THE POWER OF THE PRIVATE WORKSPACE MODEL

by

I. Gold and H. Boral*

Technical Report No. 338
October 1984

* Microelectronics' and Computer Technology Corporation,
  Austin, TX, U.S.A.
ABSTRACT

In the private workspace model of concurrency control the transaction management component of a database management system maintains a private workspace for each transaction. Data items accessed by a transaction, regardless of the access mode, are cached in the workspace. At transaction commit time, its updates are made permanent in the database. In this paper we want to focus the attention to the private workspace model as a framework for the design of concurrency control algorithms which can do without ww synchronization. We present a 2PL derivative, named Workspace 2PL (W2PL), and show that in a system employing W2PL no ww synchronization is needed, and more important, no transaction will restart on a READ request. Furthermore, under the No Blind Writes assumption we can guarantee that no READ request will cause a transaction to starve, thus no Reader will be restarted or starved.
1. Introduction

Several centralized concurrency control algorithms have been proposed during the past several years. The majority of the algorithms make use, to one degree or another, of three synchronization techniques: locking, timestamping and certification. Locking algorithms are based on the Two Phased Locking method (2PL) [ESWA76] in which blocking is used to synchronize conflicting transactions. Timestamps algorithms [BERN80] select a priori order on transaction execution, usually, by assigning timestamp to each transaction at its initiation, and resolve conflicts between transactions by forcing them obey that order. Certification algorithms allow conflicting transactions to run concurrently but use transaction restart in cases where data inconsistency could result [BADAL79], [KUNG81] (known as Certification methods because they certify a transaction for additional processing or commit, or cause it to restart).

Recently researchers have begun to compare several different algorithms in an attempt to reach some conclusion concerning their operational merit [AGRA83], [CARE82], [GALL82], [ROBI82], [ROBI82a]. A conclusion that is reached in some of these references is that assuming that the overhead associated with locking is small, transactions restarts affect throughput more negatively than locking. In other words, 2PL is generally a better algorithm than either a timestamp or certification.¹

A database management system model which utilizes private workspaces allocated to each active transaction to cache its previously read data items as well as those written by it during its execution was introduced by Bernstein and Goodman [BERN81] and Kung and Robinson [KUNG81]. In this private workspace model transaction execution can be seen as composed of two phases. The execution phase, where a transaction reads data items from the database, performs

¹ If information about the transaction's read and write sets, or access patterns, is available this may not be true.
various computations and writes new values for data items in its private workspace; and the *atomic commit phase*, which takes place after the transaction finishes all computations, and during which the transaction atomically installs its local copies of data items into the database.

Bernstein and Goodman [BERN81] showed that we can design concurrency control algorithms consisting of:

1. An *rw* synchronization technique which amounts to synchronize READ-WRITE conflicts.
2. A *ww* synchronization technique which amounts to synchronize WRITE-WRITE conflicts.
3. An interface between the above synchronization techniques which attains a total order of the transactions consistent with all *rw* and *ww* conflicts.

In this paper we examine two properties of the private workspace model. First, we show that due to the *atomic commit phase*, the need for *ww* synchronization is removed, leading to simpler algorithms. One such an algorithm is Kung and Robinsons [KUNG81] serial validation an its timestamp version in [CARE83], while we present another algorithm, a 2PL variant named Worksapce 2PL (W2PL).

Second, we observe that since writing of new values into the private workspaces of the transactions do not affect the database state, the concurrency control mechanism has a complete freedom in choosing how to synchronize conflicting operations. W2PL uses this freedom for an optimization in which no transaction will restart on execution of a READ request. Under the No Blind Writes assumption W2PL also guarantees that no READ request will cause a transaction to starve, thus no Reader will be restarted or starved. In the light of the performance evaluation work reported above we believe this makes the private workspace model, and particular the W2PL algorithm, possible attractive alternatives to current concurrency control schemas.
In Section 2 we review the private workspace model used and discuss its serializability implications. Section 3 presents the W2PL algorithm -- both data structures and the operations the W2PL performs on them on behalf of transactions. In Section 4 we prove the correctness of the algorithm. Section 5 shows how we can guarantee that no transaction will deadlock on a READ request and Section 6 proves that under the No Blind Writes assumption no READ request will cause a transaction to starve. We conclude with a summary and reference to related work in Section 7.

