Empirical Comparison of Database Concurrency Control Schemes

Peter Peinl
Andreas Reuter

Dept. of Computer Sciences, Univ. of Kaiserslautern, West-Germany

Abstract
This paper presents an empirical comparison of three classes of database concurrency control schemes: classical (r,x)-protcols, (r,a,c)/(r,a,x)-protocols based on two temporary object versions, and optimistic concurrency control. Evaluation is based on six different real-life database reference strings recorded with a network DBMS. Sizes of the underlying databases vary between 60 MB and 2.9 GB. For comparing the performance impacts of synchronization protocols, we introduce a quantitative measure which basically reflects two parameters: First, number and length of blocking situations, and second the overhead incurred by repeated transaction execution due to deadlocks and validation conflicts, respectively. The results are highly surprising compared to the expectations based on qualitative considerations.

1. Introduction

Locking as the basic means of concurrency control in database management systems (DBMS) has been introduced and fully investigated in the landmark papers (EGLT76, GLPT76). It was there that levels of consistency were defined, that serializability as the basic criterion of integrity-preserving execution was introduced, and that the concept of a transaction entered the database scene. Based on this seminal work there were many investigations concerning more formal notions of consistency and serializability, and meanwhile there is a well-established theory of database concurrency control, represented, e.g., by (BSW80). Performance impacts of locking schemes have initially been discussed in a very restricted sense: One topic was the granule of locking and its implication on transaction parallelism (GLPT76, RS77, RS79), the other one was the overhead incurred by maintaining lock tables, dependency graphs etc. during normal operation (Gr78). Both problems are still highly relevant, as can be seen from performance analyses (or better: code reviews) of nowadays commercial DBMS.

In parallel to the development of serializability theory, novel concurrency control mechanisms have been proposed, all of which differ from the classical (r,x)-scheme (since an object can have a read-lock or an exclusive lock) in one salient point: They permit readers to access an object which is currently changed by a writer, which is impossible with (r,x)-locking. It can be shown that the results are consistent, though, based on the serializability criterion, and the methods for detecting and resolving conflicts shall briefly be sketched for the two approaches we are going to compare with (r,x)-protocols in this paper.

The first one has been proposed in (BHR80), and the basic idea is to give the **old** value of an object currently under change (objects are normally pages) to all readers requesting access while the writer is busy. So an object can have an r-lock, an a-lock which allows one transaction to prepare a new version, while an arbitrary number of readers can analyze the old state, and a c-lock, if the writer wants to commit its new version. If the object is in c-mode, new read transactions will be directed to the new version, which will become the only one, as soon as the last reader using the old version has finished. There is a variant of the (r,a,c)-protocol which does not allow for concurrent usage of an old and a new **committed** version; this one is termed (r,a,x)-protocol. The other approach to the same problem originates in the observation that in some applications conflicts are extremely rare - so why burden

the system with costs for locking, instead of let things go and see what happens. The synchronization protocol based on this liberal view is called optimistic concurrency control (occ) and has first been described in (KR81). It assumes that all transactions access all objects they need without locking; updates are prepared in a private working space, and before end-of-transaction a test is performed to detect conflicts with other transactions. If there are none - which is supposed to be the normal outcome - the transaction finishes and commits its updates if there are any. Otherwise it has to be backed out and then must try again. Various possibilities for implementing this scheme are discussed in (Hä82).

There are some more synchronisation protocols, especially those based on time-stamps, but we want to investigate these three, (r,a,c), (r,a,x), occ, and compare their performance impact on a real database workload with the figures achieved by classical (r,x)-locking. For doing this, we have organized the paper as follows: In sec. 2 we present the empirical basis of our measurement, in sec. 3 we discuss the problem of meaningful quantitative performance comparison, sec. 4 contains the empirical results, and in sec. 5 we give some interpretations and identify interesting open problems.

2. Empirical Performance Comparison - Methods and Material

It is interesting to note that until recently the different synchronization protocols have only been compared w.r.t. the kind of equivalent serial schedule they produce; results on absolute and relative performance figures, however, are virtually unavailable. This is a strange situation, since all the novel proposals claim to improve performance by reducing administration overhead, transaction waits, or both. We will show in sec. 3 that establishing a meaningful performance measure for this purpose is all but trivial, but let us defer this problem for the moment and discuss the empirical foundation for such a performance comparison first.

To our knowledge, there are two papers dealing with the impacts of different synchronization schemes on transaction waits, deadlocks, etc. (KL82, MN82). In the latter paper the lock strategy as well as the database and the transactions' references are modelled by using stochastic processes and waiting queues. In (KL82) the lock protocol is actually coded, but it is driven by simulated references obtained from a random number generator. And this is the crucial problem: The real reference pattern of a multiuser database application is extremely dependent on the database structure, the data stored in the DB, and the input data to the transactions. Until now there is no way for modelling database references with sufficient precision (EH82). Note that some protocols (like occ, e.g.) are explicitly designed for certain types of reference patterns, and so their value will critically depend on whether or not you can find the required types **in reality**.

For these reasons we have decided not to rely on simulation, neither for the database and the transactions, nor for the protocols themselves. The other extreme, namely modifying a real DBMS by implementing all the synchronization protocols and then putting it into operation in some applications, was not viable, too - for reasons of expence. But the method we finally applied is very close to this approach, and we will highlight the most important features in the following.

First, we used a CODASYL-like DBMS (UDS) to record <u>logical page reference strings</u> (EH82) in 6 applications with different characteristics. A page reference string is kind of an audit trail of the DBMS-internal activities, and it contains the following record types:

- Begin-of-transaction (BOT), end-of-transaction (EOT), abort-transaction (RBT). In each case, the transaction number, run-unit identifier and some other data are recorded.
- Logical page reference
  This indicates that some module inside the DBMS has requested a page from the buffer manager on behalf of a certain transaction. The transaction no. is recorded as well as the type of reference (read or update). For instance, fetching a record via an attribute index implemented by a B*-tree would result in logical references to the root, to all pages visited down to the leaf node, and finally to the record's page.
- Page wait events
  Our DBMS does page locking with long exclusive and short read locks - a strategy we did not want to investigate. So page waits were transformed into <u>normal page references</u> and whether or not wait situations did occur was left to the resp. synchronization protocol. In other words: Transactions did not have to wait in our simulation if they had to in reality. This was left to the synchronization protocol under consideration.

