

# A Comparison of Multi-Level Concurrency Control Protocols

Markus Kirchberg, Klaus-Dieter Schewe

*Massey University, Department of Information Systems,  
Private Bag 11 222, Palmerston North, New Zealand  
[m.kirchberg|k.d.schewe]@massey.ac.nz*

## Abstract

*Locking protocols for multi-level transactions have been studied since the very beginning. More recently, a hybrid concurrency control protocol for multi-level transactions called FoPL has been developed. It employs access lists on the database objects and forward oriented commit validation. The basic test on all levels is based on the re-ordering of the access lists.*

*So far a detailed analysis of FoPL's benefits is missing. This paper describes a testbed for multi-level transactions which allows to measure transaction throughput and rollback frequency. The testbed allows to use a mix of strict two-phase locking and FoPL on up to 4 levels. The tests work on randomly generated multi-level transactions on the basis of pages, records and virtual database objects on higher levels.*

**Keywords.** [H2.4] Transaction Processing, Concurrency [H2.7] Logging and Recovery

## 1 Introduction

Multi-level transaction schedulers adapt conflict-serializability on different levels. High-level operations are implemented by operations from the next-lower level. This gives the opportunity to detect pseudoconflicts, i.e., conflicts that do not stem from a higher-level conflict. These pseudoconflicts can be ignored and hence their detection increases the rate of concurrency. Section 2 describes the main features of multi-level systems following [1, 9, 10].

Generally, the preventive approach by locking protocols, especially strict two-phase locking (str-2PL) is used in most practical systems in order to generate a (conflict-)serializable schedule. However, the known disadvantages of the approach, i.e., the impossibility to accept all (conflict-)serializable schedules, the problem of deadlocks, etc. has stimulated research on alternative concurrency control protocols.

The FoPL protocol (Forward oriented Concurrency Control with Preordered Locking) [9] is a provably correct hybrid concurrency control protocol for multi-level transactions. It consists of three phases. First, in the *propagation-phase* all operations are executed in an optimistic way. During their execution some control structures consisting of flaglists for the objects and accesslists for the operations are built up. The second phase called *validation-phase* evaluates these structures and decides if a transaction can be committed or has to be aborted. Finally, in the third phase called *commit-phase* the commit or abort—which is now a complex operation—is executed. In Section 3 we introduce these two protocols, together with their problems, restrictions and further optimizations.

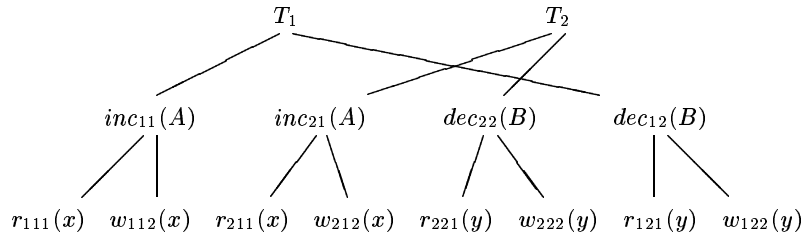
In order to evaluate concurrency control protocols and to enable an experimental comparison we implemented a testbed consisting of a database generator, a transaction generator and the transaction management system itself. The system is based on a general-purpose pages, system buffer and record administration server. This testbed has been used to evaluate and compare strict two-phase locking and the FoPL protocol.

Section 4 presents first experimental results concerning the behaviour of FoPL operating on one or more levels on a multi-level system in comparison to str-2PL. We also discuss necessary optimizations to improve the currently realized system and the tests.

The comparison extending an older evaluation in [11] has been described in detail in [6]. Technical details on the implementation of the used testbed can be found in [5].

## 2 The Multi-Level Transaction Model

A multi-level transaction is a special kind of an open nested transaction, where the leaves in the transaction tree have the same depth. Each node in the tree corresponds to some operation implemented by its successors. The root is a transaction. The lowest level  $L_0$  corre-



**Fig. 2.1.** Serializable multi-level schedule

sponds to operations that access directly the physical database.

An  $n$ -level-system  $\mathcal{L}$  consists of  $n$  layers  $L_i = (\mathcal{D}_i, \mathcal{F}_i)$  ( $i = 0, \dots, n-1$ ), where  $\mathcal{D}_i$  is a set of *objects* and  $\mathcal{F}_i$  a set of *operators*. An  $L_i$ -operation is an element of  $\mathcal{O}_i = \mathcal{F}_i \times \mathcal{D}_i$ .

We write  $\mathcal{L} = (L_{n-1}, \dots, L_0)$ . The levels are numbered in a bottom-up manner. An object  $x \in \mathcal{D}_i$  can only be accessed as a whole and in one atomic step.

