브로드케스트 무효화 기법을 이용한 인증 프로토콜

A Certification Protocol based on Broadcast Invalidation Approach

조 성 호* 강 민 구**
Sung-Ho Cho Min-Goo Kang

요 약

낙관적 동시성 제어 기법의 성능은 철회율에 매우 민감하다. 재배열 기법을 이용하여 이러한 철회율을 완화 시킬 수 있지만 대부분의 재배열 기법은 그래프나 히스토리를 유지하기 위한 공간적 낭비가 심하다. 이 논문에서는 브로드케스팅 기법에 기반한 효율적인 재배열 기법을 소개한다. CBI라고 불리는 이 기법은 그래프나 히스토리를 유지하기 위한 오버헤드 없이 철회율을 완화시킨다. 성능평가를 통하여 우리가 제안하는 기법이 철회율과 필요없는 연산을 줄인다는 것을 보여준다. 뿐만 아니라, BTS와 THS보다 제안하는 기법이 우수한 성능을 나타낸다는 것을 보인다.

Abstract

The performance of OCC is very sensitive to the transaction abort rate. Even if the abort probability can be reduced by re-ordering, most of re-ordering schemes have space overhead in maintaining a graph or histories. In this paper, we proposed an efficient re-ordering scheme based on a broadcast invalidation mechanism. Our scheme, called CBI, can reduce the abort probability without space overhead in maintaining a graph or histories. By simulation studies, we showed that CBI reduced the abort rate and unnecessary operations. Additionally, we showed that CBI outperforms not only BTS but also TSH with low space overhead.

☞ Keyword : re-ordering schemes, CBI, BTS, TSH, Broadcast Invalidation Approach, Certification Protocol

1. Introduction

Kung and Robinson[1] proposed the optimistic concurrency control scheme(OCC). Three popular variations of OCC are pure OCC(POCC)[1], broadcast OCC (BOCC)[2] and certification protocol based on time-stamp history(TSH)[3]. The POCC scheme aborts a transaction only at its commit time when the validity of its data access is checked. In the BOCC scheme, committing transactions cause abort of conflicting transactions in the middle of their execution. In TSH, the read set of the transaction being validated is compared with the current write time-stamp. If conflicts are detected, the transaction can be committed with re-ordering time-stamp based on read time-stamps and write time-stamp histories.

Consider te following example to compare these three schemes. Suppose that transaction X reads data items D_1 and D_2 and writes D_2, transaction Y reads data items D_2 and D_3 and writes D_3, and transaction Z updates data items D_2 and D_4 as in Figure 1, where $R(.)$ and $W(.)$ represent the read and write operations respectively, C indicates the commit, and the subscripts identify the transaction. Additionally, t denotes the real time.

In POCC, if transaction X is committed at time t_1, transactions Y and Z are aborted at time t_2 and t_3 respectively, since they have read a *"wrong"* version of data D_2. In BOCC, if transaction d is committed at time t_1, transactions Y and Z are aborted at time

* 정 회 원 : 한신대학교 정보통신학부 교수
zoch@hanshin.ac.kr(제1저자)
** 종신회원 : 한신대학교 정보통신학과 부교수
kangmg@hanshin.ac.kr(공동저자)

T_X :	$R_X(D_1)$,	$R_X(D_2)$,	$W_X(D_2)$,	C_X
T_Y :	$R(D_2)$,	$R_Y(D_3)$,	$W_Y(Ds)$,	C_Y
T_Z :	$R_Z(D_2)$,	$W_Z(D_2)$, $R_Z(D_4)$,	$W_Z(D_4)$,	C_Z
Time		t_1	t_2	t_3

(Figure 1) Time Line of Operations of Transactions X, Y and Z

t_1 in order to reduce unnecessary operations. However, POCC and BOCC make spurious aborts. The reason is that transaction Y could actually be committed in TSH, since its effect is equivalent to the serial log of transaction Y committing first and transaction X next. Hence, TSH can reduce the number of aborts due to read-write conflicts by re-ordering. This type of read-write conflict is called a "reorderable read-write conflict", since the conflict can be resolved by re-ordering the time-stamp assignments, and transaction Y is said to be "re-ordered". Also, this type of abort can be reduced by SGT[4] and Multi-versioning[5].

