Inform. Systems Vol. 12, No. 1, pp. 83–98, 1987
Printed in Great Britain

0306-4379/87 $3.00 + 0.00
Pergamon Journals Ltd

EVALUATION OF A MULTIPLE VERSION SCHEME FOR CONCURRENCY CONTROL

THEO HÄRDER
University of Kaiserslautern, West Germany

and

ERWIN PETRY
ETH Zürich, Switzerland

(*Received* 16 *February* 1986; *in revised form* 17 *September* 1986)

Abstract—This paper presents a thorough investigation of all relevant properties of a multiple version scheme for concurrency control. It offers conflict-free scheduling for reader transactions thereby generally reducing resource contention. By using a trace-driven simulation model we explored the effective parallelism achievable in multi-user database systems and the number of occurring deadlocks as a complementary measure. The workload was represented by six real-life object reference strings of different applications running on databases which vary in size between 60 MB and 2.9 GB. To compare and valuate the outcome of our experiments we used known results for RX-, RAC- and optimistic synchronization protocols derived under the same conditions. Furthermore, version-dependent criteria and their influence on general database performance were considered. The results obtained underline the value of the multiple version concept for concurrency control; many criteria vote for its use in future database systems with highly concurrent transactions.

1. INTRODUCTION

During the recent years quite a variety of concurrency control methods was proposed in literature [1–4]. While locking with two-phase RX-protocols was the synchronization method 10 yr ago, we have now the selection of a broad range of algorithms based on extended locking protocols [5, 6], timestamps [7, 8], versions [9–11], the optimistic idea [12–15] and their combinations [1] when an appropriate DBMS concurrency control (CC) has to be designed. Although probably more than 100 CC-algorithms can be composed by skilfully combining the known ideas, there is very little information available on their performance, even for the main concepts. Performance evaluation and comparison, however, are prerequisites for a thorough design decision.

After more than 5 yr of learning we are convinced by our experience that we will not come to a quick conclusion on all methods, system structures and applications. Why are evaluation and comparison of CC-methods so difficult? This is essentially due to the following:

—There are numerous parameters to consider, mainly concerning arrival time, parallelism, application and database characteristics.
—Analytic modeling does not seem to be reliable and accurate because of numerous simplifications to keep the formulae tractable.
—Simulation models driven by synthetic workloads usually do not capture all essential database and application parameters, e.g. size and reference pattern of the database including hot spots, etc.
—There are no accepted measures for comparison; is, for example, an increase of parallelism or a decrease of deadlock frequency more important?

Of course, some preliminary studies and results on some CC-methods are already known [16–19]. However, these results are not conclusive (and hard to compare) in various ways. There is no unanimous recommendation of a CC-algorithm neither within the framework of an experimental study nor among the various investigations.

To determine the performance of CC-algorithms, we soon abandoned the idea of pure analytic modeling and also simulation based on random numbers. We refined our models to use trace-driven simulation. The workload was modelled by traces of object references collected during the execution of transactions on a real DBMS. A sample of different object reference strings served for an accurate description of a wide spectrum of applications during simulation. With this broader basis, we hope to derive more precise and more reliable results. Some aspects of our quantitative analysis are reported in [20, 21].

Here we want to investigate the application of implicit versions [11] to enhance concurrency in centralized multi-user DBMS. After outlining the idea of version use for CC, we mainly concentrate on its performance evaluation by describing our simulation environment, performance criteria and results. To compare its benefits we refer to results gained for other CC-methods [20].

2. CONCURRENCY CONTROL USING VERSIONS

Versions are used in databases in various ways and with different meaning e.g. for time mapping (temporal or CAD databases) or for synchronization. In temporal databases, versions of an object are explicitly made visible to the user to refer to specific points or intervals in time. An implicit versioning concept is employed to reduce contention during parallel database access and to enhance concurrency, thereby hiding the notion of versions at the user interface. Of course, the use of a combination of both concepts is also conceivable.

Here we are, however, only interested in the evaluation of the implicit versioning concept and its potential gain for the performance of concurrency control. Therefore, our method does not reflect the permanent storage and the temporal access of versions. We rather apply temporary versions being discarded when no transaction is accessing them any more.

2.1. *CC with two versions*

A scheme using up to two versions of an object was proposed in [5]. This so-called RAC-scheme allows several transactions to read an object while a writer transaction creates a new version of that object. As soon as the new version is committed, it can be referenced by (new) transactions. Transactions having read the old version must see this version during their lifetime prohibiting the creation of the next version by a waiting writer. Therefore, long readers can delay a writer for a long time in a two-version scheme.

To eliminate such read–write conflicts, the version concept can be enhanced to exploit multiple old versions of a data object [11]. Here we only sketch the main idea of the algorithm, before we focus on its evaluation.

2.2. *CC with multiple implicit versions*

We need a strict separation of readers and writers; the type of a transaction must be declared at begin of transaction (BOT). A transaction is a reader if it only issues read operations and a writer if it references (among possible read operations) at least one object of the database with the intention to change it.

Reader transactions. On any demand, a reader T_i always gets that version of the referenced object having been the current one at BOT(T_i). T_i needs not issue any synchronization request and no synchronization measure is applied to the referenced object. The multiple version concept guarantees that readers always see the consistent state of the database at their BOT. The consequence is that readers always can be served immediately, are never involved in deadlock situations and hence can always terminate correctly.

Writer transactions. A writer competes in all its read and update operations for the latest versions of database objects with all concurrent writers. Therefore, writers must be synchronized. This demand may be achieved by any known CC-method. However, optimistic methods do not seem to be favorable because of a relatively high risk of conflicts, resulting from the fact that all transactions involved are updating the database. Hence, it seems adequate to use the conventional two-phase RX-protocol among writer transactions. While changing an object A from version A_i to A_{i+1}, a writer is working on a copy of A_i thereby keeping the current version A_i available for readers.

The parallelism of the version scheme is illustrated in Fig. 1. Because readers always see the DB-state which was current at their start, they can be serialized at their BOT. Writers are assigned to the serialization order at EOT as usual. Therefore, the resulting serialization order of the schedule in Fig. 1 is $T2$, $T1$, $T4$, $T3$, $T5$, that is, $T2$, for example, must not see the modifications of $T1$ because of BOT($T2$) < EOT($T1$).

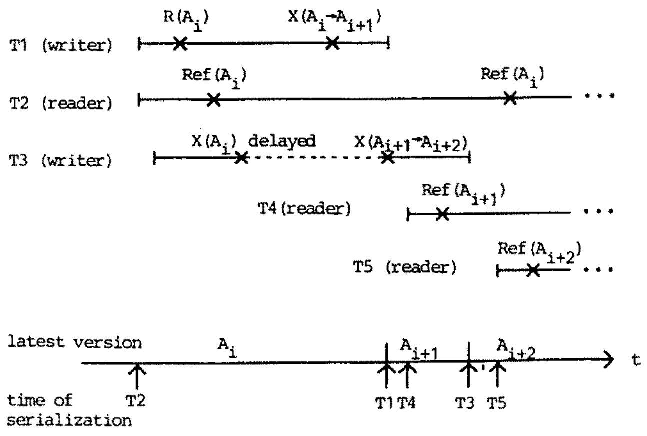

Fig. 1. Concurrent reader (Ref) and writer (R, X) references to different versions of object A and their serialization order.

It is obvious that fewer conflicts are provoked when only writers and no readers must be synchronized. This potential gain can be utilized in various ways:

—The reduction of conflicts for the same throughput compared to RX-protocols is used to greatly improve response time.
—The parallelism can be further increased to maximize throughput without aiming at minimal response times. More readers cost only the overhead of a higher multi-programming level, but they imply no additional locking conflicts.
—Deadlocks are expected to be more rare events as long as the writer parallelism is less than that of a conventional RX-scheme.

Certainly, a definite disadvantage can also be observed from Fig. 1. Transaction *T*2 gets a picture of the database which becomes more and more obsolete relative to the latest version of an object (A_i instead of A_{i+2}). This phenomenon of out-of-date object references seems to be a principal disadvantage and may cause some problems (peculiarities) with long reader transactions.

3. IMPLEMENTATION OF A MULTIPLE VERSION SCHEME

To be able to always give to readers any object having been in the database at their BOT, we must keep in some way the then actual state of the database. Because writers may produce new versions of the same object while other readers are still active, the system has to manage potentially many versions of the same object.

Typically, object versions are kept at the page level. As illustrated in Fig. 2, a viable implementation may be sketched as follows:

—The latest versions are stored in the database.
—The older (temporary) versions are kept in a version pool.
—Versions are chained in reverse chronological order. They contain identifiers and time-stamps of their creating transactions to direct readers always to the database state at their BOT.
—Pages of the database and of the version pool may be fetched into the system buffer for efficient reference.