## 2.1 Multi-Level Transactions

An  $n$ -level transaction is defined next exploiting the notion of an *index tree*, which is a finite set of finite sequences over  $\mathbb{N}$ . We let  $\mathbb{N}^*$  denote the set of all such sequences.  $|\alpha|$  denotes the length of  $\alpha \in \mathbb{N}^*$ . Furthermore, we identify numbers with sequences of length 1 and denote the empty sequence by  $\epsilon$ .

An index tree of depth  $n$  is a finite subset  $I \subseteq \mathbb{N}^*$  with  $\epsilon \in I$ ,  $\alpha(k+1) \in I \Rightarrow \alpha k \in I$  and  $\alpha \in I \wedge |\alpha| < n \Leftrightarrow \alpha 1 \in I$  for all  $\alpha \in \mathbb{N}^*$  and  $k \in \mathbb{N}$ .

An  $n$ -level-transaction  $T_j$  over an index tree  $I$  of depth  $n$  assigns to each  $\alpha \in I$  an  $L_{n-|\alpha|}$ -operation, denoted as  $op_{j\alpha}$  and partial orders  $<_i^{(j)}$  on each  $\mathcal{O}_i^{(j)} = \{op_{j\alpha} \mid |\alpha| + i = n\}$ , such that  $op_{j\alpha k} <_i^{(j)} op_{j\beta \ell} \Rightarrow k < \ell$  holds. We call  $<_i^{(j)}$  the  $L_i^{(j)}$ -precedence relation.

By abuse of notation we write  $op_{j\alpha}(x)$  for the operation  $op_{j\alpha} = (op, x)$ .

**EXAMPLE 2.1.** The trees rooted at  $T_1$  and  $T_2$  in Figure 2.1 define two 2-level-transactions over the same index tree. Here  $r$  and  $w$  correspond to read- and write-operations,  $inc$  and  $dec$  to incrementation and decrementation. Thus, we may assume  $<_0^{(j)}$  as total and  $<_1^{(j)}$  as empty ( $j = 1, 2$ ).  $\square$

The edges in a transaction tree represent the implementation of an  $L_i$ -operation by a set of  $L_{i-1}$ -operations. If  $op_{j\mu k}$  is an  $L_i$ -operation of a transaction  $T_j$ , then  $trans(op_{j\mu k}) = op_{j\mu}$  ( $0 \leq i < n$ ) is the  $L_{i+1}$ -operation that invokes  $op_{j\mu k}$ . Conversely,  $act(op_{j\nu}) =$

$\{op_{j\nu\ell} \mid \nu\ell \in I\}$  defines the set of  $L_{i-1}$ -operations implementing the  $L_i$ -operation  $op_{j\nu}$ .

Precedence relations are meant to express a necessary ordering of implementing operations to require  $op_{j\alpha} <_i^{(j)} op_{j\beta} \Leftrightarrow op_{j\alpha k} <_i^{(j)} op_{j\beta \ell}$ , whenever the involved operations exist. In this case, the transaction  $T_j$  is well-defined. In the sequel we shall tacitly assume that all transactions are well-defined.

## 2.2 Multi-Level Schedules

The execution of concurrent transactions is described by an  $n$ -level-schedule, as it is illustrated in Figure 2.1 and 2.2.

An  $n$ -level-schedule consists of a set  $\mathcal{O}_n = \{T_1, \dots, T_k\}$  of  $n$ -level-transactions and a partial order  $<_0$  on  $\mathcal{O}_0$  containing all  $L_0^{(j)}$ -precedence relations, where  $\mathcal{O}_i = \bigcup_{j=1}^k \mathcal{O}_i^{(j)}$  is the set of all  $L_i$ -operations in these transactions ( $0 \leq i < n$ ).

We write  $S = (\mathcal{O}_n, \mathcal{O}_{n-1}, \dots, \mathcal{O}_0, <_0)$  for such a schedule. Then  $<_0$  induces a partial order  $<_i$  on each level  $i > 0$  by  $op_\mu <_i op_\nu \Leftrightarrow$

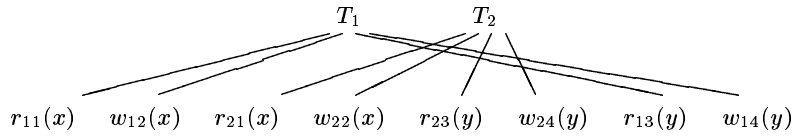
$$\forall op_{\mu k} \in act(op_\mu). \forall op_{\nu \ell} \in act(op_\nu). op_{\mu k} <_{i-1} op_{\nu \ell} \quad .$$

Using this, we may define the *level-by-level-schedule*  $S_{i,j}$  ( $j < i \leq n$ ) as the one-level-schedule  $(\mathcal{O}_i, \mathcal{O}_j, <_j)$ .