2. The Private Workspace Model

2.1. Model Description

The private workspace model utilizes private workspaces allocated to active transactions to cache their previously read data items as well as those written by the transaction during its execution. Bernstein and Goodman [BERN81] and Kung and Robinson [KUNG81] used this model previously. The following description closely models that of [BERN81], with the addition of the PRE_WRITE operation as a synchronization primitive.

A centralized database management system can be seen as composed of two components: a Transaction Manager (TM) and a Data Manager (DM). The TM controls interaction between users and the database management system and is responsible for such functions as concurrency control. The DM is responsible for management of the database itself, i.e., accessing it. Two data manipulation operations are recognized by the DM: DM_READ(X) -- in which data item X is read; and, DM_WRITE(X,NEW_VALUE) -- in which the value NEW_VALUE is assigned to data item X in the database.

Users of the database management system interact with it by running transactions. The TM maintains a private workspace for each active transaction in
which copies of records read or written by the transaction are kept. From the
database management system point of view a transaction executes four types of
requests: TRANS, READ, WRITE, and SNART. The actions taken by the TM upon
receipt of these commands are described below.

TRANS: The TM initializes a private workspace for the transaction.

READ(X): X is a data item. If X already exists in the private workspace then its
value is returned to the transaction by the TM. Otherwise the TM issues
a DM_READ(X) operation to the DM. The current value of X is returned
to the TM which writes it in the transaction's private workspace and
returns it to the transaction.

WRITE(X,NEW_VALUE): X is a data item. NEW_VALUE is a value to be assigned
to X. The TM executes a PRE_WRITE(X,NEW_VALUE) operation on the
transaction's private workspace. This has the effect of updating the
previous value of X in the private workspace to NEW_VALUE if a copy of
X existed in the private workspace. Otherwise, X is created in the
workspace with the value NEW_VALUE. Note that a PRE_WRITE opera-
tion does not alter any values in the database itself. From a synchroni-
zation point of view each WRITE causes a PRE_WRITE to be executed.

SNART: The TM issues a DM_WRITE(X) operation for every previously executed
PRE_WRITE(X) operation. This has the effect of making the last change
to X in the private workspace a (temporarily) permanent value in the
database. After all DM_WRITEs have been issued the private workspace
is discarded – the transaction has completed. From a synchronization
point of view all the transaction's DM_WRITEs are executed atomically.

A transaction execution can be seen as composed of two phases. The execution phase, where the transaction reads values from the database, performs
various computations and creates local values to data items in its private
workspace. And the commit phase, which takes place after the transaction
finishes all computations, where the transaction updates the database by mak-
ing its local values global. It is important to realize that this second phase is
atomic.  

The physical implementation of the commit procedure need not be atomic as long as it ap-
ppears atomic to the outside world. Kung and Robinson [KUNG81] discussed several ways of imple-
menting non-atomic commits. In this paper we will refer to atomic commits from a logical point of
view, we ignore issues of physical implementation.

---
The TM may 

**restart** a transaction T any time in its *execution phase* (before a DM_WRITE has been processed). The effect of restarting T is to obliterate its private workspace and to execute T from the beginning.

All references to data items in the private workspace are made through the TM. Thus the notion of a (logical) private workspace presented in this paper differs from that used by Network based database management system, where the user program "contains" its own private workspace (or user work area) which can be accessed at any time independently of the database management system.

### 2.2. Serializability

In this section we review the basic serializability theory results and show that our use of PRE_WRITE as a synchronization primitive does not affect known results. We also introduce several new precedence relations to be used later in the proof of correctness.

**Definition 1**: Let \( T = \{T_1, T_2, \ldots, T_\rho\} \) be a set of transactions. E, the *execution schedule* of \( T \), is modeled by \( L^S \), the *synchronization log* of \( T \), which consists of DM_READ, PRE_WRITE, and DM_WRITE operations in the order in which they were scheduled. L, the *execution log* of \( T \), is derived from \( L^S \) by removing from it all PRE_WRITE operations.

