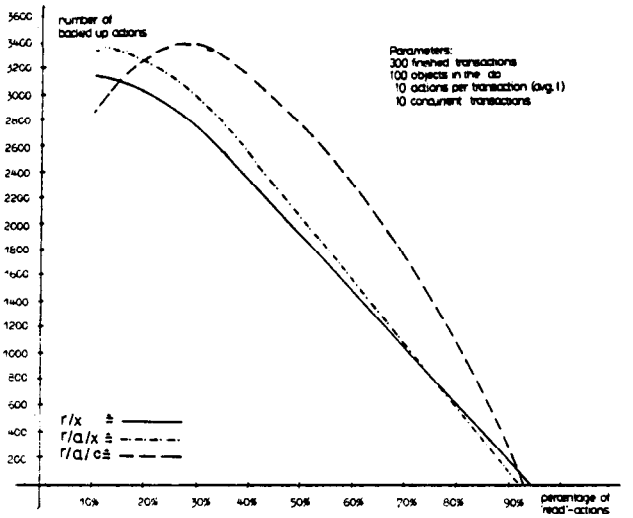

Fig. 4.4.: Relationship between the number of backed up actions and the percentage of 'read'-actions.

Experiment 3

- PAR 3 varies
- PAR 4: only pure writers are generated
- Concurrency: n_{max} = 10 transactions;

In experiment 3 we regard an important class of transactions, namely the readers, which is favored by the application of the (r,a,c)-protocol because of the compatibility of the lockmodes. Therefore we distinguish the transactions which are generated into readers and pure writers. The obtained results are represented in Fig. 4.5., Fig. 4.6. and Fig. 4.7.

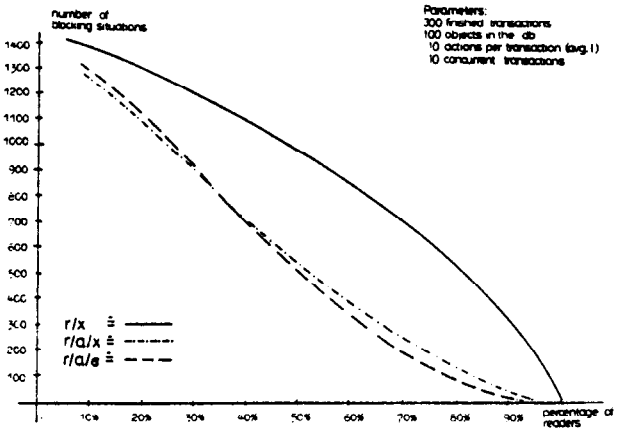

Fig. 4.5.: Relationship between the number of blocking situations and the percentage of readers within generated transactions.

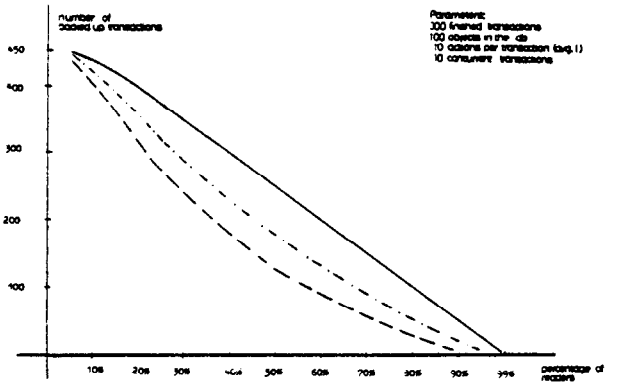

Fig. 4.6.: Relationship between the number of backed up transactions and the percentage of readers

Interpretation of the results:
Concerning the number of blocking situations (Fig. 4.5.) we receive a similar result as in Fig. 4.1. and Fig. 4.2. The correspondence with the result of experiment 2 is realistic, because in both experiments the same set of 'read'- and 'write'-actions must be processed only differently distributed on the transactions.

The property of the (r,a,c)-protocol never to back up a reader [BHR 80] involves that we now have the fewest backed up transactions using this protocol (see Fig. 4.6.). Also with the application of (r,a,x)-protocol the number of backed up transactions is reduced considerably compared to the experiment 2. Only if we use the traditional (r,x)-protocol we have the same number of backed

up transactions like in experiment 2. This means that the (r,x)-protocol does not favor a situation where transactions are distinguished into readers and writers.