The basic idea of multi-level concurrency control is to use the semantics of operations in level-specific *conflict relations*  $CON_i \subseteq \mathcal{O}_i \times \mathcal{O}_i$ . Non-conflicting operations should commute.

Same as with precedence relations the intention behind the conflict relations forces us to require the following conformity condition: If  $(op_\mu, op_\nu) \in CON_i$  holds for some  $op_\mu, op_\nu \in \mathcal{O}_i$ , then there should exist  $op_{\mu k} \in act(op_\mu)$  and  $op_{\nu \ell} \in act(op_\nu)$  with  $(op_{\mu k}, op_{\nu \ell}) \in CON_{i-1}$ . In the sequel we shall tacitly assume that this condition is satisfied by all schedules.

**EXAMPLE 2.2.** Incrementations and decrementations are commutative to one another. Therefore, for the



**Fig. 2.2.** Non conflict serializable single-level schedule

transactions in Figure 2.1 we would like to use  $((op_1, x), (op_2, y)) \in CON_1 \Leftrightarrow op_1 = op_2 = upd \wedge x = y$  – assuming  $\mathcal{F}_1 = \{inc, dec, upd\}$ . Analogously,  $((op_1, x), (op_2, y)) \in CON_0 \Leftrightarrow (op_1 = w \vee op_2 = w) \wedge x = y$  – assuming  $\mathcal{F}_0 = \{r, w\}$ . Note that the  $L_0$ -conflict relation is the usual one used for flat transactions.

Hence, the schedule in Figure 2.1 should be acceptable, but the level-by-level schedule  $S_{2,0}$  in Figure 2.2 is not conflict serializable in the usual sense for flat transactions. Thus, multi-level transactions may be expected to increase concurrency.  $\square$

### 2.3 Conflict Serializability

We have to extend the notion of conflict-serializability to multi-level transactions to make these arguments rigorous. First, an  $n$ -level-schedule with a total order  $<_n$  is called *serial*. Then *serializability* means equivalence to a serial schedule in the following formal sense.

Let  $(\mathcal{O}_n, \mathcal{O}_{n-1}, \dots, \mathcal{O}_0, <_0)$  be an  $n$ -level-schedule with induced partial orders  $<_i$  on level  $i$ . Let  $CON_i$  ( $i = 0, \dots, n-1$ ) be conflict relations. Define  $op_\mu \rightarrow_i op_\nu \Leftrightarrow (op_\mu, op_\nu) \in CON_i \wedge op_\mu <_i op_\nu$  for  $op_\mu, op_\nu \in \mathcal{O}_i$ .

Then two  $n$ -level-schedules are called (*conflict-*) *equivalent* iff their associated relations  $\rightarrow_i$  coincide for all  $i = 0, \dots, n-1$ . An  $n$ -level schedule which is conflict-equivalent to a serial one, is called (*n-level-*)*serializable*.

According to [1] an  $n$ -level-schedule  $S$  is  $n$ -level-serializable, if all level-by-level schedules  $S_{i,i-1}$  of  $S$  ( $0 < i \leq n$ ) are conflict serializable.

**EXAMPLE 2.3.** Using the conflict relations from Example 2.2 it is easily verified that the schedule in Figure 2.1 is conflict serializable, whereas the one in Figure 2.2 is not. This was already stated above.  $\square$

## 3 Multi-Level Concurrency Control Protocols

After a short introduction to the multi-level transaction model including conflict serializability let us now discuss two different protocols: str-2PL and FoPL.

### 3.1 Strict Two-Phase Locking

As already mentioned only  $L_i$ -operations accessing the same object might give rise to a conflict. Therefore, it is sufficient to define a level-specific lock  $lock_{op}$  for each operator  $op$ . Then each  $L_i$ -operation  $op_\mu(x)$  may only be executed after getting a lock  $lock_{op}$  on object  $x$ .

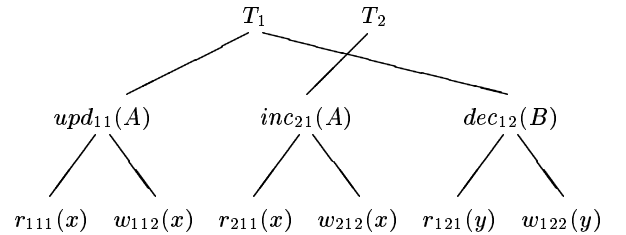
If  $op_\mu = (op_1, x)$  and  $op_\nu = (op_2, x)$  are  $L_i$ -operations on the same object  $x \in \mathcal{D}_i$ , their corresponding locks  $lock_{op_1}$  and  $lock_{op_2}$  are called *incompatible* iff  $op_\mu \rightarrow_i op_\nu$  or  $op_\nu \rightarrow_i op_\mu$  holds. Otherwise they are called *compatible*.