In subsequent lemma and theorem statements we shall assume that \( T, E, L, \) and \( L^S \) as defined in Definition 1 are given. Furthermore, references to \( T_i \) and \( T_j \) are to any two transactions in \( T \). A DM_READ(X) operation by transaction \( T_i \) will be denoted by \( r_i[X] \). Similarly, \( p_i[X] \) and \( w_i[X] \) will denote PRE_WRITE and DM_WRITE operations (respectively) on X made by transaction \( T_i \). Finally, \( 0_i[X] < 0_j[X] \) means that \( 0_i[X] \) precedes \( 0_j[X] \) in \( L^S \).

**Definition 2**: \( L^S \) is serializable if it is computationally equivalent to a serial synchronization log.

---

3 For the sake of brevity we eliminate the definitions of the various terms, such as computational equivalence.
Proposition 1: \( L^S \) and \( L \) are computationally equivalent.

Proof: Both logs contain the same sequence of \( \text{DM\_READ} \) and \( \text{DM\_WRITE} \) operations. \( \text{PRE\_WRITE} \) operations in \( L^S \) do not change database state.

Theorem 1: \( L^S \) is serializable iff \( L \) is serializable.

Proof:
(a) First we show that if \( L^S \) is serializable then \( L \) is serializable.
(1) Since \( L^S \) is serializable it is computationally equivalent to some serial synchronization log \( L_1^F \).
(2) Let \( L_1 \) be the serial execution log derived from \( L^S \) by deleting all \( \text{PRE\_WRITE} \) operations from it. From proposition 1 it follows that \( L_1^S \) is computationally equivalent to \( L_1 \).

Since \( L \) is computationally equivalent to \( L_1 \) (proposition 1), it follows from (1) and (2) that \( L \) is computationally equivalent to some serial execution log \( L_1 \).

(b) Next we show that if \( L \) is serializable then \( L^S \) is serializable.
(3) Since \( L \) is serializable it is computationally equivalent to some serial execution log \( L_1 \).
(4) \( L_1 \) is computationally equivalent to a serial synchronization log \( L_1^F \) derived from \( L_1 \), by augmenting each transaction \( T_i \) before its first \( \text{DM\_WRITE} \) operation, with a set of \( \text{PRE\_WRITE} \) operations as follows: for every \( w_i[X] \) in \( L_1 \) add a \( p_i[X] \) operation to \( L_1^F \).

Since \( L_1^S \) is computationally equivalent to \( L \) (proposition 1) it follows from (3) and (4) that \( L^S \) is computationally equivalent to some serial synchronization log \( L_1^F \).

Theorem 1 establishes that to obtain serializable synchronization logs it is sufficient to maintain serializable execution logs. That is, use of the \( \text{PRE\_WRITE} \) operation as a synchronization primitive while maintaining serializable execution logs does not affect the basic theory.

Definition 2: For every pair of transactions \( T_i, T_j \) and a data item \( X \), define the binary relations \( \rightarrow_{pu} \), where values for \( u \) are given below, as follows:

1. \( T_i \rightarrow_{rp} T_j \) if \( r_i[X] < p_j[X] \) in \( L^S \)
2. \( T_i \rightarrow_{pr} T_j \) if \( p_j[X] < r_i[X] < w_j[X] \) in \( L^S \)
3. \( T_i \rightarrow_{rpr} T_j \) if \( T_i \rightarrow_{rp} T_j \) or \( T_i \rightarrow_{pr} T_j \)
The binary relations (4)-(10), are exactly those defined in [BERN81]. The remaining relations are new relations introduced in this paper based on our use of the PRE_WRITE as a synchronization primitive. Clearly all of these relations can be derived from $L^8$.

**Theorem 2 (Decomposition):** Let $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ be associated with an execution schedule $E$ modeled by $L^8$. $E$ is serializable if

1. $\rightarrow_{rwr}$ and $\rightarrow_{ww}$ are acyclic, and
2. There is a total ordering of the transactions consistent with all $\rightarrow_{rwr}$ and all $\rightarrow_{ww}$ relationships.

**Proof:** (1) and (2) guarantee the serializability of the execution log [BERN81]. This fact and Theorem 1 guarantee also the serializability of $L^8$.