Regarding also Fig. 4.7. we can conclude that a distinction of created transactions into readers and writers favors the application of the (r,a,x)- or (r,a,c)-protocol.

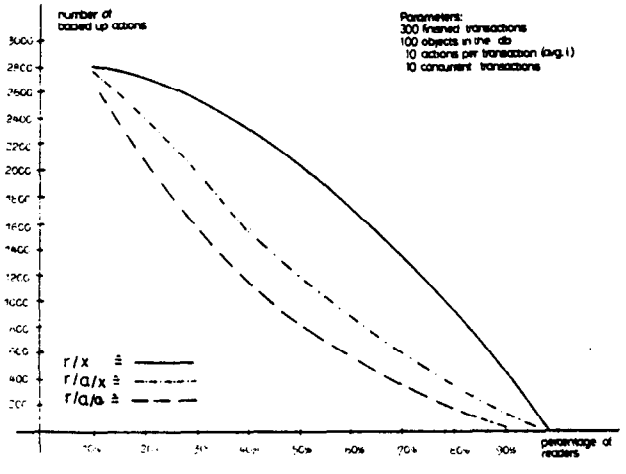

Fig. 4.7.: Relationship between the number of backed up actions and the percentage of readers.

As an alternative representation, we now show the relative relationship of blocking situations between (r,x)- and (r,a,x)-protocol and between (r,x)- and (r,a,c)-protocol.

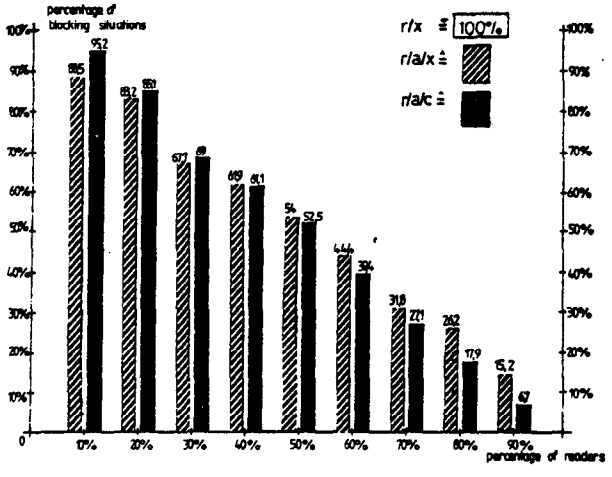

Fig. 4.8.: Relationship between the percentage of blocking situations and the percentage of readers.

From this result we derive the following rule of thumb:

With the application of the (r,a,x)- or (r,a,c)-protocol

x% readers involves about x% less blocking situations

respectively

x% 'read'-actions (within the writers) involves about x% less blocking situations.

5. Summary and Future Work

Summarizing the observations from our few basic experiments we come to the following conclusions:

The number of blocking situations is considerably reduced by both the (r,a,x)- and the (r,a,c)-protocol compared to the traditional (r,x)-protocol. Although the (r,a,c)-protocol exhibits the best behavior concerning blocking situations, the difference to the (r,a,x)-protocol is relatively small; this proposition also carries over to the average number of blocked transaction.

In general we recommend the application of the (r,a,c)- protocol for transaction mixes which allow a distinction of transactions into readers and (pure) writers. In special applications, where such a separation does not exist and where the percentage of write-operations is very high, the increased backup probability and lost work of the (r,a,x)- and (r,a,c)-protocol may cause troubles. As this phenomenon is caused by the fact, that inconsistency checks are done only at the latest possible moment (in order to minimize the cycle searching overhead), it would be promising to make inconsistency checks earlier increasing the overhead for cycle detection. However, to overcome this overhead without significant impact on concurrency the method of dynamic timestamps can be applied which replaces the costly cycle detection ([BEHR 82]).

Evidently our results should be verified by real transaction profiles and workloads. As these are commonly not available by now, it is highly desirable that existent DB-systems are benchmarked to get the required data.