Strict two-phase locking is defined by the following two rules:

1. An  $L_i$ -operation  $op_{\mu k}$  (for  $0 \leq i < n$ ), working on object  $x \in \mathcal{D}_i$  has to acquire a lock  $lock_{op}$  on this object  $x$  before actually being executed.
2. This lock is released at the end—i.e., commit or abort—of the  $L_{i+1}$ -operation  $op_\mu$  to which  $op_{\mu k}$  belongs, together with all other  $L_i$ -locks acquired during the execution of  $op_\mu$ .

It has been shown that multi-level schedules accepted by str-2PL on each level are always serializable [9].

However, it is also well-known that there exist serializable schedules that cannot be accepted by the str-2PL. Such a schedule is shown in the following picture:



**Fig. 3.1.** Serializable schedule, not acceptable by 2PL

Before the execution of  $inc_{21}(A)$  transaction  $T_1$  obtained  $lock_{upd}$  on object  $A$ . Using str-2PL, the release of this lock is not allowed before the last  $L_1$ -operation

$dec_{12}(B)$  is executed. Thus, transaction  $T_2$  cannot acquire  $lock_{inc}$  on object  $A$  because of the incompatibility of this lock with existing locks held by  $T_1$ . Nevertheless, the shown schedule is serializable.

### 3.2 The Basic FoPL Protocol

The basic structure of FoPL follows the idea of optimistic protocols but also includes preventive locks. FoPL consists of three phases: the propagation-, validation- and commit-phase. In the propagation-phase all operations are executed in an optimistic way. During their execution some control structures consisting of flaglists for the objects and access-lists for the operations are built up. The validation-phase evaluates these control structures and decides if a transaction can be committed or has to be aborted. Finally, in the commit-phase the commit or abort is executed.

The *access-set*  $AS_i^{(\mu)}$  of an  $L_i$ -operation  $op_\mu$  (for  $0 < i \leq n$ ) is defined as  $AS_i^{(\mu)} = act(op_\mu)$ . A *flag*  $z$  of an  $L_i$ -operation  $op_{\mu_j}$  (for  $0 < i \leq n$ ) consists of the operator  $op$  and the index of the parent operation  $op_\mu$ , i.e.,  $z = (op, \mu)$ . An *access-list*  $ZL_x$  of an object  $x \in \mathcal{D}_i$  is a list of flags of  $L_i$ -operations, accessing the object  $x$ .

In the **propagation-phase** each  $L_i$ -operation  $op_\mu$  executes all its  $act(op_\mu)$ -operations. Simultaneously, each  $op_{\mu_j}(x) \in act(op_\mu)$  creates a flag  $z_j$ , which will be appended to the end of the access-list of object  $x$ .

If all  $L_{i-1}$ -operations in  $act(op_\mu)$  have been executed,  $op_\mu$  activates its **validation-phase**. For this, FoPL requests locks for all objects  $x \in \mathcal{D}_{i-1}$  that were accessed by one of the  $act(op_\mu)$ -operations. To avoid deadlocks the locks are requested in a total order. These locks will be kept until the end of the commit-phase. After receiving all locks FoPL has to test if all flags that stem from  $act(op_\mu)$ -operations are still set. If at least one flag is missing, the operation  $op_\mu$  must be aborted. Otherwise FoPL has to test if the operation  $op_\mu$  was *successful* on all objects  $X$  accessed by  $act(op_\mu)$ , i.e., there are no flags  $(op_1, \nu)$  and  $(op_2, \mu)$  in  $ZL_x$  with  $\nu \neq \mu$  such that  $(op_1, \nu)$  precedes  $(op_2, \mu)$  and  $((op_1, x), (op_2, x)) \in CON_{i-1}$  holds.

If the  $L_i$ -operation  $op_\mu$  is successful, it can be committed, otherwise it must be aborted. Both actions are accomplished during the **commit-phase**. If an  $L_i$ -operation  $op_\mu$  may be committed, the access-lists of all objects  $x \in \mathcal{D}_{i-1}$  have to be updated and all locks requested during the validation-phase have to be released.

Otherwise, if the  $L_i$ -operation  $op_\mu$  has to be aborted, all operations in  $act(op_\mu)$  have to be aborted, too. In this situation we distinguish between two cases.