These data were recorded for all 6 applications with a characteristic transaction load over a certain period of time (this type of audit trail is extremely expensive), resulting in reference strings containing some 10.000 to 100.000 page references.

The underlying databases and the characteristics of the cut-out of the transaction

mixes reflected in the reference strings are summarized in Table 1.
The sizes of read and write sets used for characterizing the transaction types are in units of pages (the minimum of the resp. class is specified). Since our DBMS did page locking and the novel protocols we have investigated are especially suited for page-sized objects, we have used pages as the smallest units of resource requests in all cases. A note for those who are familiar with implementational details: pages containing system administration data like database key translation tables etc. have not been processed as objects of normal references, since they would severely impede concurrency using page locking. Since there are several techniques for synchronizing transactions on such tables **indirectly**, dropping the resp. references from our consideration was quite realistic, since each implementation would take the same approach.
Obviously, the mixes are considerably different w.r.t. transaction types and sizes. Strings nos. 1 and 2 originate from a scientific database with a large share of batch transactions; all transactions are comparatively long, since they either have to check complex integrity constraints, or have to read large portions of the database. Primary design goal of these applications was to correctly reflect the complex structure of the miniworld **completely**. Strings 3-6 have been recorded in different DB/DC-environments with interactive processing only; therefore, transactions are short in order to minimize response time.
There was one problem with these empirical data: Each string was recorded with a fixed degree of parallelism which has been specified at system start-up time. But we want to know how these reference patterns behave under the synchronization protocols mentioned at **different degrees** of multiprogramming.
The solution we have implemented is illustrated in Fig. 1.

no.of ref. string	sizes of DB	no.of TA's (total)	writers in %	transaction types (occurrences in %/read set size/write set size					
				sr	ar	lr	sw	aw	lw
1	60 MB	250	94.0	0	2.0/50/0	4.0/100/0	68.0/6/4	24.4/7o/4	1.6/110/100
2	60 MB	39	18.0	0	30.8/50/0	51.3/100/0	10.3/6/4	7.7/70/4	0
3	150 MB	669	46.5	16.2/1/0	23.5/6/0	13.7/21/0	33.0/1/1	6.9/4/2	6.7/11/5
4	2.9 GB	2014	14.8	59.2/1/0	17.2/6/0	8.7/11/0	4.0/1/1	8.7/11/5	2.1/31/14
5	280 MB	847	32.0	23.2/1/0	41.9/6/0	2.9/11/0	8.2/1/1	10.5/21/10	13.3/31/15
6	560 MB	2286	46.2	40.2/1/0	13.3/6/0	0	25.3/1/1	20.4/6/3	0.54/11/4

sr : short reader, ar : average reader, lr : long reader, sw : short writer, aw : average writer, lw : long writer

Table 1: Characteristics of transaction mixes and databases

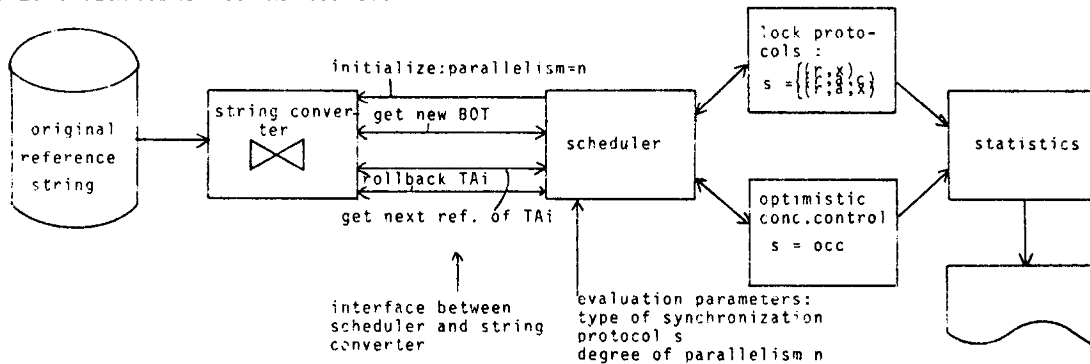

Fig. 1: Block diagram of the evaluation environment

There is a scheduler driving the evaluation programs, keeping track of the active transactions, the blocked ones, etc., in other words it is a rudimentary transaction scheduler as can be found in each DBMS. As input parameters for each evaluation it is provided with the protocol to be applied (s) and the desired degree of multiprogramming (n). The scheduler does not directly read the reference string file, rather it interfaces

with a string converter, which delivers BOT-records, logical references, etc. **as though the string were recorded** with n transactions in parallel. This is done by reading ahead in the reference string file until n BOT-records have been located and then searching for the logical references pertaining to those transactions. With each logical reference and EOT-record the scheduler invokes the synchronization protocol requested (all of them have been explicitly coded). In return, they inform the scheduler about whether the request was granted, or the transaction has to wait, or the transaction must be rolled back. Note that the logical references are not really processed; there is no real DB envolved in the evaluations. The behaviour of the complete DBMS - except the synchronization manager - is considered to be condensed in the reference strings, which are then subject to the different synchronization protocols. Of course, the real transactions on the real database executed with a DBMS using the resp. protocol would have behaved slightly different, but we believe this to be a tolerable idealization since **the reference patterns** would not change anyway.

