マルチメディア通信と分散処理 77－9
（1996．7．12）

# 書き込み保留を用いた逐次化グラフスケジューリング

多田 知正　　樋口 昌宏　　藤井 護

大阪大学 基礎工学部 情報科学科

データベースの無矛盾性を保存するようなスケジューリングアルゴリズムがいくつか知られている．その中で逐次化グラフ検査を用いたスケジューリングアルゴリズムはトランザクションの高い並行性を達成することが知られているが，この方法にはいくつかの問題点がある．すなわち，逐次化グラフの検査に時間がかかるためデータベースに対する操作が遅れてしまうことと，生成される実行がサイト故障などの障害から回復できない場合が存在することである．一方，楽観的並行制御と呼ばれる方法はすべての操作を直ちに実行できる特徴がある．また局所的コピーを用いて実際のデータベースへの書き込み操作を遅らせることにより，障害からの回復が可能な実行を得ることを可能にしている．そこで，この方法を逐次化グラフを用いたアルゴリズムに適用し，書き込み保留逐次化グラフ検査というアルゴリズムを提案する．このアルゴリズムによって，先に述べた逐次化グラフ検査の問題点を解決することができる．本稿では，このアルゴリズムの基本アルゴリズムといくつかの変形を示す．また，他のアルゴリズムと比較することにより，このアルゴリズムの有効性を示す．

# A Scheduling Method using Serialization Graph Testing with Write Deferring

Harumasa Tada　　Masahiro Higuchi　　Mamoru Fujii

Depertment of Information and Computer Science, Faculty of Engineering Science
Osaka University

Several scheduling algorithms for preserving database consistency are known. It is known that Serialization Graph Testing (SGT) achieves higher concurrency of transactions than other scheduling algorithms. However, it has some drawbacks. First, operations are delayed for a long time to check the serialization graph. Second, executions produced by SGT may be unrecoverable from some failures. On the other hand, under Optimistic Concurrency Control (OCC), all operations are executed immediately. Moreover, OCC produces recoverable executions by deferring substantial write operations using local copies. Therefore, we applied OCC approach to SGT and propose an algorithm which we call Serialization Graph Testing with Write Deferring (SGT-WD). SGT-WD overcomes above drawbacks of SGT. In this paper, we show the basic algorithm and some variant of SGT-WD. We compare SGT-WD with other algorithms, and show that it is more useful than those algorithms.

## 1. Introduction

The concurrency control in database systems is an important problem and it is studied by many researchers. We have studied the scheduling algorithm called *Serialization Graph Testing* (SGT) and proposed a scheduling algorithm for distributed database systems[4]. In SGT, a scheduler maintains a graph called serialization graph (SG) and schedules operations ensuring that the SG remains acyclic. It is known that the SGT scheduler achieves higher concurrency of transactions than other scheduling algorithms[1]. However SGT algorithm has some drawbacks, i.e.:

1. Time to check the serialization graph tends to become long, and an execution of an operation is delayed until the check completes.
2. Executions produced by SGT are not recoverable from failures. The further restriction (for example *strictness* in [1]) should be imposed to executions in practical databases in which several types of failures are possible.

To solve these problems, we focused on schedulers called *certifiers*[1]. A certifier permits all operations to execute immediately. When it is about to schedule a commit operation, it checks whether the execution including the commit operation is consistent. If the ex-

ecution is inconsistent, some transactions are aborted. Since certifiers schedule operations immediately, the processing time of transactions is shorter than other type of schedulers. On the other hand, conflicts are not detected until the transaction is about to commit.

*Optimistic Concurrency Control* (OCC)[2], which is one of certifiers, produces recoverable executions by deferring substantial write operations using local copies. We apply OCC approach to SGT and propose a new algorithm. Since it defers write operations using local copies, we call it *Serialization Graph Testing with Write Deferring* (SGT-WD).

In this paper, we propose a basic algorithm of SGT-WD. We compare SGT-WD with OCC and the usual SGT certifier in some examples. Moreover we show some variants of SGT-WD.

The paper is organized as follows. Section 2 describes the serialization graph testing. In section 3, we describe the overview of certifiers and optimistic concurrency control. Section 4 describes the basic algorithm of SGT-WD. We compare it with other algorithms in section 5. Some variants of SGT-WD are shown in section 6. Conclusions appear in Section 7.