Lemmas 1 will be used subsequently. It formally defines an important fact, that by means of the PRE_WRITE operation we are able to foresee future conflicts in $\rightarrow_{rwr}$ relation.

**Lemma 1:** $\rightarrow_{rpr} \supset \rightarrow_{rw}$

**Proof:** If $T_i \rightarrow_{rw} T_j$ then by Definition 2 there exists a data item $X$ such that $r_i[X] < w_j[X]$ in $L^8$.

In $L^8$ each $w_j[X]$ is preceded by $p_j[X]$ i.e., $p_j[X] < w_j[X]$.

1. if $r_i[X] < p_j[X] < w_j[X]$ then $T_i \rightarrow_{rpr} T_j$
2. if $pj[X] < r_i[X] < w_j[X]$ then $T_i \rightarrow_{rpr} T_j$

from (1) and (2) it follows that if $T_i \rightarrow_{rw} T_j$ then $T_i \rightarrow_{rpr} T_j$

**Lemma 2:** If $T_i \rightarrow_{ww} T_j$ or $T_i \rightarrow_{wr} T_j$ then $T_i$ committed before $T_j$.

**Proof:** $T_i \rightarrow_{ww} T_j$ or $T_i \rightarrow_{wr} T_j$ implies $w_i[X] < w_j[X]$ or $w_i[X] < r_j[X]$ in $L^8$, i.e., $T_j$ was active after $T_i$ executed a DM_WRITE operation. Since all DM_WRITE operations of a transaction are executed atomically, in its commit phase, with no interleaving operations of other transactions, we have that $T_i$ committed before $T_j$. 

Technion - Computer Science Department - Technical Report CS0338 - 1984
Theorem 3. $\rightarrow_{\text{wwr}}$ is acyclic.

Proof: Suppose $\rightarrow_{\text{wwr}}$ is cyclic. Let a cycle in $\rightarrow_{\text{wwr}}$ be of the form $T_i \rightarrow_{\text{wwr}} T_j \rightarrow_{\text{wwr}} T_k \rightarrow_{\text{wwr}} \cdots \rightarrow_{\text{wwr}} T_i$. Using Lemma 2 directly and transitively we have that $T_i$ committed before $T_j$ and $T_j$ committed before $T_k$, which is contradictory since commit is last atomic action executed by transaction. Hence, our assumption is false and $\rightarrow_{\text{wwr}}$ is acyclic.

Since restarted transactions leave no traces in the database and do not affect the consistency of other transaction, from serializability viewpoint we may rule out all restarted transactions.

Corollary 1: Let $\rightarrow$ be associated with an execution schedule $E$ modeled by $L^S$. $E$ is serializable if:

$$T_i \rightarrow_{\text{rw}} T_j \text{ implies } T_i \text{ committed before } T_j.$$

Proof: If the above condition holds, using Lemma 2 we have:

if $T_i \rightarrow_{\text{rw}} T_j$ or $T_i \rightarrow_{\text{wr}} T_j$ or $T_i \rightarrow_{\text{ww}} T_j$ then $T_i$ committed before $T_j$.

Using a similar argument to that in theorem 3 we get that $\rightarrow$ is acyclic.

The Decomposition theorem suggests that we can design concurrency control methods consisting of:

1. An $\text{rw}$ synchronization technique which amounts to synchronize READ-WRITE conflicts.
2. A $\text{ww}$ synchronization technique which amounts to synchronize WRITE-WRITE conflicts.
3. An interface between the above synchronization techniques which attains a total order of the transactions consistent with all $\rightarrow$ relations.

The atomic commit, however, guarantee a priori the acyclicity of $\rightarrow_{\text{ww}}$ and suggests that we can design simpler algorithms which can do without $\text{ww}$ synchronization. This is formulated by Corollary 1. One such an algorithm is Kung & Robinson’s serial validation [KUNGB1] and its timestamp version in [CAREB3].

3. The private Workspace 2PL (W2PL) algorithm

In this section we examine in detail a TM implemented as a W2PL scheduler.

We describe the data structures involved as well as the operations performed on them by the W2PL algorithm.
3.1. Data Structures

Two data structures are required by the algorithm for its operation: The Wait For Graph (WFG) to represent the wait-for relation among conflicting transactions. And the global Flag Table (FT) in which a list of transactions and their modes of access to data items is maintained.