Each page is marked with a time-stamp indicating EOT-time of the creating transaction. Since the pages (versions) of an object *A* are chained according to their temporal commit sequence, object access is at least conceptually easy. A writer transaction T_i always gets the top page (latest version in the database); page updates are prepared in a private buffer before they replace the current versions in the database during EOT-processing. Successful commit implies pushing the replaced pages of T_i (at least logically) to the version pool in an atomic manner. Of course, there are protocols to achieve this kind of time-critical page propagation to the system buffer for performance reasons; their actual replacement and transfer to the version pool may then be decided by some replacement algorithm when room for new pages has to be prepared.

A reader transaction *Tj* gets automatically the youngest version of *A* older than BOT(*Tj*) for all its references. Conceptually, starting at the latest version of *A* (top) the chain of versions has to be searched to find the appropriate version of *A*. A real implementation may either provide sequential search or accelerate its location by an index or hash mechanism. BOT of the oldest reader *To* determines the time range within which all versions of all objects have to be stored. Although many versions younger than BOT(*To*) may presently not be referenced, they have to be kept if there are active readers with a BOT-time in the range where the version was the current one, since some of these readers may request them later on.

To speed up the access of versions, large system buffers are assumed to be very helpful. Principally, all versions may temporarily or permanently reside in the buffer thereby using the version pool only as a backup store.

As soon as the oldest reader *To* commits, a number of versions can be freed. The next oldest reader *Tno* now limits with its BOT(*Tno*) the range of versions to be kept.

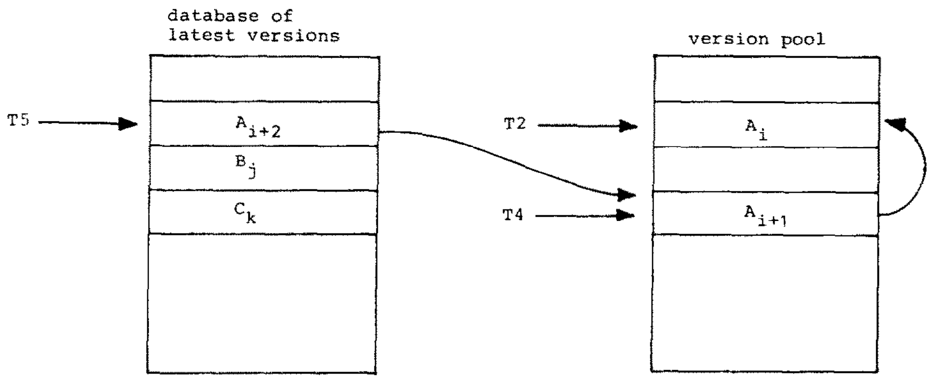

Fig. 2. Basic storage scheme for versions (references to versions of *A*: see Fig. 1).

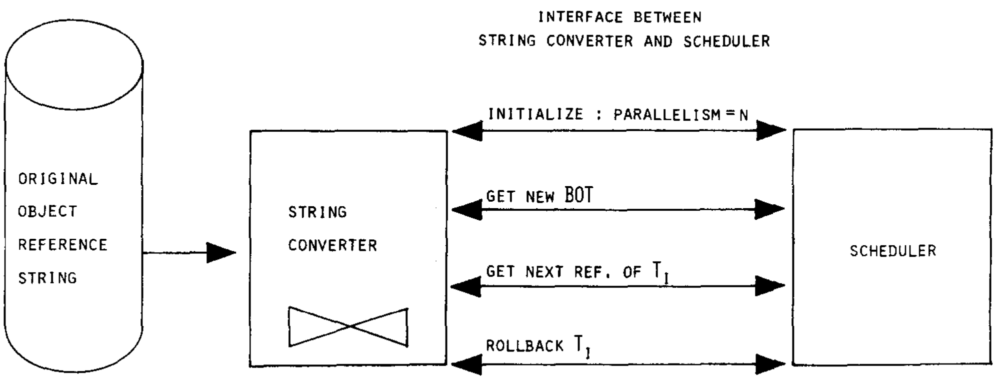
INTERFACE BETWEEN STRING CONVERTER AND SCHEDULER
ORIGINAL OBJECT REFERENCE STRING
STRING CONVERTER
INITIALIZE : PARALLELISM = N
GET NEW BOT
GET NEXT REF. OF T_I
ROLLBACK T_I
SCHEDULER

Fig. 3. Use of a string converter.

Hence, a suitable implementation has to provide a number of quite complex algorithms. The most important task is the effective organization of the version pool:

—To rapidly detemine the appropriate versions for readers;
—To locate versions not referenced any more;
—To perform fast garbage collection to regain space of outdated versions.

Since the details of the implementation are not important for our concept evaluation, we will not further discuss them. The interested reader will find an efficient proposal for the organization of the version pool including recovery management in [11].

4. SIMULATION METHOD

The practical usefulness of any CC-algorithms is strongly dependent on its performance under realistic conditions which we intend to evaluate precisely. The kind of modeling has a far-reaching influence on the accuracy of results. Probabilistic models to represent the transaction load and the referenced data of the database are very imprecise in all DB-applications, because essential aspects like non-uniform transaction reference patterns and object locality are typically not well reflected. Therefore, important DB-phenomena like high traffic data elements and hot spots having stong impact on synchronization and throughput will not be observed during the simulation. With this expectation, however, the question concerning the practical value of the prediction results must be raised.

Trace-driven simulation guarantees a much higher degree of realism because it does not require any idealization or assumptions necessary when the DBMS and transaction behavior are expressed by random numbers. It permits precise modeling of the relevant DBMS components under realistic conditions, since transaction and data references are derived from real applications. Using such an approach, object reference strings incorporate the workload model consisting of transactions and additionally the reference model for the database.

4.1. Object reference strings and their conversion

The workload model driving the simulation is represented by object reference strings (ORS) which describe the history of object references (e.g. pages) of a DBMS application. For our purpose, the essential contents of an ORS are

—a BOT-record for each transaction describing its type and origin;
—a LOGREF-record for every object reference containing type of a request, object identifier, etc.;
—an UNFIX-record for every object release as a counterpart to a LOGREF-record;
—an EOT-record for each transaction, whether successful or not.

All records in the ORS are time-stamped; they are ordered in temporal sequence of the original event execution in the DBMS and allow for appropriate derivation of synchronization information, that is, lock requests and lock modes in case of locking for example.

An ORS reflects the real load situation, e.g. number of concurrent transactions and their scheduling, of an execution interval of the corresponding DBMS application. Hence, driving a sychronization subsystem with the very ORS in recording sequence would result in exactly one simulation run—a poor outcome of the modeling efforts. Therefore, the possibilities of simulation are greatly enhanced if an ORS can be transformed in a parameterized way to generate load situations of arbitrary levels of concurrency. This is achieved by the string converter, as illustrated in Fig. 3. Upon initialization it can display the ORS with any number of transactions executing in parallel. The scheduler can activate new transactions (BOT) up to the declared level of parallelism. For active transactions, references may be requested according to scheduling decisions, wait situations, etc. They are delivered in their logical recording sequence within a transaction. An EOT-reference will usually cause the scheduler to request a new BOT-record after having performed all commit actions depending on the synchronization protocol. Transactions that have to be rolled back because of some

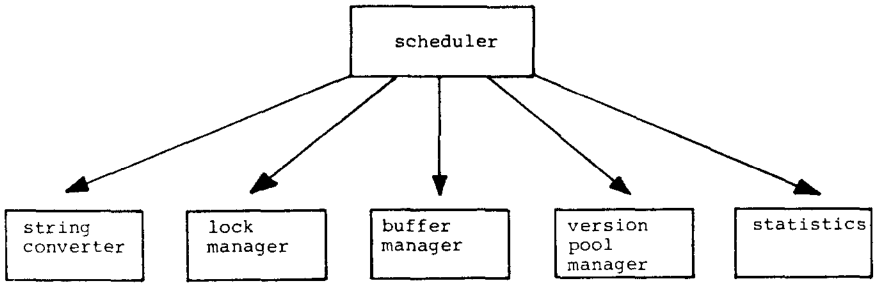

Fig. 4. Overview of the gross achitecture; static calling hierarchy.

failure, e.g. deadlock, can be returned to the string converter to be restarted at a later point in time. To support greater flexibility in simulating synchronization protocols, the scheduler is also able to dynamically decrease the level of parallelism under difficult scheduling situations, e.g. repeated deadlock of a transaction.

4.2. Functions and structure of the simulation system

For investigating the performance of CC-schemes, we designed a simulation system in PL/I. Its gross architecture is shown in Fig. 4. The synchronization algorithm is explicitly coded and all important components are represented with their essential features. Simulating the implicit version scheme requires the following components:

—*Scheduler.* It is the central component serving for system intialization, transaction activation and control, resource multiplexing and failure handling. Scheduling of transaction requests is done in a round-robin-manner subject to delays of blocked transactions. They are reactivated by the scheduler as soon as the lock manager grants their lock request, or they are rolled back when a deadlock is encountered.

—*String converter.* It is transforming an ORS and delivers the references according to the specified load model (as described above).