## 2. Serialization graph testing

### 2.1 Serialization graph

The property called *serializability* is widely used criterion for ensuring the correctness of concurrent execution of transactions. Intuitively speaking, for an execution $H$, if there is a serial execution $H_s$ which contains the same operations as $H$, and the relative orders of all pairs of conflict operations in $H$ and $H_s$ are the same, then $H$ is called serializable. The strict definition is described in [1].

The serializability of a concurrent execution of transactions is determined by analyzing a graph derived from the execution, called a *serialization graph*(SG).

**Definition 1:** For a concurrent execution $H$, the *serialization graph* $SG(H) = \langle V, E \rangle$ is a directed graph such that

$V = \{T_i \mid T_i$ is a transaction that is already started in $H\}$

$E = \{(T_i, T_j) \mid$ there exists conflicting operations $o_i \in T_i$ and $o_j \in T_j$ such that $o_i$ is scheduled before $o_j$ where $T_i, T_j \in V\}$ □

Figure 1 shows an example of a serialization graph. The following theorem is shown in [1].

**Theorem 1:** An execution $H$ is serializable iff $SG(H)$ is acyclic. □

### 2.2 Serialization graph testing

*Serialization Graph Testing* (SGT) uses a serialization graph in order to verify the serializability of concurrent execution of transactions. In SGT, a scheduler maintains an SG. The scheduler behaves as follows. Suppose that the SG is acyclic, and the scheduler receives an operation $o$ of a transaction $T$. If a node for $T$ does not yet exist in the SG, the scheduler first adds the node in the SG. It adds an edge from $T_j$ to $T$ for every previously scheduled operation $q$ of the transaction $T_j$ that conflicts with $o$. Two cases are possible:

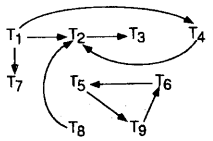


Figure 1 An example of a serialization graph

1. The resulting SG contains a cycle. In this case, the scheduler rejects $o$. The scheduler aborts the transaction $T$, and deletes $T$ from the SG and all edges incident with $T$. Deleting $T$ makes the SG acyclic again, since all the cycles involves $T$. The aborted transaction $T$ will restart later.

2. The resulting SG is still acyclic. In this case, the scheduler can accept $o$. It can schedule $o$ immediately.

In above algorithm, the nodes for committed transactions are never deleted from the SG. If this algorithm is used, as time goes on, the size of the SG grows larger and larger and so does the cost of maintaining the SG. The SGT scheduler must delete nodes and edges for committed transactions which are already unnecessary for scheduling. The detail of the method to delete unnecessary nodes from the SG is out of focus of this paper and is not mentioned here.

The major advantage of SGT is that the consistency checking in SGT is based strictly on the definition of serializability. Therefore, it achieves higher concurrency than other scheduling algorithms, for example, *Two Phase Locking* (2PL) or *Timestamp Ordering* (TO)[1].

On the other hand, SGT has some disadvantages. First, it takes time to check the serialization graph. The checking time tends to become long on distributed database systems[4]. In the case of distributed database system, intersite communication is needed for the checking on distributed system because the serialization graph has global structure. Each operation must wait until the check completes. Therefore, lifetime of transactions may become too long. Second, executions produced by SGT are not recoverable. To use SGT in practical database in which several types of failures are possible, a further restriction must be imposed to executions. For example, *strictness* is a widely used condition for recoverablity [1].

## 3. Certifiers

### 3.1 Overview of certifiers

To overcome the drawbacks of SGT, we focused on schedulers called *certifiers*.

In many scheduling algorithms, every time it receives an operation, a scheduler decides whether to accept, reject, or delay it. On the other hand, a different approach

is proposed. That is, the scheduler immediately schedules each operation it receives. From time to time, it checks to see what it has done. If it concludes that all is well, it continues scheduling. If it detects that it has inappropriately scheduled conflicting operations, it aborts some transactions. Such schedulers are called *certifiers*. Certifiers aggressively schedule operations, hoping no conflicts will happen. Therefore, the processing time of transactions is shorter than other schedulers. On the other hand, operations are scheduled even if they cause loss of integrity and it is not detected until the explicit check which is usually done at the end of the transaction. Therefore, in the case that conflicts happen frequently, lifetime of transactions under certifiers may be much longer than that under other type of scheduler. Most certifiers are constructed as variants of normal type schedulers. For example, there are certifiers based on Two Phase Locking (2PL), Timestamp Ordering (TO) and Serialization Graph Testing (SGT)[1]. The executions produced by such schedulers are not recoverable. However there is another type of certifier approach called *Optimistic Concurrency Control* (OCC) which produces recoverable executions.