A node in WFG represents an active transaction. An edge \((T_i, T_j)\) in the graph indicates that \(T_j\) is waiting for \(T_i\) to complete.

An entry in FT exists for every data item that has been accessed, and consists of several pairs \(<\text{FLAG}, \text{TRANSACTION IDENTIFIER}>\). Each pair identifies the transaction that accessed the data item and the mode of access (Read or Write). No restriction is placed on the number and/or type of pairs associated with a single data item in an entry. It is up to the W2PL algorithm to interpret the pairs in a single entry and to decide how to use that information.

Two types of flags are recognized:
(1) An \(r\)-flag indicates that a \(\text{DM}_\text{READ}\) operation was executed on this data item on behalf of the transaction holding the flag.
(2) A \(p\)-flag indicates that a \(\text{PRE}_\text{WRITE}\) operation was executed on this data item on behalf of the transaction holding the flag.

3.2. The Algorithm

The W2PL scheduler uses blocking with deadlock prevention to synchronize conflicting transaction operations. A transaction must own an \(r\)-flag on data item \(X\) before executing \(\text{DM}_\text{READ}(X)\) operation, and must own a \(p\)-flag on \(X\) before executing a \(\text{PRE}_\text{WRITE}(X)\) operation. However, taking a full advantage of the "temporary write" nature of the \(\text{PRE}_\text{WRITE}\) operation, it differs from the standard 2PL protocol in two major aspects:
(1) \(\text{PRE}_\text{WRITE}\) operations do not conflict.
(2) A transaction executing a \(\text{DM}_\text{READ}\) operation is not required to wait for transactions it precedes in WFG.
A TRANS request causes the W2PL to add a new node to WFG representing the new transaction.

A READ or a WRITE request received by the W2PL undergoes a possibly empty waiting phase before it can be executed. In that phase, the requesting transaction is forced to wait until transactions that "hold" conflicting flags on the same data item have completed execution. Edges are added to WFG to reflect the precedence relation imposed by the waiting. The definition of conflicting flags is given by the Compatibility Flag Table (CFT) in Figure 1.

<table>
<thead>
<tr>
<th></th>
<th>Ti</th>
<th>Tj</th>
</tr>
</thead>
<tbody>
<tr>
<td>W2PL</td>
<td>r</td>
<td>p</td>
</tr>
<tr>
<td></td>
<td>r</td>
<td>+</td>
</tr>
<tr>
<td></td>
<td>Ti</td>
<td>Tj -&gt; Ti</td>
</tr>
<tr>
<td></td>
<td>p</td>
<td>Tj -&gt; Ti</td>
</tr>
</tbody>
</table>

**LEGEND**

+ request granted  
- request not granted  
* if Ti precedes Tj in WFG then request granted else request not granted

Figure 1.

Compatibility Flag Table (CFT), Ti making the request

An entry in the table indicates how the algorithm interprets the values of existing flags on a data item when processing a request from a transaction to access that data item. In addition the entry specifies what edges are added to WFG whenever the request is not granted. If adding an edge causes a cycle in
the graph the transaction that caused the cycle is restarted. It is important to note that it is a nonpreemptive deadlock prevention (see discussion below).

For example, the sequence of operations $P_j(Y) \rightarrow r_i(X) \rightarrow P_j(X) \rightarrow r_i(Y)$ will not lead to deadlock. After the last operation is executed both $T_i$ and $T_j$ will hold flags on $Y$, $T_i$ will hold flag on $X$ and an edge from $T_i$ to $T_j$ will exist in WFG reflecting the fact that $T_j$ is waiting for $T_i$ to terminate. This sequence of operations would lead to deadlock in the conventional 2PL algorithm.

Execution of the request includes appending the appropriate flag to FT and issuing the appropriate DM_READ or PRE_WRITE operation to the DM.

A SNART request triggers the atomic commit phase of the transaction. The TM issues a DM_WRITE operation for each data item the transaction updated by a previous PRE_WRITE operation. After the last DM_WRITE has been executed the TM removes the WFG node and all outgoing edges of the committing transaction, along with all its FT flags.