It is clear that re-ordering can enhance the performance of the OCC schemes. However, re-ordering schemes have space overhead. For example, SGT requires substantially more storage for the serialization graph and additional overhead in maintaining the serialization graph and checking for cycle. The multi-versioning scheme keeps multiple versions for each data item in order to provide a consistent view of the database to all active transactions. It also has very high space overhead. TSH maintains one read time-stamp and k write time-stamps ($k \geq 2$) per data item. Already mentioned in [3], The size of k affects the performance of TSH. The main motivation of our approach is to reduce space overhead of re-ordering scheme. Compared with other re-ordering schemes, our scheme maintains only one time-stamp per each data item.

In the above example, although transaction Y can be committed by re-ordering, transaction Z can not be committed, because it accessed write-write conflicting data D_2. It means that transaction Z performs unnecessary operations such as $R_Z(D_4)$ and $W_Z(D_4)$, because aborts happen only in the validation phase in the OCC schemes except for BOCC[6]. Our scheme can reduce the unnecessary operations due to write-write conflict.

The final motivation is to reduce validation time. In TSH, the server accesses N data items to validate a transaction, where N denotes the number of elements in the read set of the transaction. In addition, if a transaction to be re-ordered, the server have to access $N + (M * 2)$ data items, where M denotes the number of elements in the write set of the transaction. In our scheme, not only some read-only transactions but also some updating transactions can be committed without validation based on distributed information for re-ordering. Furthermore, the server accesses only M data items for re-ordering.

In this paper, we propose a certificate protocol called CBI(Certification Protocol based on Broadcast Invalidation). Our scheme adopts a re-ordering scheme based on a broadcast mechanism. In particular, we start with simple algorithm that maintains only one time-stamp and then extend the algorithm to enhance the performance. Although our scheme is very similar to BOCC, it dose not make any spurious aborts that happen in BOCC. In addition, our scheme reduces space and time overhead compared with other re-ordering schemes.

This paper is organized as follows. We explain our scheme and show some examples in Section 2. Section 3 shows the extended algorithm to enhance the performance. Our simulation experiments are discussed in Section 4. Finally, our conclusion is presented in Section 5.

2. Certification Protocol based on Broadcast Invalidation

Recently there has been increasing interest in the development of broadcast protocols such as Amoeba[7], ISIS[8], Transis[9], Total[10] and Totem[11]. Since implementation of an efficient broadcast mechanism is not the main issue of this paper, we assume that the system has an efficient broadcasting mechanism and network is always reliable, i.e., each message is delivered preserving the sending order without loss. To simplify our algorithm, we also assume that all write operations are update(i.e., write-after-read) operations. This assumption is eliminated in Section 3.

2.1 Validation with Broadcasting

In our scheme, we maintains time-stamp $D.T$ for each persistent data D. The time-stamp $D.T$ is the time-stamp of the youngest(i.e., lastest in time) committed transaction which read the data. When a transaction wants to read data D_i, the server sends the data with the current time-stamp $D_i.T$. Hence, a transaction views each data item as a (name, version) pair (i.e., (D_i, $D_i.T$) pair).

For re-ordering, each client maintains the lower bound of re-ordering time-stamp(T^L) and the upper bound of re-ordering time-stamp(T^U). The re-ordering time-stamp(T^R) is always larger than time-stamp T^L and smaller than time-stamp T^U. The initial value for time-stamps T^L and T^U is the smallest time-stamp in the system.