3. Citeria for Performance Comparison

Given the empirical data about transaction references, the scheduler, the routines incorporating the synchronization protocols - which values can be used to characterize the performance impacts of each protocol, and which quantitative criteria can be used to compare protocol A to protocol B? In (MN82) a synthetic response time parameter is introduced for this purpose, but the underlying processor and device model is idealized to the same degree as the rest of the simulation. In (KL82) three measures are considered: the number of blocking situations, the number of backed transactions and the number of backed up actions[1]. These values are computed for different shares of read transactions in the mix. Moreover the impact of write operations in proportion to read operations within update transactions is evaluated. It is quite obvious that the number of backups, be it caused by deadlocks or other types of conflicts, is an interesting parameter for characterising a synchronization protocol. But one must see clearly that some protocols are more susceptible to backup than others. The occ-approach, e.g., resolves almost each conflict by backing up a transaction; on the other hand it has the virtue of causing no blocking situations at all. And counting backups alone will not reveal sufficient information, since it makes a great difference whether primarily short, medium-sized, or long transactions are victims of backup. This in turn depends on the type of the protocol.

A similar problem arises when counting the number of blocking situations. Even a large number of transaction waits can be tolerable if they are short, and on the other hand a few number of extremely long blockings can become intolerable. So we rather should know which transaction (transaction types) have to wait, and how long they are blocked.

Based on these considerations we have decided to use two parameters, which are defined as follows:

- $\bar{n}$ is the average degree of multiprogramming. As described in sec. 2, a maximum degree of multiprogramming, n, is specified for each evaluation. This means the scheduler will try to keep n transactions busy in parallel at each moment. But due to blocking situations the effective parallelism will decrease now and then. So with each logical reference processed by the scheduler the number of **active** (i.e. unblocked) transactions is added to a counter which is divided by the total number of references at the end. Hence, $\bar{n}$ contains the average number of active transactions during processing of the mix, and thereby reflects the length of blocking situations in an appropriate way.
- q is the relative increase of number of references in the string due to backup and re-execution of transactions. Let r be the number of logical references in the string (there are no deadlocks in the input strings), and r' the number of references actually processed by the scheduler, then r' will be greater or equal r, since each time a transaction must be rolled back it is started again and the number of references processed twice adds to the total number of references. Then q denotes the relative elongation which is independent of the absolute size of the string, i.e. $q = r'/r$.

With these two parameters we still have a plane where the different protocols will fall in, and there is no obvious way for deciding whether A is better than B. If the same reference string yields $\bar{n} = 6.2$ and $q = 1.12$ with one protocol, and $\bar{n} = 7.3$ and $q = 1.23$ with the other - which is better? For occ-schemes we will always have $n = \bar{n}$, i.e. the only distinction can be made via q. To resolve this problem in a simple way we introduce a, performance measure n^*, which is based on the following idea:

Let protocol s be characterized by $\bar{n}_s$ and $q_s > 1$. What we actually want to do is process r logical references; if $q_s \cdot r$ are required,

1) An action in (KL82) can be regarded as a logical page reference according to our terminology.

this is overhead incurred by the synchronization protocol. In other words, a certain amount of the $\bar{n}_s$ transactions running concurrently are engaged in re-processing transactions instead of doing useful work. Since r $(1 - q_s)$ is the number of re-executed references, a proportional part of $\bar{n}_s$ is required to process them. The rest is processing original references, and this is what will be denoted by n*. So we get:

$$n_s^* = \bar{n}_s/q_s$$

This simple model makes a lot of implicit assumptions, one of which shall be mentioned explicitly. We state that the underlying machine, the channels, the storage devices etc. are capable of processing twice the amount of references per time interval, if we double n. Otherwise n* would not be a useful comparative measure. In our evaluations we have varied n between 2 and 32, so this should not be too far from realistic. We will return to this in sec. 5.

But note that these are problems of performance comparison only at the level where the objects used for synchronization are visible. There are more subtle - and possibly more important - implications on lower layers of implementation, having strong implications on performance, which will also be mentioned in sec. 5.

4. Empirical Results

The empirical evaluation of the three classes of synchronization protocols yielded many interesting statistics about the dynamic behaviour of each algorithm. Presenting them all would exceed the scope of this paper, and many performance figures still require thorough analysis and interpretation. By the way, when implementing the synchronization protocols there were numerous pragmatic decisions to be made how to cope with special situations, none of which could be derived from the original papers. We have tried to implement each protocol such that the basic idea was preserved as much as possible, but we have also tried to find good solutions in terms of performance. A particular problem which is not dealt with in the conceptual descriptions of the algorithms is lock conversion and deadlock handling. In sec. 5 we will briefly discuss this issue when interpreting the results.

The following tables display the basic performance parameters introduced in sec. 3 for each reference string and each synchronization protocol.

	(r,x)			(r,a,c)		
n	$\bar{n}$	q	n*	$\bar{n}$	q	n*
2	1.92	1.05	1.72	1.90	1.21	1.57
4	2.97	1.25	2.38	3.51	1.62	2.16
8	5.16	1.47	3.62	5.90	2.03	2.91
16	8.90	1.70	5.22	9.44	1.90	4.97
32	14.99	1.83	8.20	11.90	1.93	6.18

	(r,a,x)			occ		
n	$\bar{n}$	q	n*	$\bar{n}$	q	n*
2	1.83	1.08	1.69	2.	1.27	1.57
4	3.29	1.32	2.49	4.	1.86	2.15
8	5.73	1.52	3.77	8.	1.67	4.79
16	8.98	1.75	5.13	16.	1.60	10.00
32	17.75	1.77	10.03	-	-	-

Table 2: Performance of refstring no. 1

The next mix containing only few update transactions shows almost no difference between the four synchronization protocols. Since the string comprises no more than 39 transactions, we have restricted evaluation to n < 16.

	(r,x)			(r,a,c)		
n	$\bar{n}$	q	n*	$\bar{n}$	q	n*
2	2.00	1.00	2.00	2.00	1.00	2.00
4	3.98	1.00	3.98	4.00	1.00	4.00
8	7.90	1.00	7.90	7.97	1.00	7.97
16	14.32	1.00	14.32	15.62	1.00	15.62

	(r,a,x)			occ		
n	$\bar{n}$	q	n*	$\bar{n}$	q	n*
2	2.00	1.00	2.00	2.	1.00	2.00
4	3.98	1.00	3.98	4.	1.01	3.96
8	7.91	1.00	7.91	8.	1.01	7.92
16	13.75	1.00	13.75	16.	1.03	15.53

Table 3: Performance of refstring no. 2

The results for reference strings nos. 3-6, originating from commercial DB/DC-applications with transactions being completely different from those on the first two mixes, are shown in Table 4 - Table 7.