When a transaction commits the node representing it in WFG is a root node. After committing the transaction, the TM may remove all information about it from WFG and FT. Since the node representing it is a root node removing the node (and its outgoing edges) from WFG will not lead to any loss of information. This is also the case with restarted transactions when using a nonpreemptive deadlock prevention, since we restart only active nonblocked transactions whose WFG nodes are root nodes. However, if other deadlock prevention or deadlock detection schemas are used, nodes representing restarted transactions may not be root nodes and as such may not be removed immediately from WFG. As soon as such a node becomes a root node all information about the restarted transaction may be removed from WFG and FT.
4. Correctness

In this section we show that W2PL allows only serializable execution schedules and avoids deadlock situations. The correctness of the algorithm is based on the consistency between the waiting relation induced by WFG and \(-\rightarrow_{rpr}\) relation induced by \(I^S\).

**Lemma 3:** Let \(E\) be execution schedule of \(T\) using W2PL and modeled by \(I^S\) with its associated \(-\rightarrow\) relation. If \(T_i \rightarrow_{rw} T_j\) then \(T_i\) committed before \(T_j\).

**Proof:** Suppose Lemma 3 is untrue i.e., \(T_i \rightarrow_{rw} T_j\) and \(T_j\) committed before \(T_i\).

By Lemma 1, \(T_i \rightarrow_{rpr} T_j\) implies that \(T_i \rightarrow_{rpr} T_j\).

(1) If \(T_i \rightarrow_{rp} T_j\) then by definition there exists a data item \(X\) such that \(ri[X] < pj[X]\). Since \(T_j\) committed before \(T_i\), \(T_i\) must have been an active transaction owning an r-flag on \(X\) while \(T_j\) executed \(pj[X]\), contradicting the fact that \(T_j\)'s \(PRE\_WRITE(X)\) request could not be granted by W2PL.

(2) If \(T_i \rightarrow_{pr} T_j\) then by definition there exists a data item \(X\) such that \(pj[X] < ri[X] < w_j[X]\). The W2PL algorithm could have allowed the execution of \(ri[X]\) operation while \(T_j\) was active and owned a p-flag on \(X\), only if \(T_i\) preceded \(T_j\) in WFG. But if this was the case then \(T_j\) must have been waiting for \(T_i\) to terminate and release its flags, in particular the one \(T_j\) was waiting on. Thus, \(T_j\) cannot commit before \(T_i\).

From (1) and (2) it follows that, if \(T_i \rightarrow_{rw} T_j\) then \(T_j\) could not have committed before \(T_i\), which completes our proof.

**Theorem 4:** W2PL allows only serializable execution schedules and does not deadlock.

**Proof:** Let \(E\) be execution of \(T\) using W2PL and modeled by \(I^S\) with its associated \(-\rightarrow\) relation.
Lemma 3 satisfies Corollary 1 hence, E is serializable.

Whenever $T_i$ waits for $T_j$ an edge $(T_i, T_j)$ is added to $WFG$. Since $WFG$ is maintained acyclic at all time a deadlock is prevented.

5. Avoiding deadlock on a READ request

In the previous section we showed that a transaction $T_i$ executing $READ(X)$ is not required to wait for transactions which previously executed $PRE_WRITE(X)$, and follow $T_i$ in $WFG$. Thus when the request is not granted no cycle will be created in $WFG$ due to waiting for conflicting transactions. However, when $T_i$ is added to the wait queue for $X$ a cycle in $WFG$ may develop on waiting for transaction in the queue.

In this section we present a FIFO type queue management algorithm, consistent with $WFG$, and prove that by using it no READ request will ever deadlock.

5.1. Management of WAIT QUEUE for a data item consistent with WFG

The management of WAIT QUEUE for a data item is an implementation policy decision which does not affect basic correctness of the algorithm. Algorithm correctness is based on synchronization with actual executed requests, thus FIFO, PRIORITY or any other queue management policy are acceptable (as long as nonpreemptive deadlock prevention is maintained).