If the flag of an operation is still set, a rollback of the operation is necessary. No additional locks have to

be requested, because all required locks are still kept. After rolling back all flags and all dependent flags have to be deleted. A flag  $z$  from  $op_\mu$  depends on another flag  $z'$  from  $op_\nu$ , iff  $z'$  precedes  $z$  in  $ZL_x$  and  $(op_\mu, op_\nu) \in CON_i$  holds or  $z$  depends on  $z''$  and  $z''$  depends on  $z'$  for some flag  $z''$ .

If the flag of an operation is deleted, no additional recovery operations are necessary, because an earlier rollback of another operation has already rolled back all changes from this operation.

Finally, all locks requested during the validation-phase have to be released. Same as for str-2PL all schedules accepted by FoPL are serializable.

### 3.3 FoPL Enhancements

Several optimizations of the basic FoPL protocol have been proposed [9]. The most important ones are *lazy aborts* and *early and partial rollbacks*.

In order to reduce the number of aborts we force an operation to wait and to restart after some time period. If we combine this with FoPL, we call the resulting protocol FoPL<sup>+</sup>. Using this we may hope that a preceding conflicting flag has been deleted in the meantime. Thus, aborts will only occur if they are unavoidable. It is easy to see that FoPL<sup>+</sup> will accept the schedule in our previous example.

During the propagation-phase it is possible to inform an  $L_i$ -operation  $op_\mu$  immediately, whenever one of its flags has been deleted by another  $L_i$ -operation. This prevents further  $act(op_\mu)$ -operations from being executed, as it is already known that they will be executed later.

Furthermore it is not necessary to rollback operations completely. Since only dependent flags are deleted, it is sufficient to do a partial rollback back to the earliest time point, where none of these flags was set.

## 4 Experiments and their Evaluation

In order to compare the used multi-level transaction schedulers and the two concurrency control protocols, respectively, we executed several test series. The general process for these can be divided into three phases:

1. Preparation for the series of tests:
  - (a) Creation of a test database or reuse of an existing database.
  - (b) Configuration of operator-tables and conflict relations.
  - (c) Configuration of constant test-parameters for the series of tests.

2. Execution of a test:
  - (a) Configuration of the constant test-parameters for the test-run.
  - (b) Execution of a ‘sufficient’ number of tests.
3. Evaluation of the series of tests:
  - (a) Examination of the test-results and repetition of the second phase if the test-results are not yet sufficient.
  - (b) Evaluation and assessment of the test-results.

#### 4.1 The Multi-Level Concurrency Control Testbed

Tests are processed on a testbed described in detail in [5]. The following picture illustrates the various components of the testbed:

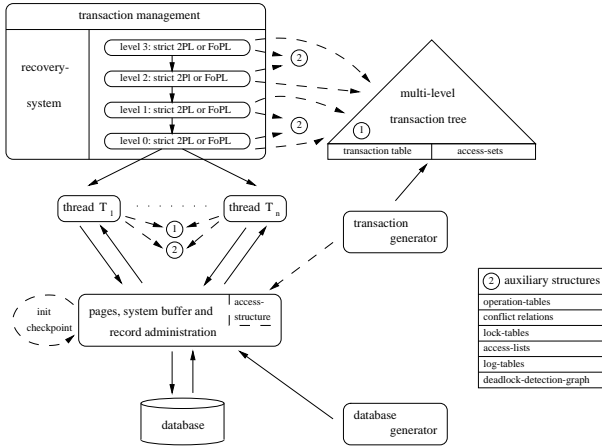


Fig. 4.1. System overview

The testbed mainly consists of four modules and several data structures. These modules are

- a database generator which generates a random database as the basis of a series of tests;
- an underlying pages, system buffer and record server which makes all requested data available in main memory in the form of pages, implements a pages and record interface, accepts all incoming queries and executes them;
- a transaction generator which generates random multi-level transactions and supervises the tests;
- the transaction management system itself which has to schedule and execute all active transactions.

The testbed supports 2-, 3- and 4-level systems. In each case records are used as the objects on level

$L_0$ . This is demanded by the use of the pages, system buffer and record server, translating the record-operations to page-operations and synchronizing their changes via short-term-locks. So far the implemented testbed uses a rather simple recovery method based on the undo-functionality of the pages, system buffer and record server. Thus, only full-transaction rollbacks are possible.

The objects on level  $L_1$  are either sets of records or sets of record-fields. On levels  $L_2$  and  $L_3$  sets of objects from the next-lower level are used. Furthermore, we restrict the possible combinations of the level-by-level transaction schedulers in such a way that FoPL never occurs on a lower level than str-2PL. In the sequel we use the notation  $l_3-l_2-l_1-l_0$  with  $l_i \in \{0, 1\}$  to denote which protocol is used on the various levels:  $l_i = 0$  corresponds to str-2PL,  $l_i = 1$  to FoPL.