—*Lock manager.* It manages all requested locks in a lock table organized as a hash structure. It checks for deadlocks by using a cycle search in a wait-for graph.

—*Buffer manager.* It controls the system buffer and manages the references to the database. Page replacement considers locality of actual and old versions.

—*Version pool manager.* It controls the version pool organized as a ring buffer and accesses the appropriate versions for readers. Garbage collection is necessary to free occupied space not needed any more. In case of threatening pool overflow, space will be reclaimed by aborting the oldest reader transaction.

—*Statistics.* A print module collects the events, computes various statistics and outputs readable lists with the results of a simulation run. Statistics are available on frequency and cause of deadlocks, blocking situations, number of restarts, etc. Furthermore, for each nominal parallelism the so-called effective parallelism roughly equivalent to throughput is obtained.

These modules implement a number of functions whose services can be requested by subroutine calls. The system structure of Fig. 4 can be easily accommodated to the simulation of other types of synchronization protocols.

Essential parameters initializing every simulation run are the following:

—parallelism (n): describes the maximal number of concurrent transactions;

—system buffer size;

—LIMIT: maximal number of restarts of the same transaction (in case of deadlocks) before special scheduling measures are used to avoid cyclic restart;

—synchronization dependent characteristics and, of course, the workload (ORS, transaction mixes (*TA*-mixes).

4.3. Workload for the simulation

Most important for the quality and significance of results is the workload of the simulation model. As explained in Section 1, our load model consists of reference strings of real DBMS applications. We selected a wide variety of ORS derived from applications of the DBMS UDS [32], which cover a broad range of applications. Their general properties are summarized in Table 1. These overall properties do not characterize all aspects of the workload necessary

Table 1. General characteristics of the object reference strings

ORS / Properties	*TA*-mix 1	*TA*-mix 2	*TA*-mix 3	*TA*-mix 4	*TA*-mix 5	*TA*-mix 6
Size of DB	60 MB	60 MB	150 MB	2.9 GB	280 MB	560 MB
Total no. of *TA*s	262	39	669	2014	860	2288
Total no. of logical references	109216	79750	40751	57959	54465	9862
Writers in %	94.6	12.8	46.0	13.2	33.5	45.6

Table 2. Classification scheme for transactions described by object reference strings

	TA-mix 1		*TA*-mix 2	
Type	RS	WS	RS	WS
*T*1	1–50	0	1–50	0
*T*2	51–100	0	51–100	0
*T*3	$\geqslant 101$	0	$\geqslant 101$	0
*T*4	6–70	1–4	6–70	1–4
*T*5	71–110	1–4	71–110	1–4
*T*6	$\geqslant 111$	5–100	0	0

	TA-mix 3		*TA*-mix 4	
Type	RS	WS	RS	WS
*T*1	1–5	0	1–5	0
*T*2	6–20	0	6–10	0
*T*3	$\geqslant 21$	0	$\geqslant 11$	0
*T*4	1–3	1	1–10	1–4
*T*5	4–10	2–5	11–30	5–13
*T*6	$\geqslant 11$	$\geqslant 5$	$\geqslant 31$	$\geqslant 14$

	TA-mix 5		*TA*-mix 6	
Type	RS	WS	RS	WS
*T*1	1–5	0	1–5	0
*T*2	6–10	0	$\geqslant 6$	0
*T*3	$\geqslant 11$	0	0	0
*T*4	1–20	1–9	1–5	1–2
*T*5	21–30	10–14	6–10	3
*T*6	$\geqslant 31$	$\geqslant 15$	$\geqslant 11$	$\geqslant 4$

Table 3. Transaction type distribution

	TA-mix 1			*TA*-mix 2		
Type	# *TA*	%*TA*	%LOGREF	# *TA*	%*TA*	%LOGREF
*T*1	0	0.0	0.0	0	0.0	0.0
*T*2	4	1.5	11.4	13	33.3	57.8
*T*3	10	3.8	14.7	21	53.8	40.0
*T*4	170	64.9	4.1	3	7.7	0.1
*T*5	74	28.2	59.5	2	5.1	2.0
*T*6	4	1.5	10.2	0	0.0	0.0

	TA-mix 3			*TA*-mix 4		
Type	# *TA*	%*TA*	%LOGREF	# *TA*	%*TA*	%LOGREF
*T*1	119	17.8	1.1	1295	64.3	11.6
*T*2	153	22.9	19.0	311	15.4	7.9
*T*3	89	13.3	53.4	142	7.0	50.6
*T*4	188	28.1	15.6	57	2.8	3.6
*T*5	65	9.7	6.2	101	5.1	7.6
*T*6	55	8.2	18.7	108	5.4	18.7

	TA-mix 5			*TA*-mix 6		
Type	# *TA*	%*TA*	%LOGREF	# *TA*	%*TA*	%LOGREF
*T*1	225	26.3	3.4	903	29.5	17.8
*T*2	303	35.2	6.3	341	14.9	17.8
T3	43	5.0	41.6	0	0.0	0.0
*T*4	160	18.6	14.5	763	33.3	26.9
*T*5	54	6.3	10.3	100	4.4	9.5
*T*6	74	8.6	23.9	181	7.9	28.0

to understand and interpret the behavior of the CC-algorithms. Therefore, we are going to describe our workload in greater detail. Since there is no commonly accepted scheme available, an appropriate *ad hoc* classification of transaction types is formed regarding the sizes of their read sets (RS) and write sets (WS). Depending on the RS-size we distinguish between short, average and long readers (types *T*1–*T*3) and vice versa between short, average and long writers considering the RS- and WS-sizes (types *T*4–*T*6). As an example, a transaction of *TA*-mix 1 is of type *T*4 (short writer) if it issues between 6 and 70 read references and between 1 and 4 references with the intention to modify. Table 2 shows our type classification for the six transaction mixes. With these specifications we obtain the type distribution of transactions and their frequencies as our most refined description of the workload (Table 3). Locality of reference is also an important issue for CC. To roughly characterize the potential locality of transaction processing, the reference frequency of pages is illustrated in Table 4. As indicated by the reference frequency, a number of pages exhibit some kind of high traffic behavior. Of course, the frequency of reference is only an approximate indicator of locality, since the relative distance of rereferences is not regarded. On the other side, more accurate description of locality would require another set of parameters making the workload characterization even more complex. Therefore, we confine ourselves with a very simple *ad hoc* characterization of locality revealing only the differences of the rereference behavior among the various *TA*-mixes. Here, a page is said to exhibit LOCALITY if it is referenced more than 8 times.

Tables 1–4 reveal substantial differences in all respects. The first two transaction mixes have been traced in a scientific database application with a large share of batch transactions; some transactions are very long, since they either have to check complex integrity constraints, or have to read large portions of the (relatively small) database. Both mixes show a high degree of locality ($\geqslant 30\%$ of pages). While *TA*-mix 1 contains a high percentage of short update transactions together with a few average and long readers, *TA*-mix 2 incorporates a very large fraction

Table 4. Reference frequency of pages

Ref. frequency of pages	*TA*-mix 1	*TA*-mix 2	*TA*-mix 3	*TA*-mix 4	*TA*-mix 5	*TA*-mix 6
1	264	337	838	3443	2573	906
2–3	775	871	779	1947	1200	785
4–8	1331	1727	631	1098	1132	347
9–20	786	1802	277	393	485	69
21–50	98	163	138	285	276	5
51–200	55	30	134	255	111	11
> 200	73	56	17	4	14	13
Total no. of different referenced pages	3382	4986	2814	7395	5791	2136
Locality	30%	41%	20%	12%	15%	5%

of average and long readers competing with a relatively small number of short and average writers.

The four remaining transaction mixes originate from different interactive DB/DC-environments with mainly short transactions; in order to discriminate different transaction types we have changed the respective RS- and WS-sizes (see Table 2). *TA*-mixes 3 and 5 run on a medium size database. *TA*-mix 3 has a substantial share of writers ($> 45\%$) and also a number of relatively long readers. *TA*-mix 5 contains a similar structure ($> 33\%$ writers). Both mixes provide an average degree of locality and hide a great potential of mutual hindrances.

TA-mixes 4 and 6 have been recorded on quite large databases with more than 2000 transactions in each string. While *TA*-mix 4 is composed of a small fraction of writers and many short readers, *TA*-mix 6 consists of many extremely short transactions with a relatively high percentage of writers ($> 45\%$). Both exhibit very low locality. Because of size of database and structure of transactions it is assumed that both mixes are mainly conflict-free.

5. PERFORMANCE OF CC-ALGORITHMS

Using the described simulation system and the empirical reference data of various classes of applications, we want to learn as much as possible about the practical behavior of the multiple version concept (MVC) and about the conditions of its appropriate or unfavorable use. Therefore, our goal is its thorough performance evaluation. But what are the measures for CC-algorithms?