### 3.2 Optimistic concurrency control

Optimistic Concurrency Control (OCC), proposed by Kung and Robinson[2], is one of certifiers. OCC defers substantial write operations using local copies. In OCC, an execution of a transaction is divided into the following three phases.

**read phase:** In this phase, all read operations are executed immediately. They are completely unrestricted. All write operations take place on local copies which cannot be accessed by other transactions.

**validation phase:** In this phase, it is checked whether the changes the transaction made will cause inconsistency of database. If not, the validation is successful. Otherwise it fails.

**write phase:** In this phase, the data items in local copies are written into real database. At this point of time, the modification made by the transaction become effective. The transaction commits at the end of this phase.

All transactions first enter the read phase. When a transaction is about to execute a commit operation, it enters the validation phase. If the validation succeeds, the transaction enters the write phase and it is committed. Otherwise, the transaction will be aborted and restarted.

In OCC, in order to verify that serializability is preserved, the scheduler explicitly assigns each transaction a unique integer called *transaction number* $t(i)$ during the course of its execution. The meaning of transaction numbers in validations is the following: there must exist a serially equivalent execution in which transaction $T_i$ comes before transaction $T_j$ whenever $t(i) < t(j)$. Transaction numbers are assigned at the end of the read phase. In the validation phase, the validation condition is checked. For a transaction $T$, we define $readset(T)$ and $writeset(T)$ as follows.

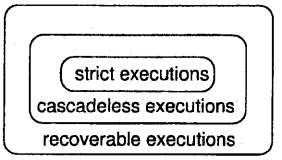


Figure 2 Relation among three classes of executions

$readset(T)$: the set of data items which have been read by $T$

$writeset(T)$: the set of data items which have been written by $T$

For each transaction $T_j$ with transaction number $t(j)$, and for all $T_i$ with $t(i) < t(j)$; one of the following three conditions must hold.

1. $T_i$ completes its write phase before $T_j$ starts its read phase.
2. $writeset(T_i) \cap readset(T_j)$ is empty and $T_i$ completes its write phase before $T_j$ starts its write phase.
3. $writeset(T_i) \cap (readset(T_j) \cup writeset(T_j))$ is empty and $T_i$ completes its read phase before $T_j$ completes its read phase.

If none of above three conditions hold for some $T_i$, the validation of $T_j$ fails.

OCC has two merits. First, operations are scheduled immediatedly. On the other hand, as other certifiers, there are some cases in which abortions of transactions occur frequently under OCC too. Second, abortion of a transaction can be done easily. Write operations are executed on real database only when the transaction commits. Therefore, when a transaction aborts, there are no data items modified by the transaction. Accordingly, no more abortions are caused by the abortion. That is, all executions produced by OCC cause no cascading aborts. We call such executions *cascadeless*. It is also said that the executions *avoid cascading aborts*. As shown in figure 2, the class of *cascadeless* executions is included in that of *recoverable* executions[1].

The OCC scheduler explicitly assigns a transaction number to each transaction. This is regarded as a kind of timestamp. In this meaning, OCC scheduler is considered as a certifier based on TO scheduler. However, there are some difference between OCC and pure TO certifier. First, OCC uses local copies to defer write operations until the validation completes, while TO execute them immediately. Second, OCC and TO certifier differ in the way to check consistency of the database. TO checks the consistency according to the order of operations, while OCC checks the consistency according to overlapping of concurrent executions.

**Table 1** The basic SGT-WD algorithm

when a transaction $T$ is in read phase

```
1 when received a read operation read(x)
2   add edges to the serialization graph
3   read x from the database
4 when received a write operation write(x)
5   write x to a local copy
    (do not add edges to the serialization graph)
6 when received a commit operation commit
7   T enters validation phase
```

when $T$ is in validation phase

```
1 add edges which is caused by write operations to
  the serialization graph
2 check the serialization graph whether there is a
  cycle
3 if there is a cycle then
4   remove the node T and incident edges from the
    serialization graph
5   abort and restart T
6 else
7   T enters write phase
```

when $T$ is in write phase

```
1 write all data written by the transaction in the
  local copies to real database
2 commit T
```

## 4. Algorithm

We apply OCC approach to SGT and propose a new algorithm which overcomes the disadvantage of SGT. The important points are the following:

1. As certifiers, operations are scheduled immediately and they are validated later. Therefore, the processing time of transactions is shorter than non-certifiers.