	(r,x)			(r,a,c)		
n	$\bar{n}$	q	n*	$\bar{n}$	q	n*
2	1.94	1.00	1.94	1.99	1.00	1.99
4	3.52	1.01	3.48	3.81	1.03	3.71
8	6.05	1.01	5.99	6.87	1.03	6.68
16	8.37	1.02	8.24	10.39	1.05	9.93
32	10.98	1.06	10.35	17.20	1.13	15.15

	(r,a,x)			occ		
n	$\bar{n}$	q	n*	$\bar{n}$	q	n*
2	1.95	1.00	1.95	2.	1.08	1.85
4	3.59	1.00	3.59	4.	1.21	3.30
8	6.14	1.00	6.14	8.	1.25	6.40
16	9.36	1.05	8.92	16.	1.36	11.76
32	14.15	1.05	13.48	32.	1.57	20.38

Table 4: Performance of refstring no. 3

n	(r,x) $\bar{n}$	q	n*	(r,a,c) $\bar{n}$	q	n*
2	2.00	1.00	2.00	2.00	1.00	2.00
4	3.77	1.00	3.77	3.84	1.00	3.84
8	6.42	1.01	6.35	6.77	1.00	6.75
16	9.06	1.06	8.51	9.56	1.09	8.79
32	13.86	1.09	12.77	16.38	1.10	14.89

n	(r,a,x) $\bar{n}$	q	n*	occ $\bar{n}$	q	n*
2	2.00	1.00	2.00	2.	1.01	1.98
4	3.80	1.00	3.80	4.	1.19	3.36
8	6.75	1.00	6.75	8.	1.22	6.56
16	9.71	1.09	8.90	16.	1.28	12.50
32	19.66	1.07	13.70	32.	1.42	22.53

Table 5: Performance of refstring no. 4

n	(r,x) $\bar{n}$	q	n*	(r,a,c) $\bar{n}$	q	n*
2	1.95	1.00	1.95	1.98	1.00	1.97
4	3.74	1.01	3.69	3.67	1.02	3.59
8	5.94	1.06	5.58	5.99	1.05	5.69
16	8.44	1.08	7.80	9.27	1.09	8.50
32	11.73	1.11	10.51	13.12	1.11	11.84

n	(r,a,x) $\bar{n}$	q	n*	occ $\bar{n}$	q	n*
2	1.95	1.00	1.95	2.	1.22	1.64
4	3.63	1.02	3.54	4.	1.19	3.36
8	5.92	1.05	5.64	8.	1.28	6.25
16	9.01	1.07	8.42	16.	1.55	10.32
32	12.10	1.09	11.10	32.	1.91	16.75

Table 6: Performance of restring no. 5

n	(r,x) $\bar{n}$	q	n*	(r,a,c) $\bar{n}$	q	n*
2	1.99	1.01	1.97	2.00	1.01	1.97
4	3.93	1.02	3.86	3.98	1.05	3.78
8	7.56	1.06	7.11	7.79	1.11	7.04
16	13.38	1.15	11.64	14.45	1.25	11.54
32	23.89	1.25	19.12	25.45	1.39	18.29

n	(r,a,x) $\bar{n}$	q	n*	occ $\bar{n}$	q	n*
2	2.00	1.01	1.97	2.	1.01	1.97
4	3.97	1.04	3.82	4.	1.05	3.83
8	7.68	1.11	6.92	8.	1.15	6.94
16	14.18	1.23	11.53	16.	1.39	11.53
32	24.48	1.40	17.49	32.	1.99	16.06

Table 7: Performance of refstring no. 6

Before entering discussion and interpretation of these results, let us first illustrate some of the performance figures by related data. Parameter q indicates the relative increase of the number of logical references that must be processed in order to complete all transactions in relation to the net number of references in the string. A q-value > 1 is always caused by transaction rollback, but the relative number of failing transactions does not grow proportional to q. The reason is that at low parallelity transactions run comparatively long, and have processed many references before they are rolled back due to deadlock or validation conflict. With a high degree of multiprogramming this happens much earlier, i.e. the number of transaction restarts can grow faster than does the corresponding q. This is especially true for occ-schemes where in case of a validation conflict either the validating transaction is aborted, or the other transactions being in conflict with it are killed. To get an impression of what level 3 consistency costs with the protocols under investigation in terms of deadlock frequency and transaction restart, look at Table 8.

n	(r,x) #dead-locks	#lock con. r→x	(r,a,c) #dead-locks	#lock conv. r→a
2	9	497	14	503
4	15	506	44	545
8	59	567	97	617
16	143	692	227	811
32	246	817	354	1001

n	(r,a,x) #dead-locks	#lockconv. r→a	occ #restarts	#validats.
2	14	503	17	900
4	38	536	40	2667
8	99	621	120	6224
16	203	774	322	13605
32	348	1011	833	28578

Table 8: Transaction restart frequency for refstring no. 6

This table applies to reference string no. 6, containing many very short transactions, i.e. a type of application which can be found in many DB/DC-environments. The parameter deadlocks means the same for locking-schemes as does restart for occ-schemes, namely the number of transaction restarts. The number of lock conversions is particularly interesting for the discussion in 5.3. The number of validations is one cost measure for EOT-handling in occ-schemes; it indicates how many read- and write sets have been compared in total for validating transactions at EOT.
Looking at the (r,a,c)-protocol, it is quite interesting to note that the majority of deadlocks is caused by a certain type of resource request, namely an attempt to convert a read-lock into an update lock. The figures are given in Table 9.

n	# of deadlocks caused by		
	cyclic wait	conv. r→a	conv. a→c
2	0	14	0
4	0	43	1
8	1	88	8
16	0	202	25
32	2	316	36

Table 9: Requests causing deadlock using (r,a,c) with refstring no. 6

The same observation is true for the (r,a,x)-scheme, and we will briefly investigate the problem in 5.3.
As a final overview over the results look at Fig. 2. The horizontal axis displays the degree of multiprogramming, n, and the vertical axis corresponds to the number deadlocks (restarts) of Table 8. The curves are shown for (r,x), (r,a,c) and occ. The surprising fact is that restarts of occ grows faster than linear, as could be expected. In fact, it should grow with n^2. The growth of deadlock rate in the lock-oriented schemes, however, slows down for high parallelity, and this is in contradiction to some analytic models. This phenomenon, however, requires some further investigation.