Our planned future work on lockprotocol evaluation is as follows:

The next step is to complete the present investigations by a comparison of the synchronization overhead required by the $LM_{\{r,x\}}$, $LM_{\{r,a,x\}}$ and $LM_{\{r,a,c\}}$, including also the dynamic timestamp method from [BEHR 82] and static timestamp methods as described in [RSL 78]. Furthermore, these lockprotocols will be compared to optimistic concurrency control methods as suggested in [KR 79]. Finally, extensions of these one-level lockprotocols to hierarchical lockprotocols as discussed in [GLPT 76] and [BAY 76] should be evaluated, with a special attention to gain reliable results with respect to an optimal lockgranularity.

Most of these planned investigations are assumed to require only slight modifications of the ex-

isting basic simulation model. The goal of these activities is to finally get a reliable assessment and selection procedure for a suitable lock-protocol given a specific transaction workload.

Acknowledgements:

Many thanks to Prof. Rudolf Bayer and Dr. Angelika Reiser for critical comments on a first draft of this paper.

6. References

[BAD 80] Badal, D.Z.:
'The Analysis of the Effects of Concurrency Control on Distributed Database System Performance', Proc. Int. Conf. on VLDB 1980, pp. 376-383

[BAY 76] Bayer, R.:
'Integrity, Concurrency, and Recovery in Databases', ECJ Conf. 1976, Berlin, Springer 76, Lect. Notes in Comp.Sci. 44, pp. 77-106

[BEHR 82] Bayer, R.; Elhardt, K.; Heigert, J.; Reiser, A.:
'Dynamic Timestamp Allocation for Transactions in Database Systems', in Distributed Data Bases, edited by H.-J. Schneider, North-Holland, 1982, pp.9-20

[BHR 80] Bayer, R.; Heller, H.; Reiser, A.:
'Parallelism and Recovery in Database Systems', in ACM Transactions on Database Systems, Vol. 5, No. 2, June 1980

[BRY 80] Bryant, R.M.:
'SIMPAS User Manual', Univ. of Wisconsin-Madison, June 1980, Comp.Sc. Techn. Report No. 391

[EGLT 76] Eswaran, K.P.; Gray, J.N.; Lorie, R.A.; Traiger, J.L.:
'The Notions of Consistency and Predicate Locks in a Database System', in Comm. ACM 19, 11, Nov. 1976, pp.624-633

[GLPT 76] Gray, J.N.; Lorie, R.A.; Putzolu, G.F.; Traiger, J.L.:
'Granularity of Locks and Degrees of Consistency in a Shared Data Base', in Modeling in Data Base Management Systems, G.M. Nijssen editor, North-Holland, 1976 pp. 365-394

[GRA 80] Gray, J.N.:
'A Transaction Model', IBM Res. Lab. San Jose, RJ 2895 (36591), 8/7/80

[KR 79] Kung, H.R.; Robinson, J.:
'On Optimistic Methods for Concurrency Control', Proc. Int. Conf. on VLDB, Rio de Janeiro, Oct. 1979

[LAN 82] Landherr, G.:
'Simulationsuntersuchungen zur Parallelität von Datenbanksperrprotokollen', Techn. Univ. Munich, Institute of Informatics, Master's Thesis Oct. 1982

[MN 82] Menasce, D.A.; Nakauishi, T.:
'Optimistic versus Pessimistic Concurrency Control Mechanisms in Database Management Systems', in Inf. Syst., Vol. 7, No. 1, 1982, pp. 13-27

[PL 80] Potier, O.; Leblanc, Ph.:
'Analysis of Locking Policies in Database Management Systems', in Comm. ACM, Oct. 1980, Vol. 23, No. 10, pp.584-593

[RS 79] Ries, D.R.; Stonebraker, M.R.:
'Locking Granularity Revisited', in ACM Transactions on Database Systems, Vol.2, No. 1, March 1977, pp. 91-104

[RSL 78] Rosenkrantz, O.J.; Stearns, R.E.; Lewis, P.M.:
'System Level Concurrency Control for Distributed Database Systems', ACM Transactions on Database Systems, Vol.3, No. 2, June 1978, pp. 178-198