When transaction X is ready to enter its commit phase, the client of the transaction sends to the server a message containing time-stamps $X.T^L$, $X.T^U$, sets $X.S^R$ and $X.S^W$, where $X.S^R$ and $X.S^W$ contain ID's of the data items read and written by transaction X, respectively. When the server receives the message,

```
/* Whenever transaction Y reads a data item Di */
If (Y.T^L < Di.T)
{       Y.T^L = Di.T;
        If (Y.T^U ≠ the initial value and Y.T^L ≥ Y.T^U
        - δ) Abort transaction Y; }
Y.S^R = Y.S^R + {Di};
```

(Figure 2) Check algorithm whenever a transaction accesses a data item

the server assigns a unique committing time-stamp $X.T^C$ to the transaction for certification. After that, transaction X is committed without validation only if time-stamp $X.T^U$ is the initial value, i.e., the server reflects updated data items in set $X.S^W$ to the database, and sets time-stamp $D_i.T$ to $X.T^C$ for each data D_i in set $X.S^R$.(We will describe how transaction X can be committed without validation in Section 2.2.) After that, the server broadcasts an invalidation message to each client that contains time-stamp $X.T^C$ and set $X.S^W$.

When a remote client gets the information, if transaction Y has written some data in set $X.S^W$, then transaction Y is aborted in order to reduce unnecessary operations. Actually, transaction Y has to be aborted in its validation phase, because it has accessed write-write conflicting data. Otherwise, if transaction Y has read some data items invalidated by transaction X, transaction Y can be committed only with re-ordering. Besides, time-stamps $Y.T^L$ and $Y.T^U$ are updated.

Time-stamp T_L is updated by the following rule. Whenever transaction Y reads data D_i, time-stamp $Y.T^L$ is compared with the time-stamp $D_i.T$. If time-stamp $Y.T^L$ is less than the time-stamp $D_i.T$, then time-stamp $Y.T^L$ is set to $D_i.T$. Hence, time-stamp T^L is never decreased after it is changed from the initial value. Whenever time-stamp $Y.T^L$ is increased, time-stamp $Y.T^L$ is also compared with time-stamp $Y.T^U - \delta$, where δ is an infinitesimal quantity. If time-stamp $Y.T^U$ is not the initial value and time-stamp $Y.T^L$ is

```
/* Whenever the client gets an invalidation message
that contains X.T^C and X.S^W */
If (Y.S^W ∩ X.S^W ≠ Ø) Abort transaction Y;
If (Y.S^R ∩ X.S^W ≠ Ø)
{       If (Y.T^U = the initial value) Y.T^U = X.T^C
        Else If (Y.T^U > X.T^C) Y.T^U = X.T^C;
        Y.S^I = Y.S^I + (Y.S^R ∩ X.S^W );
        If (Y.T^L ≥ Y.T^U - δ) Abort transaction Y; }
```

(Figure 3) Check algorithm when a client gets an invalidation message

```
/* Whenever transaction Y writes a data item D_i */
If ( D_i ∈ Y.S^I ) Abort Transaction Y;
Y.S^W = Y.S^W + {D_i};
```

(Figure 4) Check algorithm whenever a transaction writes a data item

larger than or equal to $Y.T^U - \delta$, then transaction Y is aborted, because the client can not find any re-ordering time-stamp for transaction Y. (See Figure 2)

Now, we describe how a time-stamp T^U is updated. When the client of transaction Y gets an invalidation message that contains $X.T^C$ and $X.S^W$, the client checks using the following rules. (See Figure 3)

- For each data D_i in set $X.S^W$, if transaction Y has written data D^j, the client aborts the transaction Y immediately. It means that transaction Y has accessed a write-write conflicting data item, because transaction X has written data D_i and committed.
- For each data D_j in set $X.S^W$, if transaction Y has read data D_j, then the client changes the value of time-stamp $Y.T^U$. Time-stamp $Y.T^U$ is compared with time-stamp $X.T^C$. If time-stamp $Y.T^U$ is the initial value, then time-stamp $Y.T^U$ is set to $X.T^C$. Otherwise, time-stamp $Y.T^U$ is set to time-stamp $X.T^C$ if time-stamp $Y.T^U$ is larger than $X.T^C$. Hence, time-stamp $Y.T^U$ is never increased after it is changed from the initial value. After that, data D^j in set $X.S^W$ and $Y.S^R$ is inserted in to the set $Y.S^I$ to prevent unnecessary operations.
- Whenever time-stamp $Y.T^U$ is changed, if $Y.T^U - \delta$ is less than or equal to $Y.T^I$, then transaction Y is aborted. It means that the client of transaction Y can not find any re-ordering time-stamp $Y.T^R$.