2. Write operations are deferred until the validation completes. This makes executions produced by this algorithm cascadeless.

We call the algorithm which we propose *serialization graph testing with write deferring*(SGT-WD). We show basic SGT-WD algorithm in table 1. As OCC does, SGT-WD also divides an execution of a transaction into three phases.

## 5. Comparison

In this section, we show some examples of concurrent executions in figures. In the figures, following symbols are used.

- S means the start of the transaction.
- R(x)(W(x)) means the read(write) operation to data item $x$.
- V means the validation. In OCC and SGT-WD, it includes the validation phase and consequent write phase.

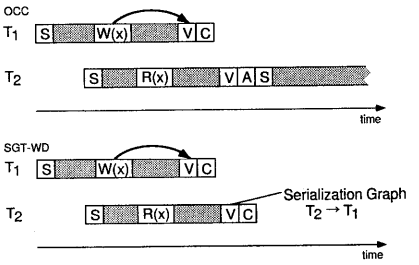


Figure 3 Comparison between OCC and SGT-WD

- C means the commit of transaction.
- A means the abort of transaction.

### 5.1 Comparison with optimistic concurrency control

Figure 3 shows an execution produced by OCC and SGT-WD. In the case of OCC, since $readset(T_2)$ and $writeset(T_1)$ conflict, condition 1 in section 3.2 must hold in the validation of $T_2$. However, $T_2$ starts its read phase before $T_1$ completes its write phase. Therefore, the validation fails and $T_2$ is aborted and restarted.

In the case of SGT-WD, since conflicts between $T_1$ and $T_2$ cause no cycles in the SG, the validation is successful and $T_2$ is committed. In this situation, the processing time of $T_2$ under SGT-WD is shorter than that under OCC.

Generally, suppose that $read(x)$ of a transaction $T_i$ is scheduled after $write(x)$ of a committed transaction $T_j$. Under OCC, if $T_i$ has been started before $T_j$ was committed, the validation of $T_i$ fails and $T_i$ must be aborted. In the case of SGT-WD, the validation of such $T_i$ succeeds unless $T_i$ writes the same data item. Such a situation happens frequently with database systems in which processing time of transactions tend to become long. This fact shows the advantage of SGT-WD over OCC. The class of executions produced by SGT-WD contains properly than that of OCC because the validation of SGT-WD is based strictly on the definition of serializability, which is a correctness criterion both algorithms use.

### 5.2 Comparison with serialization graph testing certifier

Figure 4 shows an execution produced by the usual SGT certifier and SGT-WD when a certain failure occurred. Consider the case of the usual SGT certifier first. A transaction $T_1$ is aborted by the failure, then $T_2$ which reads the data item $x$ written by $T_1$ must be aborted, that is, cascading abort occurs. However, since $T_2$ has already been committed when $T_1$ is aborted, $T_2$ cannot be aborted and the database cannot recover to consistent state any longer. This fact shows that the usual SGT certifier may produce executions which is not recoverable. To make the execution recoverable, the usual SGT certifier should defer the commit of $T_2$ until $T_1$ is

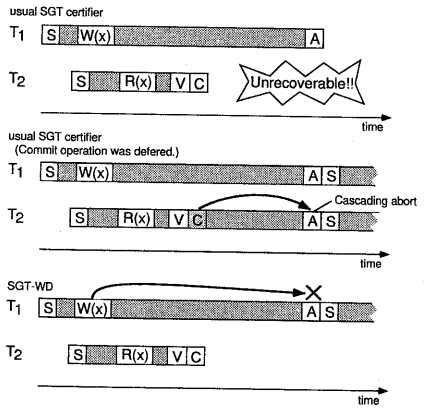


Figure 4 Comparison between the usual SGT certifier and SGT-WD (when a failure occurred)