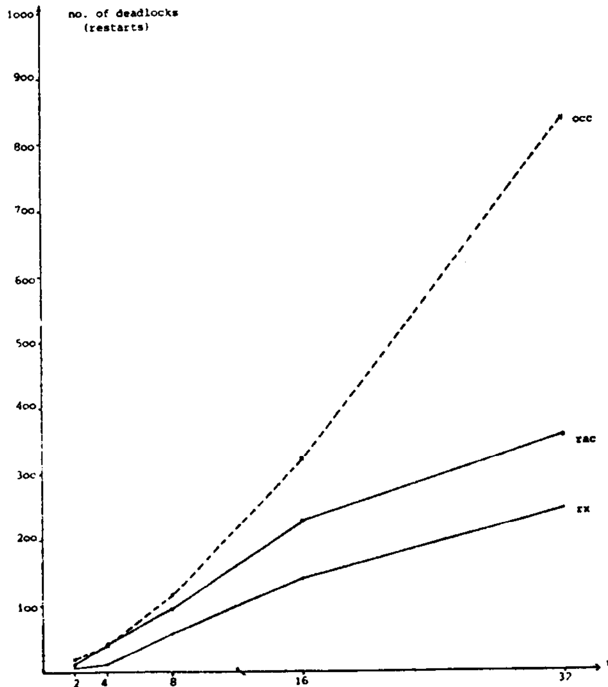

Fig. 2: Deadlock frequencies at different degrees of multiprogramming

5. Interpretation And Open Problems

To say the least, the results reported in sec. 4 have been a great surprise to us. The new approaches to concurrency control have been designed to improve classical (r,x)-locking in different situations. As we explained in the beginning, the occ-scheme tries to avoid locking control overhead in applications where conflicts are **rare**. The (r,a,x)/(r,a,c)-schemes want to provide a restricted kind of read- and write-concurrency in cases which would cause a blocking situation with normal locking protocols. These have sometimes been called **pessimistic** protocols in that they assume conflicts to be frequent and thus tend to be overly restrictive in cases where this is not true. But looking at Tables 2-7, occ seems to be superior in most cases, especially at a high degree of multiprogramming, whereas (r,x)-protocols and the improved locking schemes exhibit quite similar performance figures, with obvious advantages for (r,a,x)/(r,a,c). Since this is not what one might have expected, we must investigate the results in more detail. The transaction loads characterized in Table 1 are not likely - different as they are - to be adequate for occ-schemes, with the only exception of reference string no. 2, which represents an almost pure retrieval load. Strings 3 to 6 have been recorded in typical commercial applications with a significant share of update transactions; hence, the optimistic assumption certainly does not hold. But the results shown in sec. 4 do indicate that - at least in terms of our performance parameter n* - optimistic concurrency control performs excellently compared to the pessimistic schemes in most of the "unfavourable" environments. Are these results likely to be correct, or are they due to a fault in the evaluation? And if they are correct - how can they be explained w.r.t. the optimistic/pessimistic dichotomy?

5.1 The Performance Criteria Reconsidered

The performance comparison presented in sec. 4 rests on two variables which can be observed directly: $\bar{n}$, i.e. the average number of unblocked transactions at a nominal degree of multiprogramming of n; and q, which denotes the relative increase of references in the string due to transaction backout. Comparing the empirical results, one can see clearly that occ-schemes generally have the highest q-values. In other words, validation conflicts (KR80) cause transactions to be executed repeatedly to a much higher degree than do deadlocks in the locking oriented schemes. Hence, the overhead of references not contributing to useful work, which can be measured by $(1-q)\cdot r$ for each reference string, is significantly higher for occ-schemes. This is what could have been expected for this type of application. The comparatively good

performance figure n^* is due to the fact that in occ-schemes there are no blocking situations, i.e. at each moment n transactions do execute concurrently[1], so $\bar{n} = n$, which is not true for pessimistic schemes. But what does it mean to compare the n^* of a synchronization protocol where 32 transactions are executed in parallel at each moment, with another one where in the average only, say, 24 are active while the others are blocked? In order to be meaningful, the comparison assumes an underlying processor which is powerful enough to actually service 32 concurrent transactions such that none of them is impeded by any other - w.r.t. its processor requirements (see sec. 3). If this assumption holds, then it does not matter if a certain amount of useless work is done (high q-value); if parallelism is high enough, we will get our transactions through the system at a speed which is proportional to the value of n^*.

After all - how realistic is this kind of processor model? Remember that at least the occ- and the (r,a,c)/(r,a,x)-schemes do require a very large database buffer in order to be implementable with reasonable performance (Our evaluation has always assumed a sufficiently large buffer; storage costs have been neglected). Hence, overlapping of I/O-operations for timesharing between transactions will scarcely occur. One might argue that with a large mainframe of, say, 4-6 MIPS, the delay incurred by terminal I/O, user think times, etc. should be sufficient for efficiently servicing 32 transactions in an interleaved manner. But in large interactive environments terminals are not directly connected to database subtasks. Rather they are scheduled by DB/DC-systems, where terminal input and output is queued in the DC-subsystem, which will deliver new requests to the DBMS as soon as a transaction service station has become available. Hence, to classical centralized systems our processor model may not apply completely, at least for large values of n. In such systems, the key parameter for synchronization protocol performance is q, i.e. the smaller q is, the smaller will be the overhead of useless references, and the better will be the throughput. This criterion yields almost identical results for (r,x) and (r,a,c)/(r,a,x)-schemes (with slight advantages for (r,x)), whereas occ is definitely more expensive (see sec. 5.2).