5.1. Criteria for performance comparison

There are different approaches to this problem in the literature—a synthetic response time parameter in [23], the number of blocking situations and the number of transactions in [17], etc. It is argued in [20] that these measures may be favorable for certain synchronization protocols, but unsuitable for others. Peinl and Reuter [20] propose a more general measure which can be applied to any CC-method. Hence, we will follow their proposal—also for the sake of comparing their results with ours.

The rationale of the quality measure is as follows:

For each simulation run, the maximum number of parallel transactions, n, is specified. Although the scheduler tries to keep n transactions permanently running in parallel, the current number of active transactions, CP_i, will often be lower caused by blocking situations. The CP_i-values have to be determined appropriately, e.g. in k equidistant observations. Here, the current parallelism was measured after every logical reference actually executed ($k = \#$ LOGREFACT). Hence, the average parallelism is yielded by

$$\bar{n} = \sum_{i=1}^{k} CP_i/k.$$

Since $\bar{n}$ contains the average number of active transactions, it reflects in some way the length of blocking situations. Due to rollback and re-execution of transactions, there are actually more ORS-references to execute in a simulation run ($\#$ LOGREFACT) compared to the references in the original string ($\#$ LOGREFMIN). We obtain a relative increase of references

$$q = \# \text{LOGREFACT}/\# \text{LOGREFMIN},$$

which is independent of the size of the string. This repetition factor reflects the overhead of backups or work to be done twice for a given CC-method.

Based on $\bar{n}$ and $q (q \geqslant 1)$, a single measure is proposed in [20] for the effective parallelism:

$$n^* = \bar{n}/q.$$

Since a certain amount of the n transactions running in parallel is re-executing aborted transactions instead of doing useful work, this fact is taken into account by the calculation of n^*. Hence, the effective parallelism n^* can be used directly as a relative measure of transaction throughput.

5.2. Empirical results of the MVC-protocol

First, we present the results of the MVC-protocol which are derived under the following premises:

—The executed schedules guarantee serializability of transactions.

—A protocol equivalent to consistency level 3 is observed [24] (long R- and X-locks to warrant the prevention of inconsistent analysis, lost update, etc.).

Technical parameters of all simulation runs were

—round-robin scheduling;

—LRU-replacement in a system buffer of 256 pages (it turned out that variations of buffer size had very little impact on the results);

—version pool size large enough to contain all version pages (< 3000);

—buffer management with STEAL and FORCE using (a simulation of) ATOMIC propagation [27];

—LIMIT = 3;

—the ORS using *TA*-mixes 1–6;

—the maximum parallelism n, which was evaluated in the range 2–32.

These premises and parameters were also necessary for the intended comparability of the results.

Table 5 displays the MVC-performance figures for our six transaction mixes. Let us now crudely interpret our first impression of the results. Without doubt, *TA*-mix 2 delivers the best results. This is due to the large fraction of readers and a small number of short and average writers. MVC seems to be tailored to this combination of workload. In contrast to that it is very susceptible to high update rates with strong hindrances. As shown with *TA*-mix 1, MVC

Table 5. Performance figures of the MVC-protocol

	TA-mix 1			*TA*-mix 2		
n	$\bar{n}$	q	n^*	$\bar{n}$	q	n^*
2	1.90	1.01	1.89	2.00	1.00	2.00
4	3.30	1.21	2.72	4.00	1.00	4.00
8	5.71	1.56	3.67	8.00	1.00	8.00
16	9.64	1.86	5.19	16.00	1.00	16.00
32	18.98	1.95	9.75	—	—	—

	TA-mix 3			*TA*-mix 4		
n	$\bar{n}$	q	n^*	$\bar{n}$	q	n^*
2	1.99	1.00	1.99	2.00	1.00	2.00
4	3.92	1.00	3.92	3.98	1.00	3.98
8	7.59	1.00	7.59	7.84	1.00	7.81
16	14.91	1.03	14.50	14.50	1.02	14.15
32	27.65	1.02	27.04	21.03	1.06	19.92

	TA-mix 5			*TA*-mix 6		
n	$\bar{n}$	q	n^*	$\bar{n}$	q	n^*
2	1.99	1.00	1.99	2.00	1.01	1.98
4	3.79	1.01	3.74	3.95	1.02	3.89
8	6.21	1.04	5.99	7.64	1.05	7.27
16	9.20	1.05	8.74	13.94	1.15	12.17
32	14.54	1.09	13.38	24.66	1.27	19.47

cannot do much with such mixes. The low values for $\bar{n}$ at higher degrees of parallelism ($\bar{n} \sim n/2$) are actually lowered for n^* by a factor of ~ 2 due to re-processing. It is expected that also other algorithms do not like this kind of workload and that the obtained results will be close to those of RX-protocols. Surprisingly well does *TA*-mix 3, although it contains 46% writers. An explanation can be found in the share of references. The relatively long readers performing about 75% of the references do not hurt, and the writers with the remaining share of references must be short and only weakly overlapping. *TA*-mixes 4 and 5 also obtain a very low q. The moderate $\bar{n}$ and n^* values at high degrees of parallelism (32) must be provoked by writer conflicts and blocking situations (not leading to deadlocks), e.g. due to high traffic data elements. *TA*-mix 5 has a larger fraction of writers which are responsible for the low effective parallelism ($n^* = 13.38$ for $n = 32$). Even with extremely short writers, *TA*-mix 6 does not so well as expected (at least for $n = 32$). With increasing parallelism, $\bar{n}$ shrinks and q grows more than proportional. This effect of increased blocking and re-processing is clearly revealed by the quotient n^*/n which has the values 0.91 and 0.61 for $n = 8$ and $n = 32$. It is obvious that our hypothesis drawn from the static workload characterisation in 4.3 expecting conflict-free workloads for *TA*-mixes 4 and 6 does not hold. Dynamic synchronization and their interpretation is much more difficult.

5.3. Analysis of deadlocks

MVC uses locks for writers and, hence, is a pessimistic approach. We distinguish two sources of deadlocks:

—R- and X-requests of different transactions cause a cyclic wait situation (CW) blocking all participating transactions permanently.

—Lock conversion is also an important issue. Often, simulations of locking protocols tacitly assume that all objects are either referenced in R- or in X-mode. In real systems, however, as pointed out by our ORS, access to an object is very frequently performed by issuing a read-reference first and then converting it to an update reference later on. Such lock conversions (LC) may result in a deadlock, too, as explained by [20] in detail. Running level 2 consistency, some of these deadlocks will be avoided, however, by risking "inconsistent analysis" or similar phenomena for the concerned transactions.

Table 6 gives a summary of the number of deadlocks occurred where their causes—cyclic wait and lock conversion—are separately considered. Deadlock resolution was achieved by rolling back the transaction causing the situation. When the LIMIT was reached for a particular transaction, the scheduler tried to avoid further rollback of this conflict transaction. It was achieved by temporarily lowering the actual concurrency by retaining new transactions until the conflict transaction has finished. Some experiments in [25] used other cost measures like number of locks held for selecting a rollback victim. The results obtained were very similar to the presented ones. As a side remark, LIMIT = 2 generally produced a slightly diminished deadlock rate because then the protocol lowered the concurrency more

Table 6. Rollback frequencies

	TA-mix 1			*TA*-mix 2			*TA*-mix 3		
n	CW	LC	ΣD	CW	LC	ΣD	CW	LC	ΣD
2	7	0	7	0	0	0	0	0	0
4	22	0	22	0	0	0	0	1	1
8	110	8	118	0	0	0	1	0	1
16	214	37	251	0	0	0	3	1	4
32	339	58	397	—	—	—	7	6	13

	TA-mix 4			*TA*-mix 5			*TA*-mix 6		
n	CW	LC	ΣD	CW	LC	ΣD	CW	LC	ΣD
2	0	0	0	0	0	0	0	7	7
4	0	0	0	4	3	7	2	14	16
8	2	0	2	11	13	24	16	42	58
16	17	4	21	21	56	77	60	115	175
32	57	15	72	43	81	124	57	227	284

frequently but resulting in a reduced effective parallelism.

It can be stated that the deadlock frequency grows with the degree of parallelism—as expected. In most mixes the growth seems to be linear. However, the ideal *TA*-mix 2 is deadlock-free. Except for the pathological *TA*-mix 1, MVC gets comparatively small or moderate deadlock rates computed as the number of deadlocks related to the number of transactions to be executed. $n = 32(16)$ yields the following:

TA-mix 1	*TA*-mix 2	*TA*-mix 3	*TA*-mix 4	*TA*-mix 5	*TA*-mix 6
1.515	0	0.019	0.035	0.144	0.124

This kind of characterization reveals that in the *TA*-mixes 3–6 between 2 and 14% of the transactions had to be reprocessed whereas *TA*-mix 1 provoked a rollback of more than 150%. This means that each transaction had 2.5 processing attempts in the average.