Let $T_i$ be a transaction to be added to the wait queue for some data item $X$. In case FIFO policy we may synchronize $T_i$ with the other transactions on the queue immediately upon inserting $T_i$ at the end of the queue (by adding to $WFG$ edges $(T_j, T_i)$ for each $T_j$ on the queue). In this policy the order of transaction arrival is maintained. However, in general, transactions may pass each other in the queue. This can be achieved by reorganizing the queue periodically; possibly upon arrival of a new request or upon removal of a flag held on $X$ from $FT$. In any case, the order of transactions in the wait queue must be CONSISTENT with the
order of WFG, or else unnecessary transaction deadlocks may occur.

For example:
Let \( T = \{ T_1, T_2, T_3, T_4 \} \) be defined as follows:

\[
\begin{align*}
T_1 &: \text{TRANS READ}(Y); \text{WRITE}(Y); \text{READ}(X); \text{SNART} \\
T_2 &: \text{TRANS WRITE}(X); \text{WRITE}(Y); \text{SNART} \\
T_3 &: \text{TRANS READ}(Z); \text{READ}(X); \text{SNART} \\
T_4 &: \text{TRANS READ}(Z); \text{WRITE}(X); \text{SNART}
\end{align*}
\]

Round Robin execution of \( T \), after \( T_4 \) executing \( \text{WRITE}(X) \) will create the following \( L^S \) and WFG:

\[
\begin{align*}
L^S &= r_1(Y)p_2(X)r_3(Z)r_4(Z)p_1(Y)p_4(X) \\
\text{WFG} &= \{ (T_1, T_2), (T_2, T_3) \}
\end{align*}
\]

where \( T_2 \) is placed on \( Y \)'s wait queue and \( T_3 \) on \( X \)'s wait queue. Execution follows by \( T_1 \)'s \( \text{READ}(X) \). request. Since it is not granted (because of conflict with \( T_4 \) ) \( T_1 \) is placed on \( X \)'s wait queue. If the FIFO policy is used for wait queue management then \( T_1 \) will follow \( T_3 \) in the wait queue for \( X \), and will deadlock when the edge \( (T_3, T_1) \) is added to WFG.

However, if \( T_1 \) precedes \( T_3 \) in the wait queue for \( X \) as WFG dictates, execution of \( T \) will complete successfully without deadlocks.

**Algorithm 1**: FIFO type QUEUE management consistent with WFG.

**Input**: Transaction \( T_i \) and a queue of transactions - \( Q \):

**Output**: \( Q \) augmented by \( T_i \) such that: for every \( T_j \) in \( Q \), if \( T_i \) precedes \( T_j \) in WFG then \( T_i \) precedes \( T_j \) in \( Q \).

**Method**: Execute procedure \( \text{AddTransactionToQ} \).

\[
\begin{align*}
\text{procedure AddTransactionToQ}(T_i) \; \\
&\{ \\
T_i &\leftarrow \text{Start transaction on } Q \\
&\text{while } T_j \text{ is defined } \\
&\{ \\
&\text{if } T_i \text{ precedes } T_j \text{ in WFG }
\}
\end{align*}
\]
{{
    insert T_i before T_j
    stop
} else T_j <- Next transaction on Q;
    add T_j to end of Q
}

Lemma 4: Algorithm 1 guarantees that for every pair of transactions T_k, T_j on Q, if T_k precedes T_j in WFG then T_k precedes T_j on Q.

Proof: Trivial if we note that WFG is maintained acyclic at all times.

Lemma 5: Let T be a set of transactions using W2PL with Algorithm 1. No transaction will deadlock on READ request.

Proof: Let T_i be a transaction executing READ(X) request.

Also let WRITERS(X) = \{ T_k \mid T_k holds a p-flag on X \}
and let WAITERS(X) = \{ T_k \mid T_k is on X's wait queue \}

T_i may wait only for transactions in WRITERS(X) or WAITERS(X). We will show that no such transaction may cause T_i to deadlock. Suppose T_i causes a cycle in WFG by adding an edge (T_j, T_i). It follows that T_i precedes T_j in WFG.

(1) T_j cannot belong to WRITERS(X) because if it did then W2PL would grant T_i's READ request upon conflict with T_j and no new edge would be added to WFG.