On the other hand, for real high performance systems of the "one processor per transaction/sec."-type, our processor model is quite adequate, and we can use n^* as the only performance criterion, which makes things look completely different.

Besides the quantitative criteria, there is an other aspect to be considered, which might be called "fairness of processing". As our experiences show, the pessimistic schemes give almost equal chances to all types of transactions - from "short readers" to "long writers" - to be successfully processed. The only exception will be discussed in 5.2. With the occ-scheme, however, there are three classes of transactions:

- the small ones causing no validation conflicts,
- the medium sized, causing occasional validation conflicts,
- the long transactions, causing permanent validation conflicts.

Transactions of the latter type have sometimes been re-executed over and over again, till to the end of the reference string, when they could be processed exclusively. Since this made the q-value grow beyond all reasonable boundaries, we had to implement some dynamic load balancing, which is briefly sketched in 5.4.

5.2 Deadlock Handling

The most difficult problem during the evaluation of different synchronization protocols was the phenomenon of recurring deadlocks - in case of lock protocols - and of permanent validation conflicts - in case of optimistic concurrency control, as mentioned above. Such problems did not occur in the original applications, so what was the reason in our reference string - driven evaluation? There are two answers: First, the DBMS used for recording the reference strings did only hold **short** read locks (level 2 consistency), i.e. there were less conflicts and, consequently, less deadlocks than with the level 3-protocols we have implemented. The second reason is more important. In the recording environments with their fixed degree of multi-programming, reference density of the single transactions was very irregular. User think times, page faults, log I/O etc. caused several references of **one** transaction to be processed, while **another** transaction did not issue any reference. If a transaction failed due to deadlock, its restart would take a relatively long time in terms of references processed meanwhile. For evaluation purposes, however, we have converted the reference strings to arbitrary degrees of concurrency (see Fig. 1), and therefore had to impose some scheduling on the transformed string of references. As a first approach, we have used

---

[1] Of course, at the end of the reference string when no more transactions can be started, our occ-evaluation yielded a smaller $\bar{n}$, too. But except very small strings (string no. 2, e.g.) this does not significantly influence the overall degree of multiprogramming.

a simple round robin strategy. Each active transaction which is ready to execute (i.e. which is not blocked) issues one reference, then comes the next, etc. As soon as one transaction has finished, the next one is started - provided there is a next one in the input reference string. So each transaction will see a more or less different environment than in the original application, and the actual degree of concurrency will be higher. If we run an evaluation with a nominal parallelity of n = 8, then the scheduler will always keep 8 transactions active. In reality, however, only, say, 6 transactions will be active in the average, due to delays in the DC-subsystem, the operating system, user think time, etc. All this contributes to a higher frequency of deadlocks.

As a first approach to deadlock resolution we implemented rollback of the transaction causing the cyclic wait condition, a method which is used in many commercial DBMS. But this proved to be a poor solution, since with 3 reference strings we got permanent cyclic restarts (livelocks) for n > 8. This is to say that a set of 3 to 5 transactions ran into deadlock with each other over and over again with no chance of resolution, even at the end of the string. As explained in sec. 3, we have neglected all pages containing system tables etc. in our synchronization protocols, since these would be candidates for high blocking frequencies. We have also made sure that pages with free storage space, which are requested by transactions executing a STORE-operation, did not cause unnecessary conflicts. But there are, in each database, some schema-dependent "hot spots" (Re82), i.e. records which are frequently modified by many transactions, and these were generally involved when permanent deadlocks occurred. In order to avoid this anomaly (which caused q-values of > 10!) we opted for backing out the cheapest (or youngest, respectively) transaction. But since deadlock victims are back and active very soon - which is a general problem of the reference string - driven evaluation - we found sets of deadlock-susceptible transactions remaining stable over a relatively long period of time even with this method. Again, q was increased dramatically, thus producing meaningless results. The method which proved to be best implies some dynamic load balancing, and works as follows: Each transaction is assigned with a counter containing the number of rollbacks this transaction has already suffered. A deadlock victim is scheduled behind all other transactions that have been found on the reference string up to this moment; so if it is started again, it will hopefully see a different pattern of activities and will not run into deadlock again. If a transaction is started with a rollback counter greater than a predefined tolerance level (we have used 3 throughout the evaluation), then the scheduler will start no new transactions until this critical transaction has left the system - by either finish or rollback. Thus parallelity is sometimes decreased to 1, but it will immediately grow to its old value as soon as the critical transaction is through. With long reference strings and n > 16 such a decrease of concurrency down to 1 has been observed 6-8 times. Decreases caused by critical transactions in other reference strings were less frequent and less drastical. Without this counter measure, the average parallelity n would be somewhat higher, but since q would increase much faster, we would yield a worse overall performance in terms of n*.

The same idea can be applied to optimistic schemes in their forward oriented version (Hä82, PSU82, Sc81, UPS83), too, as will be explained in 5.4.

5.3 Observations on (r,a,c)/(r,a,x)-Schemes

Surprisingly the results obtained showed comparatively small differences between (r,x)-protocols on one hand and the (r,a,c)/(r,a,x)-protocols on the other hand, except for string no. 3 and n > 8 which displays clear advantages of (r,a,c)/(r,a,x) over (r,x). The expected superiority of (r,a,c) over (r,a,x) is also contradicted by some of our data. Though our results in general reveal slight advantages of (r,a,c)/(r,a,x) they are in contrast to (KL82) where simulations based on synthetic reference strings are strongly in favor of the (r,a,x)/(r,a,c)-approach (mixes with 50% read-only transactions generate half as much backups and blocking situations using (r,a,c) than with (r,x)). But this is largely due to the assumptions made about the database size (100 objects) and the conflict potential (10 objects accessed per transaction and 100 transactions concurrently which implies that roughly 40% of the database is locked at any instant during the entire simulation) and the performance measures chosen in (KL82). As we already mentioned, the absolute numbers of transaction backups and blocking situations alone do not adequately reflect under-utilized processor capacity as long as the length of blocking periods is not taken into account.

But why are the values of n so close together in many cases and why are (r,a,c)/(r,a,x) so prone to deadlocks in our evaluations? The reasons are manifold, and due to the complexity of this synchronization protocol we can rather sketch them than explain the problems in detail.