5.4. Performance comparison based on empirical results

Our discussion so far commented the results of MVC for themselves and did not focus on the selection decision for a CC-algorithm. Apart from special cases (the ideal *TA*-mix 2), we do not know how good MVC really is. To valuate MVC we must figure out how well other algorithms behave with critical workloads. Throughput characterized by pure factors for effective parallelism and deadlock frequency is not very helpful for the valuation of a CC-method, but in comparison with a number of candidate CC-methods is particularly useful for selecting the appropriate synchronization protocol. The increase of transaction throughput per time unit or, in our case, the relative gain of n^* and the difference in the deadlock rate compared to other algorithms serve as illustrative factors to estimate the quality of CC-methods. Therefore, we refer to known results of three synchronization protocols [20] to compare and interpret our own results. Hence, we quote the

—RX-protocol [24] as a one-version scheme (only latest object state for readers/writers);
—RAC-protocol [5] as a two-version scheme (latest object state and up to one next older state for readers);
—OCC-method [13] (latest object state for readers/writers and, in a sense, up to m private copies for writers where only one of them can be serialized); hence, it is some sort of a two-version scheme.

Our MVC-protocol is a n-version scheme (n arbitrary) providing the latest object state for readers/writers and up to n-1 old versions for readers.

As for MVC, deadlock resolution is done for RX and RAC by rolling back the transaction causing the deadlock and by lowering the degree of concurrency as soon as a transaction reaches the LIMIT.

OCC offers more parameters to observe. In order to get a well-tuned method, a number of investigations were performed analyzing backward oriented (BOCC) and forward-oriented OCC (FOCC). According to our empirical studies [20, 26] it is safe to say that BOCC having less tuning facilities delivers worse results than FOCC. Hence, we concentrate our efforts on FOCC for the synoptical comparison.

Validation conflicts in OCC correspond to deadlocks in locking schemes. Whereas BOCC does not allow any freedom, conflict resolution in FOCC suggests various possibilities:

Pure ABORT. A validating transaction is aborted when a conflict is detected. An immediate restart often caused problems similar to livelocks. Even delayed restarts could sometimes not prevent thrashing situations. This conflict resolution method hides the danger of cyclic restarts that execution may never end. Thus, it is not very useful.

Pure KILL. A validation conflict is resolved in favor of the committing transaction by rolling back all conflicting transactions (KILL). It tends to produce high rollback rates for long transactions, but it will finally end, because the validating transaction always wins.

Hybrid scheme. A transaction is aborted up to LIMIT times, then it is marked golden. For committing a golden transaction, KILL is used. As a supporting scheduling measure, the degree of parallelism is lowered by blocking all but one golden transaction to guarantee the rollback LIMIT for golden transactions.

We select the results of pure KILL and of the hybrid scheme—referenced as FOCC-K and FOCC-H—for our synoptical comparison. For completeness, the figures of the five protocols for n^* and ΣD (number of deadlocks/rollbacks) are summarized in Tables 7 and 8. For convenience, we refer to a graphical synopsis, as shown in Fig. 5.

TA-mix 1 offers a tough nut for all locking algorithms. OCC was applied as the FOCC-variant using a KILL-strategy which assures that a transaction will survive once it has entered its validation phase. Since no blocking situations occur, it is always $\bar{n} = n$. Due to validation conflicts transactions have to be rolled back. With FOCC, these are only in-progress transactions. We do not definitely know whether or not such transactions have processed many references before abort. At a low degree of parallelism a conflicting transaction may run comparatively long before it will be finally aborted, whereas the probability grows with higher parallelism that it will be killed sooner. Since OCC always runs with n (max. concurrency), the highest abort rate can be expected. Our factor q does not fully express this fact because it does not describe the amount of lost work per aborted transaction. Here we got $n^* = 10$ for $n = 16$ implying $q = 1.6$. For this difficult mix, OCC succeeded with a better result despite the high share of short writers. It is obvious that special measures have to be taken for long readers to avoid cyclic restart.

Table 7. Synopsis of performance figures for n^*

mix	*TA*-mix 1					*TA*-mix 2				
n	RX	RAC	FOCC-H	FOCC-K	MVC	RX	RAC	FOCC-H	FOCC-K	MVC
2	1.68	1.57	1.54	1.50	1.89	2.00	2.00	2.00	2.00	2.00
4	2.46	2.16	2.24	2.62	2.72	3.98	4.00	3.95	3.95	4.00
8	3.62	2.91	3.80	6.01	3.67	7.90	7.97	7.27	7.92	8.00
16	5.92	4.97	4.72	10.59	5.19	14.18	15.62	13.65	15.18	16.00
32	10.05	6.18	6.89	16.32	9.75	—	—	—	—	—

mix	*TA*-mix 3					*TA*-mix 4				
n	RX	RAC	FOCC-H	FOCC-K	MVC	RX	RAC	FOCC-H	FOCC-K	MVC
2	1.94	1.99	1.87	1.85	1.99	2.00	2.00	1.96	1.97	2.00
4	3.56	3.71	3.73	3.54	3.92	3.77	3.84	3.45	3.32	3.98
8	5.93	6.68	6.48	6.67	7.59	6.33	6.75	5.25	6.45	7.81
16	8.13	9.93	9.62	10.91	14.50	8.40	8.79	7.51	12.13	14.15
32	11.06	15.15	16.60	18.14	27.04	12.62	14.89	10.89	20.78	19.92

mix	*TA*-mix 5					*TA*-mix 6				
n	RX	RAC	FOCC-H	FOCC-K	MVC	RX	RAC	FOCC-H	FOCC-K	MVC
2	1.94	1.97	1.83	1.66	1.99	1.97	1.97	1.89	1.93	1.98
4	3.69	3.59	2.69	3.25	3.74	3.85	3.78	3.34	3.71	3.89
8	5.62	5.69	3.86	5.75	5.99	6.99	7.04	5.44	6.66	7.27
16	7.87	8.50	5.83	8.64	8.74	11.21	11.54	8.60	11.03	12.17
32	10.71	11.84	9.84	14.78	13.38	18.28	18.29	14.40	15.35	19.47

TA-mix 2 was ideal for MVC. In general, it should not be so easy for the competitor algorithms to cope with the very large fraction of average and long readers which is no problem with MVC because of the multiple versions. Their excellent results seem to point to delightful reference patterns which allow for low-conflict concurrency despite the presence of writers. The outcome of *TA*-mixes 3 and 4 demonstrates a clear advantage of MVC. As it should be expected, the ranking of the effective parallelism (n^*) is determined by the degrees of freedom—from one version for RX to n versions for MVC.

TA-mix 5 produces a comparably low concurrency, slightly better than *TA*-mix 1. The superiority of OCC coincides also here with the poor effective parallelism of the remaining candidates. Maybe, these situations indicate that the sheer "trial and error"-approach of OCC wins even at the cost of excessive victim transactions to be re-processed whenever the other approaches limit their effective parallelism to a low value. The relatively high percentage of writers in *TA*-mix 6 seem to be a handicap for OCC. As for the previous mix, MVC turns out to be clearly the best locking approach.

It may be stated as a preliminary conclusion that MVC has done well for all mixes. Having a dominating share of readers, then such a scheduling capability could be expected. But surprisingly, it also delivers good values for mixes with a considerable fraction of writers (*TA*-mixes 3 and 6).

Table 8. Synopsis of rollback frequencies

mix	*TA*-mix 1					*TA*-mix 2				
n	RX	RAC	FOCC-H	FOCC-K	MVC	RX	RAC	FOCC-H	FOCC-K	MVC
2	2	14	24	16	7	0	0	0	0	0
4	25	76	75	51	22	0	0	1	1	0
8	80	154	137	94	118	0	0	6	1	0
16	175	220	315	198	251	0	0	17	7	0
32	220	206	514	436	397	—	—	—	—	—

mix	*TA*-mix 3					*TA*-mix 4				
n	RX	RAC	FOCC-H	FOCC-K	MVC	RX	RAC	FOCC-H	FOCC-K	MVC
2	0	0	12	11	0	0	1	15	10	0
4	16	2	73	28	1	1	1	82	32	0
8	25	4	121	55	1	15	5	247	111	2
16	33	34	192	112	4	45	146	420	253	21
32	56	78	267	245	13	122	126	518	424	72

mix	*TA*-mix 5					*TA*-mix 6				
n	RX	RAC	FOCC-H	FOCC-K	MVC	RX	RAC	FOCC-H	FOCC-K	MVC
2	2	5	29	12	0	9	14	21	15	7
4	2	25	113	34	7	15	44	87	41	16
8	40	68	280	134	24	59	97	234	127	58
16	85	176	385	307	77	143	227	400	345	175
32	125	199	452	558	124	246	354	530	893	284

As a final argument, FOCC-K must be observed with caution. To underline its negative image as a scheduling strategy tending to thrashing situations, we list the maximum numbers of restarts of a transaction in the various mixes:

TA-mix 1	*TA*-mix 3	*TA*-mix 4	TA-mix 5	TA-mix 6
20	13	16	19	41

6. FURTHER ASPECTS OF MVC

The advantages of MVC are exclusively based on the version pool use. To evaluate its overhead, it is therefore necessary to investigate various aspects of the version pool.