The whole system is implemented in C on a Linux operating system and contains about 33,000 lines of code. The work in [5] contains a detailed description of the testbed implementation exploiting standard database implementation techniques [2, 3, 4].

#### 4.2 The Tests

We ran three test series. The first test series compares the 4-level transaction schedulers on sequentially executed transactions. The transaction generator only creates a new transaction if no other transaction is active. This series of tests can only indicate implementation overheads in the protocols, but does not really influence the final assessment of the protocols. Indeed, we got nearly the same results for all the protocols.

The second test series uses a high conflict rate (about 75%) and runs several tests with different parameters to

- confirm the disadvantages of the FoPL protocol under these circumstances,
- test the behaviour of those transaction schedulers that use a combination of the introduced concurrency control protocols,
- test under which conditions a transaction scheduler, which uses only FoPL achieves nearly the same results as a str-2PL-based multi-level transaction scheduler.

The third test series tries to find out how low the conflict rate has to be such that a FoPL-based transaction scheduler achieves feasible results in comparison to all other possible transaction schedulers. Based on this result we execute further tests with few concurrent, large multi-level transactions or a lot of small multi-level transactions, respectively. The common goal of this third test series is to get a picture of conditions under which FoPL would be advantageous.

The evaluation of all these results is based on the throughput of the executed multi-level transactions considering the number of necessary rollbacks and aborts, respectively and the time needed to execute all transactions. We calculate the throughput  $T$  by

$$T = \frac{not - noa}{t},$$

where  $not$  is the number of transactions,  $noa$  is the number of aborts and  $t$  is the execution time for the test-run.

### 4.3 Evaluation of the Test-Results

Under the conditions in the second test series str-2PL produces many waiting periods and some deadlocks and the FoPL protocol aborts a lot of transactions because of negative validation results. First we ran a series of tests on a 4-level system with large transactions. About 108  $L_0$ -operations were executed by each transaction and we varied the number of parallel active transactions. The results are shown below:

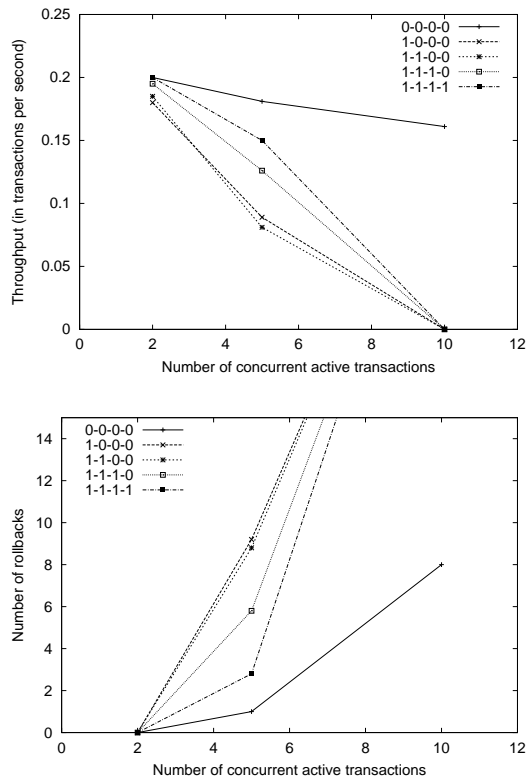


Fig. 4.2. Results of the 1st test-run of test 2

Scheduling five or more transactions at the same time already resulted in tremendous differences between the

measured throughputs of the transaction schedulers. If we even increase the number of parallel executed transactions the differences between str-2PL and the (partly) optimistic transaction schedulers rose quickly up to a point where the sequential execution of the transactions would even be faster. The reason for this is the high conflict rate causing repeated rollbacks.

In the next step within the second series of tests we ran a 2-level system with few  $L_0$ -operations and a lot of parallel transactions—within a period of 25 seconds 100 transactions were activated. We varied the number of top-level transactions. The results are shown below:

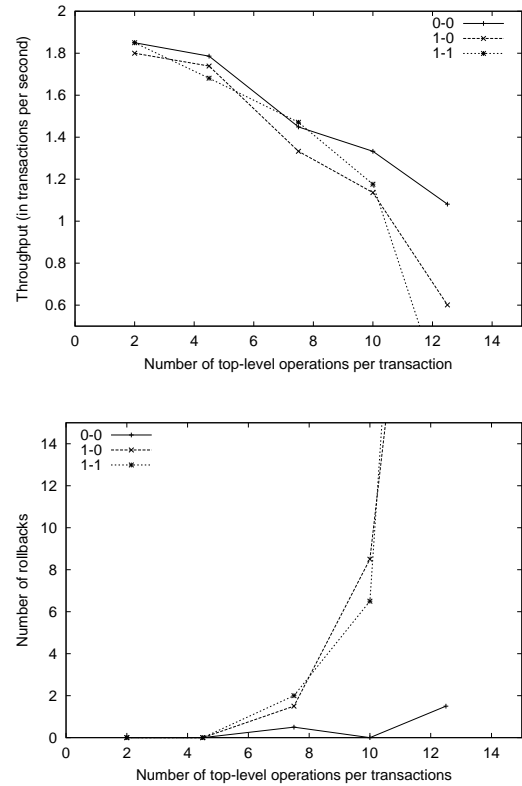


Fig. 4.3. Results of the 2nd test-run of test 2

For five or less  $L_1$ -operations nearly equal results have been measured, but this could be explained by the fact that the execution of such a small transaction is faster than the creation of a new transaction. Hence these transactions were almost executed sequentially.

For transactions with less than ten  $L_1$ -operations we got nearly the same results for the possible 2-level transaction schedulers. The reason for this is that the probability of two operations of different transactions accessing the same object being very low, because we have

only few small transactions. If we increase the number of top-level operations again, the differences between the (partly) FoPL-based transaction schedulers and str-2PL rises fast. The reason for this is the increase of the average number of rollbacks per transaction, caused by the increase of the conflict probability.

For the third test series we first compared 4-level transaction schedulers with reduced conflict rate. The results of this series of tests are shown below:

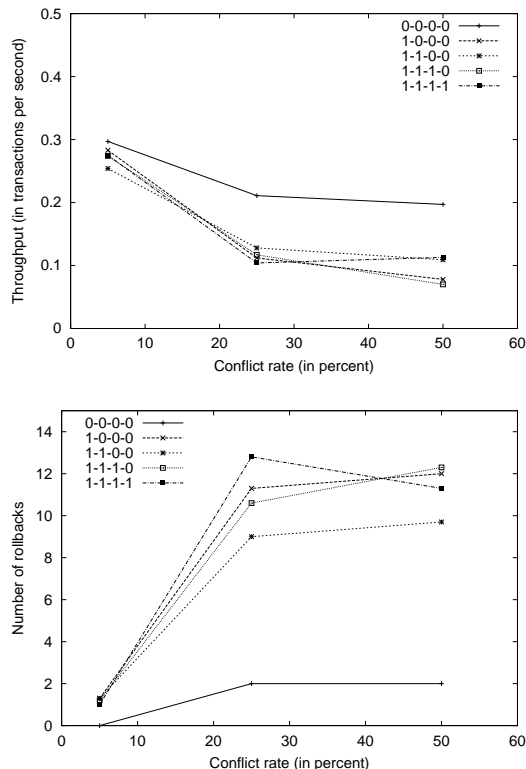


Fig. 4.4. Results of the 1st test-run of test 3

As expected a (partly) FoPL based transaction scheduler is worth to be used if we have a lower conflict rate. The results show that below 15% of conflicts the differences between the throughputs decrease. With about 5% we get best results for the (partly) FoPL-based transaction schedulers. The main-reason for the (partly) FoPL-based transaction schedulers never being better than the str-2PL are as follows:

1. The decrease of the conflict rate also reduces the probability that two operations executed concurrently are in a conflict such that their access to the same object has to be synchronized by exclusive locks. This causes an increase of the throughput of a str-2PL-based scheduler.

2. Currently, we only use the basic FoPL protocol for the tests, but the theoretical advantages in comparison to str-2PL arise only with the enhancements we have discussed.

Finally, we continued the tests with a conflict rate of about 5%. It turned out that it is unfavourable to use sets of records as  $L_1$ -objects. This again is caused by a higher conflict probability.

As an extreme case we got the situation where the FoPL-based transaction scheduler had a 24% higher throughput than the str-2PL transaction scheduler. Because of this we ran a further series of tests to confirm or relativate this result. The result of these additional tests was that only under some favourable conditions the FoPL-based transaction scheduler was better than str-2PL. If we slightly variate these conditions, then FoPL loses again.

#### 4.4 Assessment

The executed tests show that only str-2PL-based transaction schedulers achieve acceptable results on all used  $n$ -level systems under all test-conditions. These protocols prevent a high number of rollbacks and hence guarantee at least a reasonable throughput. The conditions detected for multi-level transaction schedulers based only on the FoPL protocol to achieve useful throughput are as follows:

1. A low conflict rate of less than 10% on each level of the multi-level system.
2. A restricted number of multi-level transactions executed concurrently. This number depends on the average number of executed  $L_0$ -operations per transaction.
3. A restricted number of  $L_0$ -operations per multi-level transaction depending on the average number of transactions executed concurrently and the used object granularity.