One of the intricacies of the (r,a,c)-protocol is that objects cannot be freed immediately after transaction commit. All objects read or

written by a transaction T remain locked, the modified ones in a special c-(conversion) mode, until all transactions which have seen the old version of any object changed by T have finished, i.e. released their locks, too. Keeping locks after transaction commit causes blocking situations which cannot occur in (r,x)-schemes and our results indicate that the inactive phases can become very long. Figure 2 depicts the length of the inactive phases measured in terms of references processed between commit and finish. In order to make the values comparable over all reference strings the inactive phase's length is expressed in multiples of the average transaction length. These values vary considerably with n and the type of reference string. However for n > 8 the inactive phase is always several times the transaction length and not only a fraction of it.

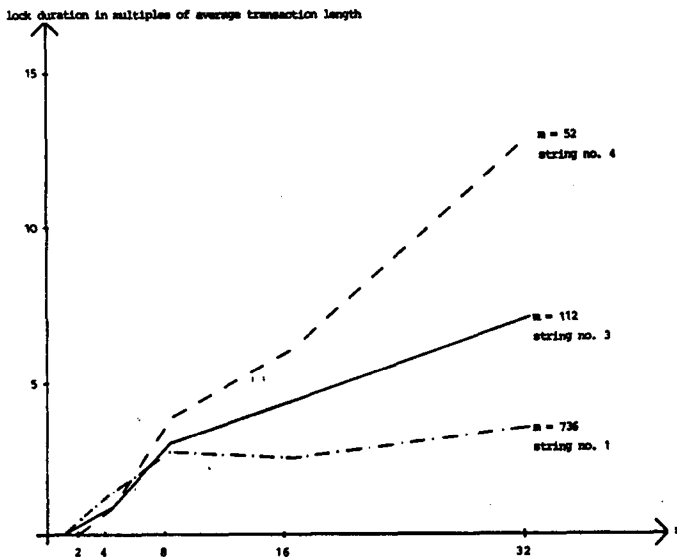

Fig. 3: Duration of c-locks after transaction termination

Another, and very important issue is lock conversion. All simulations of locking protocols (including (KL82)) have implicitly assumed that references to an object are either of the "read" or of the "update"-type. But this is not the case in real systems. In DBMS using procedural DML, as ours, and - at a lower level - in relational systems as well, references very frequently are read-references at first and are converted into update-references later on. Lock conversion turned out to be a major cause of deadlock, as table 9 demonstrates for (r,a,c). In general the (r,a,c)-scheme produced more deadlocks than (r,a,x) which in turn performed worse than (r,x) w.r.t. deadlock frequency. We have identified several typical conversion scenarios resulting in deadlock for (r,a,x) and/or (r,a,c) but not for (r,x). One case is illustrated by the following example. Assume an object O in the database read by several transactions some of which try to convert their locks to exclusive mode or analyze mode respectively. Under the (r,x)-scheme the first transaction T making a conversion request would block subsequent readers (including those which might also convert later on) from accessing O until T has left the system. Provided that the period between setting the read lock and requesting conversion is short, no deadlock will occur. In the two version approach, the first convert operation is admitted immediately since readers are compatible with at most one writer. In contrast to the (r,x)-case subsequent readers will still be admitted. Now the next reader trying to convert will cause a deadlock.
Note that this is only one situation we have found to be characteristic for the behavior of the two types of protocols. There are many other, but such details would exceed the scope of the paper.
However besides the r→x/r→a-conversion inherent to all locking protocols, the (r,a,c)/(r,a,x)-schemes depend on another fundamentally different type of conversion during the commit operation. This conversion into c- or x-mode may reveal conflicts having occurred much earlier, but which have not been detected due to the compatibility of r- and a-locks.
In order to overcome this problem, we have tried a variant of the (r,a,c)-scheme which tests for deadlocks more early than the original one, and in some cases does even delay readers, but the results have not been significantly different. The authors of the (r,a,c)-protocol currently investigate a combination of this locking scheme with a time-stamp mechanism (BEHR82), but evaluation of such strategies based on reference strings will cost some additional work on our side.

5.4 Observations on occ-Schemes

As has been already mentioned in sections 5.1 und 5.2, we encountered severe problems when we tried to guarantee "fairness of processing" especially to long transactions. In particular we had to apply different conflict resolution strategies in order to achieve a sufficiently fair scheduling. This ruled out occ-schemes in their backward oriented version (Hä82) from the beginning, since there the only choice is rolling back the transaction which unsuccessfully tries to validate. Hence all results apply to the forward oriented occ-schemes (focc). If conflicts with parallel

transactions are detected under focc there are three basic policies how to proceed, namely abort and restart the validating transaction, or kill all conflicting transactions, or defer validation until the conflicting transactions have finished. Because the latter policy introduces the possibility of deadlock and thus much additional administrative overhead we did not include it in our simulation experiments. Instead we tried some hybrid policies which favor multiply reset trans-actions. Surprisingly the best results for n* were obtained with unconditionally killing the set of conflicting transactions at the expense of several long transactions being killed over and over again. The superior performance of the pure kill policy is due to the fact that no validation ever fails. In order to solve the livelock problem and enable long transactions to commit with a fairly low number of restarts, we pursued two strategies. On the one hand we incorporated the dynamic load balancing algorithm outlined in section 5.2 in our simulations of the focc-scheme. But note that each kind of dynamic load control applied to occ-schemes in order to support "critical" transactions required some type of dynamic lock-out mechanism for other transactions - which is in contradiction to the original optimistic idea of letting things go. But obviously this step backward improves the protocol by limiting the maximum number of restarts for any transaction.
At first when using the pure kill strategy we observed several transactions being aborted more than 10 times or even 20 times depending on the reference string and the degree of parallelism. These values are not tolerable in a real application environment. So we applied a hybrid policy of the following type: We defined a restart limit for all transactions, and in case of conflict no transaction exceeding this limit must be aborted - unless the validating transaction itself is beyond the limit. With this we could drastically reduce the number of transactions with extremely high restart rates. For instance allowing for up to 5 unsuccessful validations before a transaction was authourized to kill conflicting ones, reduced the number of transactions reset more than 10 times by a factor of 2 compared to unconditional killing. We are not sure, whether all parameters (e.g. tolerance levels for number of rollbacks) have already been adjusted to their optimal values; but this will be investigated in a series of systematic evaluations, in particular to access the tradeoffs between limiting the number of restarts for individual transactions and the resulting loss of performance measured by n*.