6.1. Version pool size

For each object, a chain of versions must be stored covering the interval of the current point in time to BOT(*To*) (oldest active reader). In Table 9, we have listed two measures of version pool occupancy:

—the maximum number of pages used for versions during the simulation;

—the average pool occupancy for versions.

The number of versions grows strongly when many writers coincide with long readers, as demonstrated with *TA*-mixes 1 and 4 . Either writers or long readers alone cause only small version pool sizes (see *TA*-mixes 2 and 6). The increase of the version pool is also influenced by *n*. After a fast growth with smaller values of *n* it seems to reach a certain saturation for high degrees of parallelism. The average number of versions is often less than half the maximum.

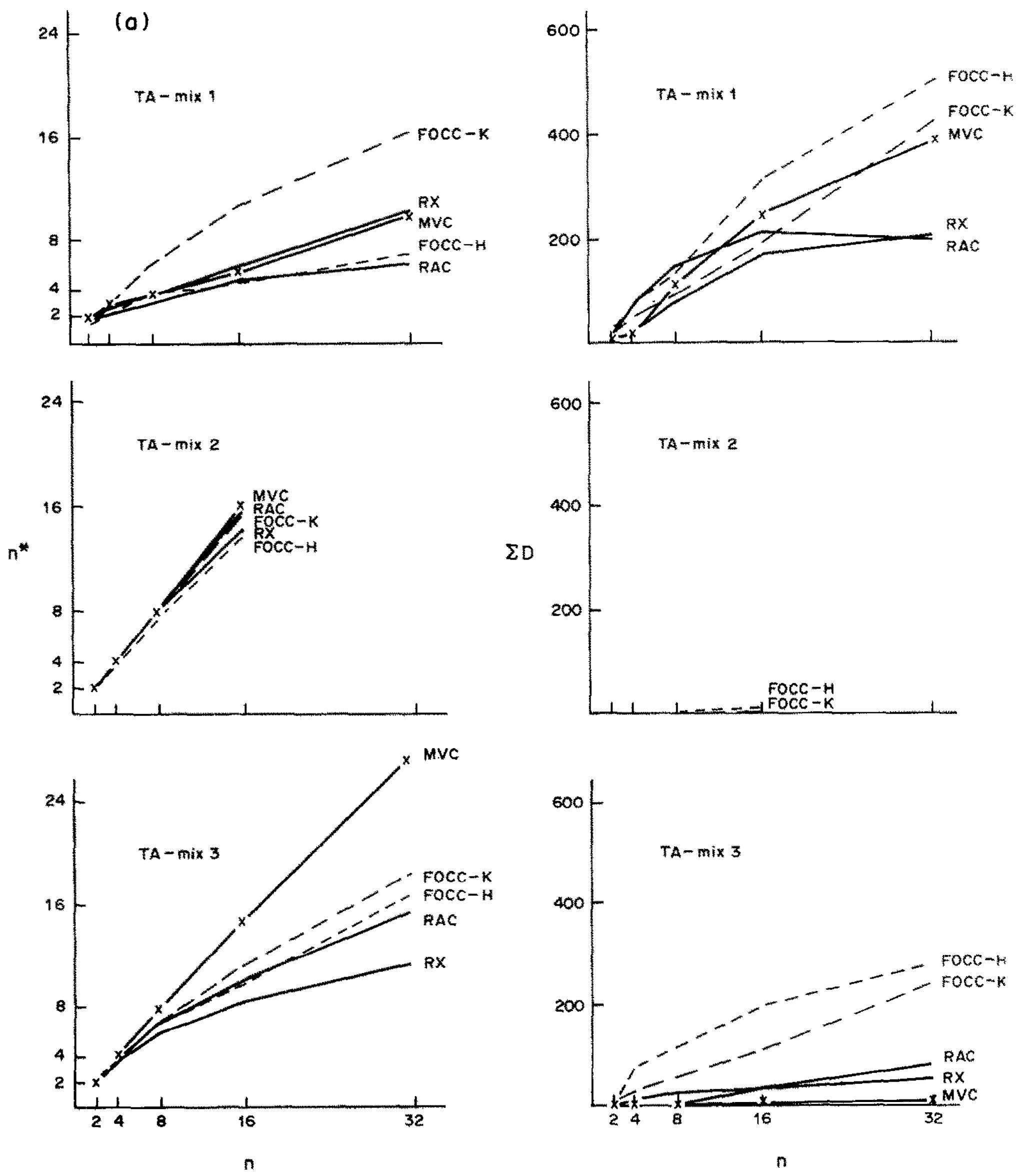

Fig. 5. Synoptical comparison of CC-algorithms.

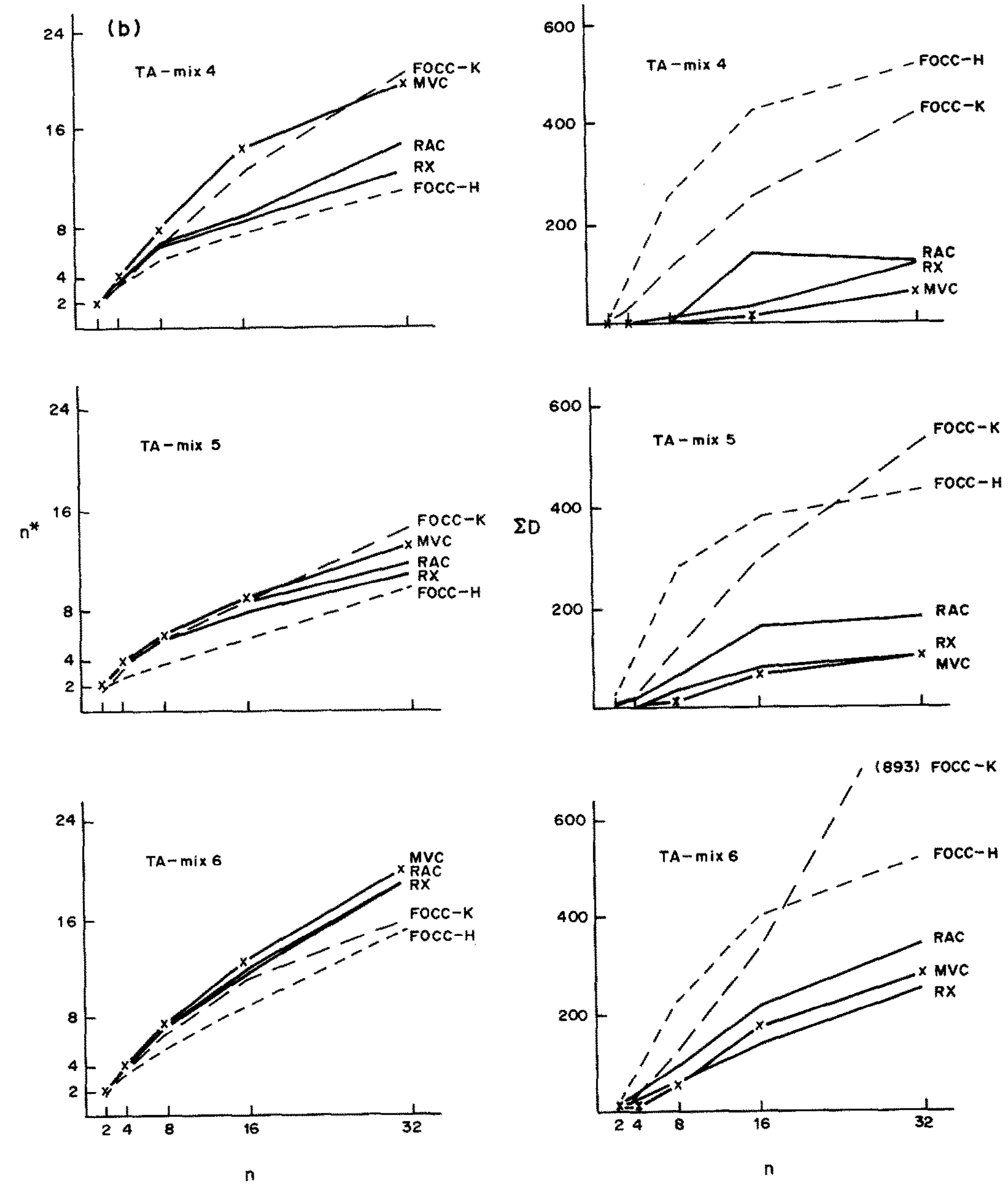

Fig. 5 (Continuation). Synoptical comparison of CC-algorithms.