This does not yet imply that str-2PL is the only reasonable choice and there is no more need to look for alternatives. In particular, our assessment depends on the implemented system with all its current restrictions. The main reasons for the restrictive conditions on FoPL are the poor multi-level recovery system and the enhancements for FoPL not yet being implemented. For instance, it has been shown in [9] that FoPL<sup>+</sup>, i.e., the enhanced protocol with lazy aborts, is able to accept all serializable schedules. Furthermore, the sophisticated ARIES/ML recovery system [9] based on work in [7, 8] is

much more suitable for FoPL. Thus, we have to improve the testbed.

Generally speaking it is not advantageous to use the basic FoPL protocol in one or more levels as a concurrency control algorithm unless it is extended with the known enhancements. Furthermore, it may be necessary to rethink the restrictions from the testbed implementation itself.

## 5 Conclusion

This article describes first experimental results on a hybrid concurrency control protocol called FoPL (Forward oriented Concurrency Control with Preorder Locking) used on one or more levels of a multi-level system. To get useful results we compared this protocol with the well-known and most common strict two-phase locking protocol.

The hybrid protocol FoPL was developed to provide an alternative to the classical concurrency control protocols by combining their advantages and avoiding their problems. Namely, an optimized version of FoPL can accept all possible (conflict-)serializable schedules and realizes a high rate of concurrency by the use of an optimistic approach in its first phase. FoPL has been designed to be used in cases of low conflict rates and short transactions.

The basic FoPL protocol has no real advantages in comparison to the str-2PL, but several enhancements are known to improve its efficiency such as lazy aborts, early and partial rollbacks, etc.

On the basis of a testbed implementation we ran several test series to confirm the expected disadvantages of FoPL in the case of high conflict rates and to examine conditions under which (partly) FoPL-based multi-level transaction schedulers produce nearly the same or better results than the preventive transaction scheduler. A further test series tried to find out conflict rates, where a (partly) FoPL-based transaction scheduler has general advantages or at least no essential disadvantages compared with the str-2PL-based transaction scheduler.

The general result of all these tests is that a (partly) FoPL-based scheduler is not a real alternative to str-2PL without implementing further optimizations such as

- a sophisticated multi-level recovery-system as given by ARIES/ML,

- the known optimizations of the basic FoPL protocol, and
- improvements to the generation of the multi-level transactions and more realistic conditions to execute tests.

At this stage we can state that it is worth to improve the testbed and to run further series of tests, but only after implementing the discussed optimizations. We expect that we shall be able to detect cases where str-2PL will no longer appear as the undoubted best protocol.

## References

1. C. Beeri, P. A. Bernstein, N. Goodman. A Model for Concurrency in Nested Transaction Systems. *Journal of the ACM*, Vol. 36, No. 2, April 1989, Pages 230-269.
2. D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley, 1997.
3. W. Effelsberg, T. Härder. Principles of Database Buffer Management. *ACM Transactions on Database Systems*, Vol. 9, No. 4, December 1984, Pages 560-595.
4. T. Härder, E. Rahm. *Datenbanksysteme - Konzepte und Techniken der Implementierung*. Springer, 1999, in German.
5. M. Kirchberg. Ein experimenteller Vergleich von Transaktionsschedulern für Mehrschichten-Transaktionen. Master Thesis, Technical University Clausthal, 2000, in German.
6. M. Kirchberg, K.-D. Schewe. An Experimental Comparison of Concurrency Control Protocols for Multi-Level Transactions. Technical Report 4/2000. Massey University, Dept. of IS. <http://fims-www.massey.ac.nz/~kdschewe/pub/articles/MUrep00b.pdf>
7. D. B. Lomet. MLR: A Recovery Method for Multi-level Systems. In M. Stonebraker (Ed.). *Proc. SIGMOD 1992, San Diego 1992*, Pages 185-194.
8. K. Rothermel, C. Mohan. ARIES/NT: A Recovery Method Based on Write-Ahead Logging for Nested Transactions. *Proc. 15th VLDB, 1989*, Pages 337-346.
9. K. D. Schewe, T. Ripke, S. Drechsler. Hybrid Concurrency Control and Recovery for Multi-Level Transactions. In *Acta Cybernetica Volume 14, 2000*, Pages 419-453.
10. G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, Vol. 16, No. 1, March 1991, Pages 132-180.
11. G. Weikum, C. Hasse. Multi-level Transaction Management for Complex Objects: Implementation, Performance, Parallelism. In: *The VLDB Journal*, (2)4, 1993.