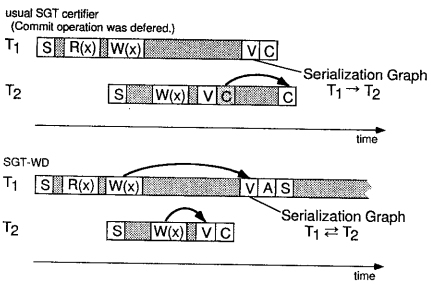


Figure 5 Comparison between the usual SGT certifier and SGT-WD (when no failures occurred)

committed or aborted (as shown in the middle of the figure 4). On the other hand, SGT-WD avoids such an execution by deferring the write operation of $T_1$. Under SGT-WD, $x$ is not modified when $T_2$ reads it, while $T_2$ reads $x$ which is modified by $T_1$ under the usual SGT certifier. Therefore, under SGT-WD, $T_2$ don't have to abort when $T_1$ is aborted. Clearly, SGT-WD is preferable to the usual SGT certifier in this case. Consider another execution in figure 5. At first sight, $T_1$ and $T_2$ don't form a cycle. Under SGT-WD, however, the cycle is formed because the real execution of W(x) of $T_1$ is deferred until the write phase. Therefore $T_1$ is aborted and restarted. In the case of the usual SGT certifier, such a cycle is not formed and $T_1$ is not aborted. Therefore, it may be concluded that the usual SGT certifier has an advantage over SGT-WD in this case. However, under the usual SGT certifier, the commit operation of $T_2$ is deferred until $T_1$ is committed. This means that a transaction waits another transaction. That is, the merit of certifiers, they aggressively schedule operations, is damaged. On the other hand, no transactions wait other transactions in SGT-WD. In this case, it depends on the circumstance whether SGT-WD is preferable to the usual SGT certifier.

**Table 2** Edge addition procedure

procedure *add-read-edge*$(T, x)$
(add edges for *read*$(x)$ of transaction $T$)

```
1 for any transaction T_i
2   if x ∈ writeset(T_i) then
3     add an edge T_i → T
4 add x to readset(T)
```

procedure *add-write-edge*$(T, x)$
(add edges for *write*$(x)$ of transaction $T$)

```
1 for any transaction T_i
2   if x ∈ readset(T_i) ∪ writeset(T_i) then
3     add an edge T_i → T
4 add x to writeset(T)
```

## 6. Variants

In this section, we describe some variants of SGT-WD.

### 6.1 Additional graph checking

SGT scheduler (not certifier) checks the serialization graph *every* time it receives an operation. In basic SGT-WD, as in SGT certifier, the graph is checked only once during an execution of a transaction. A disadvantage of this method is that conflicts are not detected until the validation phase. We can improve SGT-WD by checking the serialization graph during the read phase as well as the validation phase. The aim of this additional graph checking is not to make sure that an operation can be scheduled immediately but to detect conflicts as soon as possible. Therefore, the scheduler needs not check the graph every time it receives an operation. It is possible that the check is executed once for several operations. Nevertheless, the graph check takes long time as mentioned in section 2.2. If an operation waits for the check to complete, an execution of transaction is suspended for a long time. Practically, the graph check should not even be synchronized with an execution of a transaction except from the check in the validation phase. That is, we can introduce concurrent subprocesses which we call *graph checking processes* (GCP). There exists one GCP for each transaction. When a cycle is detected, the GCP notifies the transaction of the occurrence of the cycle. In this way, a transaction execution is hardly affected by the time needed for the checking. Of course, the checking in the validation phase must be completed before the transaction enters the write phase.

### 6.2 Adding write edges in read phase

The edge addition procedure of SGT is shown in table 2. For each transaction $T$, *readset*$(T)$ and *writeset*$(T)$, defined in section 3.2, are maintained to add edges to the serialization graph.

In SGT-WD, if an edge $T_i \rightarrow T$ is caused by a write operation *write*$(x)$ of a transaction $T$, procedure *add-write-edge*$(T, x)$ is executed. Suppose that it is executed

immediately—in the read phase. By line 4 of *add-write-edge*$(T, x)$, $x$ is added to *writeset*$(T)$. If an operation *read*$(x)$ of $T_j$ are executed later, an edge $T \rightarrow T_j$ will be added by *add-read-edge*$(T_j, x)$. However this edge is not true because *write*$(x)$ of $T$ is deferred until the write phase of $T$. To avoid this, *add-write-edge*$(T, x)$ is executed in the validation phase.

However, the order of conflicting operations of $T_i$ and $T$ does not change even if *write*$(x)$ of $T$ is deferred. Therefore, the edge $T_i \rightarrow T$ *must* be added to the graph unless $T$ or $T_i$ is aborted. To detect a cycle in the serialization graph as soon as possible, it is desirable that such edges are added in the read phase. Practically, it is possible to execute the procedure *add-write-edge*$(T, x)$ in the read phase with a little modification.