Table 9. Maximum and average number of versions in the version pool

	TA-mix 1		*TA*-mix 2		*TA*-mix 3	
n	max	av.	max	av.	max	av.
2	502	41.6	8	0.87	47	11.9
4	870	160.4	9	4.37	135	50.8
8	827	159.2	16	11.22	285	134.2
16	989	259.2	25	17.56	707	360.2
32	1057	444.0	—	—	833	468.8
	TA-mix 4		*TA*-mix 5		*TA*-mix 6	
n	max	av.	max	av.	max	av.
2	280	45.6	207	53.7	73	2.21
4	615	200.1	422	115.0	73	5.44
8	1183	539.0	785	276.6	73	17.48
16	2252	1256.9	1244	410.9	94	35.26
32	2589	1461.9	1122	411.2	137	48.37

As already mentioned, a ring buffer implementation is preferable for the version pool. Then, the maximum number of pages needed can be easily controlled by transaction aborts.

6.2. *References to the version pool*

Readers obtain part of their pages from the version pool. To estimate the impact of version pool references on the overall performance, it is necessary to investigate the frequency of such events since each reference may involve several physical I/O to the version pool.

The percentage of pages fetched from the version pool is plotted in Fig. 6. As expected, it enhances with increasing parallelism, but it seems to saturate at a

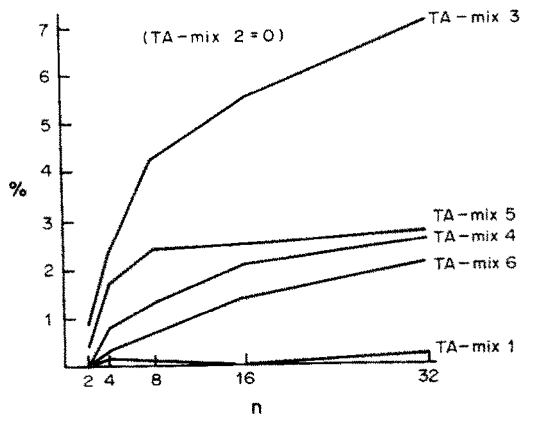

Fig. 6. Percentage of references satisfied by the version pool.

certain *n*. Average values are 0.5–2.5% for *TA*-mixes 4–6. This rather small overhead augments the effective parallelism of MVC substantially compared to the RX-protocol (see Fig. 5). Merely *TA*-mix 3 uses up to 7% version pool references. However, this amplified reference activity does also explain its excellent scheduling results (cf. Fig. 5). On the other hand, the demand for version pool pages is low enough to be effectively optimized by special implementation techniques, for example, by using an extended system buffer or a main memory resident index to limit a version pool reference to one physical I/O.

Now we have answered the question concerning the overhead of the version pool to be carried by the readers. Yet, the versions must be produced before, causing a certain effort for the writers. As already mentioned, the system buffer is managed under a STEAL/FORCE-policy. FORCE as the critical attribute responsible for the overhead means the enforced propagation of all modified pages at EOT, that is, versions are written to the version pool before (STEAL) or not later than EOT [27]. Hence, we get a considerable output overhead for update-intensive mixes, above all in case of short writers. The following output activity expressed as percentage of all LOGREFs was observed:

	TA-mix 1	*TA*-mix 2	*TA*-mix 3	*TA*-mix 4	*TA*-mix 5	*TA*-mix 6
% of LOGREF	1.61	0.05	2.62	5.88	6.96	20.42

These costs, however, cannot be attributed to the version scheme. In any case, a modified page must also be rewitten in a version-free data base (update-in-place). On the other hand, the youngest version of an object saved in the version pool may be taken as its before-image serving as UNDO-information for transaction abort. Therefore, the version use in an integrated recovery scheme incorporates a substantial optimization potential compared to recovery mechanisms based on page logging for UNDO-information.

Again, with the availability of large system buffers a NOSTEAL/NOFORCE-policy can be applied which offers the advantage of relying only on REDO-information. Then, the entire version pool (or at least a considerable part of it) should be integrated in the system buffer avoiding all version-related I/O. In such a scheme, all versions are kept in main memory for their life time; disk storage may be neccessary only in overflow situations.

6.3. *Age of antiquated versions*

The issues discussed so far involve more or less matters of resource use or access overhead. But an important problem has not yet been explored in detail. We have stated that MVC-schedules are serializable; nevertheless, the results of reader transactions exhibit essential differences compared to those of other protocols, since reader sometimes refer to (slightly) antiquated versions. In a sense, up-to-date evaluation is sacrificed for performance. How serious is the problem in practical situations?

The issue of antiquated versions is outlined by the transaction schedule in Fig. 7 which generates the serialization order *Tr*, *T*1, *T*2, *T*3. The view of *Tr* is fixed to the database state before EOT (*T*1) where the then current versions A_0, B_0 and C_0 hold. Since (short) writers permanently change the state of the database, the reader's view gets older and older. The reader *Tr* refers under the MVC-protocol the versions A_0, B_0 and C_0, no matter how long it runs. Note, *Tr* would have got the versions A_1, B_2 and C_3 under an RX-protocol (with the serialization order *T*1, *T*2, *T*3, *Tr*)—a fact that should be considered carefully.

In order to quantify the "antiquation" problem, we analyze age and frequency of version references. Most of the mixes have only a number of references of age 1 and a few of age 2. Only two mixes listed in

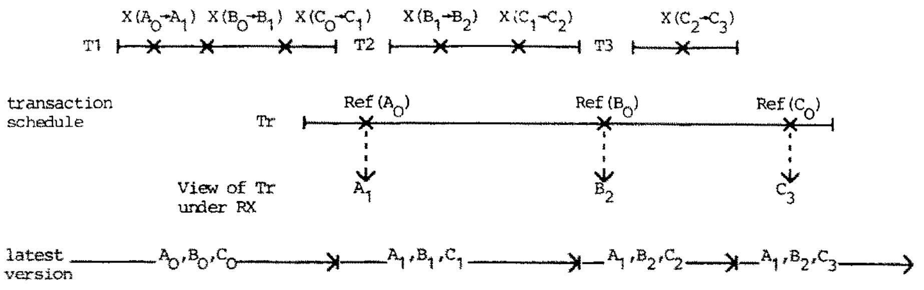

Fig. 7. Reader references to antiquated versions.

Table 10. Number of versions referenced by readers ordered by their age

n	*TA*-mix 3				
age	2	4	8	16	32
0	28,257	27,830	27,296	26,980	26,530
1	255	668	1057	1136	1244
2	0	14	159	364	629
3	0	0	0	32	73
4	0	0	0	0	36

n	*TA*-mix 5				
age	2	4	8	16	32
0	22,485	22,168	22,020	22,009	21,948
1	78	133	171	137	175
2	1	0	26	19	23
3	0	100	15	15	24
4	6	162	123	22	16
5	0	7	215	110	39
6	0	0	0	170	176
7	0	0	0	81	162
8	0	0	0	7	7

Table 10 possess a well-marked reference behavior indicating the seriousness of the problem. References of age 0 are to the latest version whereas those of age *k* lead to antiquated versions whose current version is *k* modifications ahead. The remarkable observation is the quite surprising age 8 of versions—certainly in rare cases—as proved by *TA*-mix 5. Despite such pathological exceptions, the lion's share of reader references is directed to the latest version. Therefore, the "average" age of a version requested by a reader is very low, as illustrated in Fig. 8. For example an average age of 0.1 indicates that on 10 references a reader gets one version of age 1 besides 9 current versions.

7. CONCLUSIONS

In this paper, we have tried to investigate all relevant properties of a multiple version scheme for concurrency control. Prime importance was put on its performance evaluation in a realistic environment. For this task, we designed a trace-driven simulation program and used real-life object reference strings from sizeable databases rather than database references generated by random numbers. The most important performance measures introduced were the effective parallelism, n^*, and the number of deadlocks.

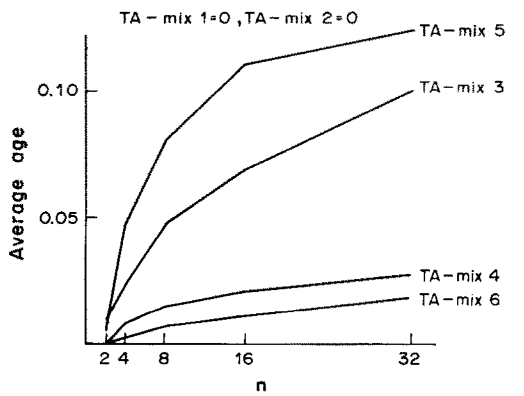

Fig. 8. Average age of versions for readers.

To evaluate the obtained performance figures, we compared them to the corresponding ones of RX-, RAC- and OCC-protocols. A unified basis—the same reference strings and the same configuration and scheduling parameters—supported an accurate valuation of the algorithms. Further performance-relevant aspects such as references to and size of the version pool or the "antiquation" problem were also considered.

The effective parallelism is essentially a relative throughput measure being good for the direct comparison of candidates. It must be weighted with the complementary measure of transaction rollbacks caused by deadlocks/validation conflicts because they have a negative influence on the transaction's response time (and may become an inconvenience for the user). The performance comparison based on our reference strings allows for the following safe conclusions:

—MVC is clearly superior in most mixes neglecting the pathological *TA*-mix 1.