(2) T_j cannot belong to WAITERS(X) since algorithm 1 would insert T_i before T_j in X's wait queue.
6. The No Blind Writes (NBW) Assumption

Under the NBW assumption a transaction must read first a data item before attempting to update it. In this section we will show that under this assumption READ requests will not cause transactions to starve. Combining this and the fact that no transaction deadlocks on a READ request, W2PL guarantees that under the NBW assumption no Reader will be restarted or starved.

The NBW assumption is defined formally as follows:

\[ \forall X \in \text{Writeset}(T_i) \quad r_i[X] < p_i[X] \text{ in } L^S. \]

Lemma 6: Under the NBW assumption there is at most a single writer owning a p-flag on a data item.

Proof: Let \( T_j \) be the first writer to own a p-flag on some data item \( X \). Since \( T_j \) already owns a p-flag on \( X \) it must have executed a \( P_j[X] \) operation which under the NBW assumption was preceded by an \( r_j[X] \). Thus, \( T_j \) must also own an r-flag on \( X \).

Let \( T_k \) be a new writer attempting to execute \( \text{PRE}-\text{WRITE}(X) \). Under the NBW assumption \( T_k \) must have already executed \( r_k[X] \)-operation and own an r-flag on \( X \). However, W2PL would have allowed execution of \( r_k[X] \) while \( T_j \) was active and owned a p-flag on \( X \), only if \( T_k \) preceded \( T_j \) in WFG. Since \( T_k \) requests a p-flag on \( X \) while \( T_j \) owns an r-flag on \( X \), W2PL will suspend \( T_k \) and restart it immediately because the added edge \((T_j,T_k)\) creates a cycle in WFG.

Lemma 7: Under the NBW assumption no transaction will starve on a READ request.

Proof: Let \( T_j \) be the single writer owning a p-flag on \( X \) as defined by Lemma 6. At any time before \( T_j \) completes, every transaction \( T_k \) attempting to execute \( \text{PRE}-\text{WRITE}(X) \) operation will be restarted by the argument used in the proof of
Lemma 6. As a consequence, all transactions on X's wait queue request ownership of an r-flag on X (while processing a READ(X) request). When T_j completes and its p-flag on X is removed, all pending requests on X's wait queue may be applied and granted.

Theorem 5: Let T be a set of transactions using W2PL. Under the NBW assumption no Reader will be restarted or starved.

Proof: Follows immediately from Lemma 5 and Lemma 7.

7. Conclusions

7.1. Summary of Contributions

In this paper we presented the private workspace model as a framework for the design of concurrency control algorithms without synchronization. We have proposed the W2PL algorithm in which write requests do not conflict and read requests do not cause transactions to restart. Under the No Blind Writes assumption W2PL also guarantees that read requests will not cause transactions to starve, thus no Reader will be restarted or starved.

The key ideas proposed in this paper are:

1. The atomic commit guarantee a priori the acyclicity of $\rightarrow_{\text{wrt}}$ relation.
2. The use of the PRE_WRITE operation as a synchronization primitive in addition to the previously used DM_READ and DM_WRITE operations. Conflicts involving a PRE_WRITE operation essentially foresee possible conflicts between DM_READs and DM_WRITEs. This information may be used by the concurrency control to avoid unnecessary cycles in WFG.
3. The management of wait queue for a data item consistent with WFG avoids unnecessary deadlocks.

7.2. Related Work

Other authors, [BERNB0b] and [ROB182], proposed algorithms that employ Thomas Write Rule [THOM79] to “decouple” writes to the database. Essentially,
the rule enables ignoring a WRITE to a data item for which a "later" version exists. However, since this is achieved, to one degree or another, by a timestamp mechanism, we view both proposals as employing a timestamp-based synchronization. Moreover, they cannot guarantee that read requests will not cause transactions to restart.

Bayer et al [BAYE80] presented a multiversion locking algorithm, using PRE_WRITEs with an idea similar to ours. Their algorithm do not use the private workspace model, however, it takes the advantage that for recovery reasons we often have two version for a data item: the old committed one and the newly prepared one, and selects the version to read, without transaction delay, on ground if it causes a cycle in the Serialization Graph or not. In their algorithm READ request is always granted and never causes a transaction to restart. However, writers are severely restricted, since only one write request can be executed simultaneously on the same data item.

8. References