Whenever transaction Y writes on a data item, The client of transaction Y checks whether the data is in set $Y.S^I$, If so, transaction Y is aborted (See Figure 4). However, maintaining set S^I is optional, because transaction Y has to be aborted in the validation phase even if the client dose not abort transaction Y.

2.2 Committing Transactions

Now, we describe how a read-write conflicting transaction can be committed in our scheme. When transaction Y requests a commit, transaction Y can be committed with re-ordering time-stamp $Y.T^R$ ($Y.T^R = Y.T^U - \delta$) only if time-stamp $Y.T^U$ is not the initial value. However, the server has to check whether there are indirect conflicts or not, because time-satmp $Y.T^U$ denotes the minimum time-stamp among the time-stamps of the committed transactions that are conflicts directly with transaction Y. Hence, after setting $Y.T^R$, the server checks whether $Y.T^R$ is always larger than time-stamp $D_i.T$ for each data D_i in set $Y.S^W$ (Indirect conflicts check). If time-stamp $Y.T^R$ does not satisfies the rule, transaction Y is aborted. Otherwise, transaction Y is committed with time-stamp $Y.T^R$. In commit processing, the server reflects updated data in set $Y.S^W$ to the database, and sets time-stamp $D_j.T$ to $Y.T^R$ for each data D_j in set $Y.S^R$ only if time-stamp $D_j.T$ is less than $Y.T^R$. After transaction Y is committed, the server broadcasts an invalidation message to each client. The message contains re-ordering time-stamp $Y.T^R$ and set $Y.S^W$.

In our scheme, when transaction X requests a commit, the transaction is committed without validation only

```
/* When transaction Y requests a commit */
/* Validation Phase */
If (Y.T^U ≠ the initial value)
{       Y.T^C = Y.T^U - δ;
        For (each data D_i ∈ Y.S^W)
        If (D_i.T ≥ Y.T^C) Abort transaction Y;
}
Else assign time-stamp Y.T^C

/* Commit Processing */
For (each data D_j ∈ Y.S^W) Updates data D_j;
For (each data D_k (Y.S^R)
        If (D_k.T < Y.T^C) D_k.T = Y.T_C;

/* Broadcast an invalidation message */
Broadcast an invalidation message containing Y.T^C and Y.S^W;
```

(Figure 5) Validation algorithm when a transaction requests a commit

if time-stamp $X.T^U$ is the initial value, because it means that transaction X did not access any data items modified by other committed transactions in its execution phase. However, serialization can be broken when the server and the client of transaction X send messages to each other at the same time. This case can be treated by several ways. For example, the server can use a *Broadcasting Queue(BQ)* and a *threshold* which is slightly larger than the largest message delay in the system. At any moment t, the *BQ* maintains messages broadcasted in the time interval [*t - threshold, t*]. When the server gets a commit request message from transaction X, it checks the *BQ* and changes the time-stamp $X.T^U$ based on the algorithm in Figure 3 if the *BQ* has the data that have been read by transaction *X*. If the *BQ* does not have any data that have been read by transaction *X*, transaction *X* is committed without validation. Otherwise, transaction *X* is treated as a re-orderable transaction.

An alternative way is to be use a two-phase protocol. Before sending a commit request message, the client of transaction *X* sends a commit pre-request message to the server. After that, the client can send a commit request message to the server only after it receives an acknowledgement of the commit pre-request message. When the server gets a commit pre-request message, it stops broadcasting and waits until it gets the commit request message from the client. Hence, we will not deal with this kind of problem any more, because it is rare and can be treated. Thus, when transaction *X* requests a commit, the transaction is committed without validation if time-stamp $X.T^U$ is the initial value. In Figure 5, we summarize the check algorithm to be used when the server gets a commit request.