—In cases of a very high percentage of writers (*TA*-mix 1), MVC behaves nearly like the RX-protocol;

—RX, RAC and FOCC-H are distinctly inferior in all cases;

—FOCC-K is competitive in the *TA*-mixes 2, 4 and 5 w.r.t n^* values at the cost of high abort rates. MVC also produced the best overall results concerning deadlocks/validation conflicts;

—The price to pay for the MVC (storage, antiquation) is relatively low compared to the performance gain.

The OCC-algorithms sometimes cause substantially more conflicts than MVC. Hence, they are too optimistic. On the other hand, the RX-scheme is too pessimistic and RAC—limited to two versions—does not allow the use of the full power of versions, obviously. Compared to that, MVC can apply all degrees of freedom introduced by the version idea.

The appropriate selection of the CC-algorithms becomes more important with increasing concurrency (see Table 7). Since higher degrees of parallelism can be expected in future high-performance database systems, MVC seems to be a good candidate. With a growing share of reader transactions its performance gets (relatively) better because all readers are taken away from the resource competition.

MVC does not offer an immediate solution for high-performance transaction systems [28] with high percentages of short writer transactions (DEBIT–CREDIT). Their high performance features mainly rely on special synchronization mechanisms for high traffic data elements [29, 30], asynchronous I/O and special log protocols. However, it would permit the conflict-free scheduling of long readers in parallel.

Even in DB-sharing systems [31], the use of MVC may be advantageous (assuming a clever implementation) because old versions are read-only and cannot be invalidated. But its implications in such systems must be investigated, yet.

The version concept is not suitable for synchronizing hot spots or high traffic data. In these cases, efficient protocols should be designed to circumvent the problem, or better, they should be removed by proper database design.

MVC is no remedy for long writers. Their minimum conflict serialization is hard for every CC-algorithm. Therefore, such transactions should be avoided by appropriate application design.

MVC requires a certain storage overhead. In the sketched implementation with an external version pool, a small percentage of external version references ($\sim$0.5–2.5) is necessary which, nevertheless, may cause performance problems. But in future database systems, storage may be sacrificed for obtaining higher degrees of parallelism. Hence, the MVC-approach becomes more and more realistic with larger memory sizes where the version pool may be entirely integrated in the system buffer making version references and garbage collection to main memory operations. To reduce memory space versioning may be based on record entries instead of pages. Such an approach, of course, requires efficient implementation techniques separating the logging from the version concept. With NOSTEAL buffer management, UNDO-information becomes superfluous. Page commitment at EOT just generates a new version in the buffer (NOFORCE); the necessary REDO-information should be collected on an entry (not page) basis and should be buffered to reduce logging I/O (group commit).

A final problem is the reference of readers to antiquated versions. This is inherent to the MVC and cannot be solved. But it may not be so significant in a practical environment. At any rate, it is no problem for readers running seconds or minutes. With a runtime of several hours the user should be aware that his view to the database is frozen to his BOT-time in order to avoid confusion on the user's side. There is nothing wrong with this kind of interpretation, since consistent results are guaranteed.

REFERENCES

[1] P. A. Bernstein and N. Goodman: Concurrency control in distributed database systems. *Ass. Comput. Mach. Comput. Surv.* **13** (2), 185–222 (1981).

[2] P. Dadam: Synchronization in distributed databases: A survey. *Inform. Spektrum* **4,** 175–184 (1981) (in German).

[3] K. P. Eswaran, J. N. Gray, R. A. Lorie, I. L. Traiger: The notions of consistency and predicate locks in a database system, *Communs Ass. Comput. Mach.* **19** (11), 624–633 (1976).

[4] W. H. Kohler: A survey of techniques for synchronization and recovery in decentralized computer systems. *Ass. Comput. Mach. Comput. Surv.* **13,** (2), 149–184 (1981).

[5] R. Bayer, H. Heller, A. Reiser: Parallelism and recovery in database systems. *Ass. Comput. Mach. Trans. Database Systems* **5** (2), 130–156 (1980).

[6] R. Bayer, K. Elhardt, J. Heigert and A. Reiser, Dynamic timestamp allocation for transactions in database systems. In *Proc. 2nd Int. Symp. on Distributed Data Bases*, pp. 9–20. Berlin (1982).

[7] P. A. Bernstein and N. Goodman, Timestamp based algorithms for concurrency control in distributed database systems. In *Proc. 6th Int. Conf. Very Large Data Bases*, pp. 285–300. Montreal (1980).

[8] W. K. Lin, J. Nolte, Basic timestamp, multiple version timestamp, and two-phase locking. In *Proc. 9th Int. Conf. on VLDB*, pp. 109–119. Florence (1983).

[9] D. P. Reed, Naming and synchronization in a decentralized computer system. Ph.D. dissertation, Dept of Electrical Engineering, M.I.T., Cambrige, MA (1978).

[10]. R. E., Stearns, J. Rosenkrantz, Distributed database concurrency controls using before values. In *Proc. ACM SIGMOD Conf.*, pp. 74–83 (1981).

[11] A. Chan, *et al.* The Implementation of an integrated concurrency control and recovery scheme. In *Proc. ACM SIGMOD Conf.*, pp. 184–191. Fl (1982).

[12] H. T., Kung, J. T. Robinson, On optimistic methods for concurrency control. *Ass. Comput. Mach. Trans. Database Systems* **6** (2), 213–226 (1981).

[13] T. Härder, Observations on optimistic concurrency control schemes. *Inform. Systems* **9** (2), 111–120 (1984).

[14] G. Schlageter, Optimistic methods for concurrency control in distributed database systems. In *Proc. VLDB*, pp. 125–130. Cannes (1981).

[15] R. Unland, U. Praedel, G. Schlageter, Design alternatives for optimistic concurrency control schemes. In *Proc. 2nd Int. Conf. on Databases*, pp. 288–297. Churchill College, Cambridge (1983).

[16] M. Carey, Modeling and evaluation of database concurrency control algorithms, UCB/ERL 83/56. PhD dissertation, University of Berkeley, CA (1983).

[17] W. Kiessling, G. Landherr, A quantitative comparison of lockprotocols for centralized databases. In *Proc. 9th Int. Conf. on VLDB*, pp. 120–131. Florence (1983).

[18] D. R. Ries, M. Stonebraker, Effects of locking granularity in a database management system. *Ass. Comput. Mach. Trans. Database Systems* **2** (3), 233–246 (1977).

[19] D. R. Ries and M. Stonebraker, Locking granularity revisited. *In ACM Trans. on Database Systems* **4** (2), 210–227 (1979).

[20] P. Peinl, A. Reuter, Empirical comparison of database concurrency control schemes. In *Proc. 9th Int. Conf. on VLDB*, pp. 97–108. Florence (1983).

[21] T. Härder, P. Peinl, A. Reuter, Performance analysis of synchronization and recovery schemes. *IEEE Database Engng* **8** (2), 50–57 (1985).

[22] E. Petry, Simulation and analysis of an implicit version concept for database systems. *Diploma* thesis, University of Kaiserslautern (1984) (in German).

[23] D. A. Menasce, T. Nakanishi, Optimistic vs. pessimistic concurrency control mechanisms in database management systems. *Inform. Systems* **7,** (1), 13–27 (1982).

[24] J. N. Gray, R. A. Lorie, G. R. Putzolu and I. L. Traiger, Granularity of locks and degrees of consistency in a shared database. In *Modeling in Database Management Systems* (Edited by G. M. Nijssen) pp. 365–394. Elsevier North-Holland, New York (1976).

[25] I. Arifin, Empirical investigations of locking proctocols in database systems. Diploma thesis, University of Kaiserslautern (1983) (in German).

[26] E. Gerstner, Empricial investigations of optimistic concurrency control in database systems. Diploma thesis, University of Kaiserslautern (1983) (in German).

[27] T. Härder, A. Reuter, Principles of transaction oriented database recovery. *Ass. Comput. Mach. Comput. Surv.* **15** (4), 287–317 (1983).

[28] J. Gray *et al.*, One thousand transactions per second. In *Proc. IEEE Spring Computer Conf.*, pp. 96–101, San Francisco, CA (1985).

[29] D. Gawlick, Processing "hot spots" in high performance systems. In *IEEE Spring Computer Conf.*, pp. 249–257, San Francisco, CA (1985).

[30] Reuter, A.: Concurrency on High-Traffic Data elements. In *Proc. Conf. on Principles of Database Systems*, pp. 83–93. Los Angeles, CA, (1982).

[31] K. Shoens *et al.*, The AMOEBA project. In *Proc. IEEE Spring Comput. Conf.*, pp. 102–105. San Francisco, CA (1985).

[32] *UDS, Universal Data Base Management Systems* UDS-V4 Reference manual package, Siemens AG, Munich (1983).