In procedure *add-write-edge*$(T, x)$ in table 2, $x$ is added to *writeset*$(T)$ in line 4. If the procedure is executed in the read phase, This step should be skipped. The data item $x$ should be added to *writeset*$(T)$ later in the validation phase.

### 6.3 Shadow transactions

One variant of OCC, called *speculative concurrency control* (SCC), is proposed by Bestavros[3]. Instead of waiting for the validation of a transaction to fail and then restarting the transaction, the SCC algorithm uses additional (or redundant) resources to start on *speculative* corrective measure. Such measures are called *shadow transactions*. Shadow transactions are subprocesses forked by uncommitted transactions. A shadow transaction is forked when a transaction conflicts with other transaction and the conflict may cause inconsistency. Forked shadows are blocked until the transaction is aborted or committed. If the transaction is aborted, one of blocked shadows of it replaces the transaction and resumes executions, while other shadows are still blocked. If the transaction is committed, all blocked shadows of it are aborted. More details of SCC are described in [3].

This SCC approach can be applied to SGT-WD. It contributes to reducing the time needed to restart transactions. A shadow transaction $T_i$-*shadow*$(o)$ is forked when an operation $o$ of a transaction $T_i$ is executed. $T_i$-*shadow*$(o)$ is blocked before executing $o$. If $T_i$ fails the validation, there is a cycle in the serialization graph. $T_i$ must be aborted and restarted. However, it is not necessary that to restart $T_i$ from the start. $T_i$ have only to redo from the operation $o_{cycle}$ which causes an outgoing edge in the cycle, whereby the cycle disappears. Therefore, $T_i$-*shadow*$(o_{cycle})$ replaces $T_i$ and resumes the execution from $o_{cycle}$. An example of an execution is shown in figure 6. In the figure, $T_2$ fails the validation by a cycle formed by $T_1$ and $T_2$. In this case, R(x) of $T_2$ corresponds to $o_{cycle}$ because it causes outgoing edge from $T_2$ ($T_2 \rightarrow T_1$). Therefore, $T_2$-*shadow*(R(x)) replaces $T_2$ and resumes the execution from R(x).

## 7. Conclusions

In this paper, we proposed the scheduling algorithm which we call *Serialization Graph Testing with Write Deferring* (SGT-WD).

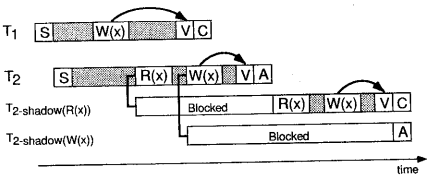


Figure 6 An execution of SGT-WD with shadows

SGT-WD, as OCC does, divides an execution of a transaction into three phases, read phase, validation phase, and write phase. In the read phase, all read operations can be executed, write operations are executed into internal copies which cannot be accessed by any other transactions. That is, write operations are deferred. In the validation phase, the serialization graph is checked its acyclicity. Write operations to real database are executed in the write phase which follows the validation phase. The advantages of SGT-WD are as follows:

1. Operations can be executed immediately.
2. Only recoverable executions are produced.

We compared SGT-WD with OCC and the usual SGT certifier and showed that SGT-WD is more useful than these algorithms.

The major disadvantage of SGT-WD is common to all certifiers. That is, transaction conflicts are detected later than noncertifiers. We showed that it is improved at the expense of additional graph checking and this additional checking can be implemented without affecting executions of transactions. Additionally, we described some other variants of SGT-WD. It is our future work to evaluate effectiveness of SGT-WD and its variants quantitatively.

## References


[1] Bernstein, P.A., Shipman, D.W. and Wong, W.S., *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[2] Kung, H.T., Robinson, J.T., "On Optimistic Methods for Concurrency Control", *ACM Trans. Database Systems* Vol.6, No.2, pp.213-226, June 1981.

[3] Bestavros, A. and Braoudakis, S., "Value-cognizant Speculative Concurrency Control", in *Proc. 21st Intl. Conf. on Very Large Data Bases*, Zurich, Swizerland pp.122-133, September 1995.

[4] Tada, H., Terasaki, Y., Higuchi, M. and Fujii, M., "A Scheduling Algorithm using Serialization Graph Testing for Distributed Database System", *The Technical Report of the IEICE*, SS94-41, November 1994.