6. Conclusions

In this paper we have tried to compare different synchronization mechanisms for multiuser databases on a **realistic** and **unified** basis. We have used real-life page-reference strings from databases with a relevant size rather than random number simulations. It could be shown that this helped to reveal some very important problems - think of lock conversion. Second, we introduced a performance measure n*, which may be considered the **effective** parallelity doing useful work, as a means for comparing the synchronization schemes on a quantitative scale. Using the criterion, we found the classical (r,x)-schemes to perform as well as the improved (r,a,c)/(r,a,x)-schemes in many situations, which is an interesting fact. The average degree of concurrency, $\bar{n}$, is usually smaller with (r,x), but this is outweighed by a smaller q, i.e. less transaction rollback. Optimistic schemes achieve excellent performance figures in all cases, but this evaluation does heavily depend on the (idealized) processor model underlying the definition of n*. This is especially remarkable since the applications have a comparatively high amount of update transactions.
We do clearly realize the problems with our simplistic performance measure n*. It has been introduced for the only purpose of getting the different approaches compared at all. A more realistic comparison must, of course, be based on response time and throughput. But in order to map the events we can observe now (blocking situations, deadlocks, etc.) onto elapsed time and transaction rates, a more detailed model is required. The list of effects it has to comprise looks as follows:

- Implementational aspects of the synchronization protocols: instructions for manipulating control structures, length of critical sections, storage overhead, length of EOT-processing, costs for deadlock detection, etc.
- Realistic scheduling of references: times of transaction de-activation due to communication with the DC-subsystem, wait times due to physical I/O (this requires implementation of a DB-buffer and a log subsystem).
- A processor model: number and speed of physical processors dedicated to DB-processing, type of processor synchronization (via shared memory or via messages), etc.
- A physical DB-model: distribution of the DB-segments over physical devices, association between devices and channels, channels and processors, etc. This is particularly necessary to estimate the degree of **actual**

parallelity of transactions in the presence of I/O-wait times. This parameter is definitely overestimated in our above model.

References

BEHR82 Bayer, R., Elhard, K., Heigert, J., Reise, A.: Dynamic Timestamp Allocation for Transactions in Database Systems, in: Proc. 2nd International Symposium on Distributed Data Bases, Berlin, 1982, pp. 9-20.

BHR80 Bayer, R., Heller, Reiser, A.: Parallelism and Recovery in Database Systems, in: ACM TODS, Vol. 5, No. 2, Juni 1982, pp. 139-156.

BSW80 Bernstein, P.A., Shipman, D.W., Wong, W.S.: Formal Aspects of Serializability in Database Concurrency Control, in: IEEE Transactions on Software Engineering, Vol. SE-5, 3 (May 1979), pp. 203-215.

EGLT76 Eswaran, K.P., Gray, J.N., Lorie, R.A., Traiger, I.L.: The Notions of Consistency and Predicate Locks in a Database System, in: CACM, Vol. 19, No. 11, November 1976, pp. 624-633.

EH82 Effelsberg, W., Härder, T.: Principles of Database Buffer Management, Research Report, No. 51/82, University of Kaiserslautern, 1982.

GLPT76 Gray, J.N., Lorie, R.A., Putzolu, G.R., Traiger, I.L.: Granularity of Locks and Degrees of Consistency in a Shared Data Base, in: Modelling in Data Base Management Systems, G.M. Nijssen (Ed.), Elsevier North-Holland Inc., New York, 1976, pp. 365-394.

Gr78 Gray, J.N.: Notes on Data Base Operating Systems, in: Lecture Notes in Computer Science 60, pp. 394-481, Springer Verlag, Berlin 1978.

Hä82 Härder, Th.: Observations on Optimistic Concurrency Control Schemes, IBM Research Report RH 3645 (42501), 10/15/82, San Jose, 1982.

KL82 Kiessling, W., Landherr, G.: A Quantitative Comparison of Lockprotocols for Centralized Databases, Research Report, Sonderforschungsbereich Programmiertechnik, TU München, 1982.

KR81 Kung, H.T., Robinson, J.T.: On Optimistic Methods for Concurrency Control, in: ACM TODS, Vol. 6, No. 2, June 1981, pp. 213-226.

MN82 Menasce, D.A., Nakanishi, T.: Optimistic vs. Pessimistic Concurrency Control Mechanisms in Database Management Systems, in: Information Systems, Vol. 7, No. 1, pp. 13-27, 1982.

PSU82 Prädel, U., Schlageter, G., Unland, R.: Einige Verbesserungen optimistischer Sperrverfahren, in: Proc. GI-Jahrestagung, 1982, pp. 684-698 (in German).

Re82 Reuter, A.: Concurrency on High-Traffic Data Elements, in: Proc. 1982, Conf. on Principles of Database Systems, 1982, Los Angeles.

RS77 Ries, D.R., Stonebraker, M.: Effects of Locking Granularity in a Database Management System, in: ACM TODS, Vol. 2, No. 3, September 1977, pp. 233-246, University of California-Berkeley.

RS79 Ries, D.R., Stonebraker, M.R.: Locking Granularity Revisited, in: ACM TODS, Vol. 4, No. 2 (June 1979), pp. 210-227.

Sc81 Schlageter, G.: Optimistic Methods for Concurrency Control in Distributed Database Systems, in: Proc. VLDB, 1981, Cannes, pp. 125-130.

UPS83 Unland, R., Prädel, U., Schlageter, G.: Ideas on Optimistic Concurrency Control II: Design Alternatives for Optimistic Concurrency Control Schemes, Technical Report, Univ. Hagen, 1983.