3. Extension to Enhance the Performance

The CBI protocol can make spurious aborts even if such aborts are rare. Consider the following example. Suppose that transaction *X* reads data items D_1 and writes D_2, transaction *Y* reads data items D_2 and D_1 and writes D_3, i.e., $R_X(D_1)$, $R_Y(D_2)$, $W_X(D_2)$, C_X, $R_Y(D_1)$, $W_Y(D_3)$, C_Y. If transaction *X* is committed at time ts_5, transaction *Y* is aborted when transaction *Y* reads data D_1. The reason is that transaction *Y* can not find any re-orderable time-stamp, because the value of time-stamp $Y.T^L(=ts_5)$ and that of $Y.T^U(=ts_5)$ are the same. This kind of unnecessary aborts can happen when transaction *Y* has read the data updated by transaction *X* before transaction *X* is committed and it reads again the other data read by transaction *X* after transaction *X* is committed. (we call it *unnecessary aborts due to cross read*) In this section, we extend our algorithm to eliminate such unnecessary aborts and the assumption that all write operations are update operations. In addition, we use *Thomas' Write Rule* to improve the performance.

In the extension, called CBI/2, we maintain read time-stamp $D^R.T$ and write time-stamp $D^W.T$ for each persistent data *D*. The time-stamps $D^R.T$ and $D^W.T$ are the time-

stamp of the youngest committed trans- action which read and wrote the data, respectively. A transaction views each data item as a (D_i, $D_i^W.T$) pair. When transaction X is ready to enter its commit phase, the server assigns a unique committing time-stamp $X.T^C$ to the transaction for certification. After transaction X is committed, the server reflects updated data items in set $X.S^W$ to the database, sets time- stamp $D_i^R.T$ to $X.T^C$ for each data D_i in set $X.S^R$, and sets time-stamp $D_j^W.T$ to $X.T^C$ for each data D_j in set $X.S^W$. After that, the server broadcasts an invalidation message to each client that contains $X.T^C$, $X.S^R$ and $X.S^W$.

In the extension, time-stamp T^L is updated by the same rule(See Figure 2). When the client of transaction Y gets the invalidation message, the first rule for updating T^U is replaced with the following rule.

- For each data D_i in set $X.S^R$ and $X.S^W$, if transaction Y has read and written data D_i, the client aborts the transaction Y immediately. It means that transaction Y can not be re-ordered, because transaction X has updated data D_i and committed.

When transaction Y requests a commit, if time-stamp $Y.T^U$ is not the initial value, the server checks whether $Y.T^R$ is always larger than time-stamp $D_i^R.T$ for each data D_i in set $Y.S^W$ (*indirect complete check*). If time-stamp $Y.T^R$ does not satisfy the rule, transaction Y is aborted. Otherwise, transaction Y is committed with time-stamp $Y.T^R$. In the commit phase, for each data D_j in set $Y.S^W$, the server sets time-stamp $D_j^W.T$ to $Y.T^R$ and reflects updated data to the database only if time-stamp $D_j^W.T$ is less than $Y.T^C$. Otherwise, reflection to the database is ignored(*Thomas' Write Rule*). In addition, for each data D_k in set $Y.S^R$, time-stamp $D_k^R.T$ is set to $Y.T^R$ if time-stamp $D_k^R.T$ is less than $Y.T^R$. After transaction Y is committed, the server broadcasts an invalidation message to each client. The message contains re-ordering time-stamp $Y.T^R$, $Y.S^R$ and set $Y.S^W$. We omit the extended algorithm because of page limitation.

Consider again the above example. After transaction X is validated at time ts_5, the server sets $D_1^R.T$ and $D_2^W.T$ to ts_5 and broadcasts a message containing $X.T^C$, $X.S^R$ and $X.S^W$. After Client 2 gets the message, transaction Y is not aborted when transaction Y reads data D_1, because the value of time-stamp $Y.T^L(=ts_1)$ and that of $Y.T^U(=ts_5)$ are different (See Table 7). Hence, transaction Y can be committed with re-ordering time-stamp ts_4.

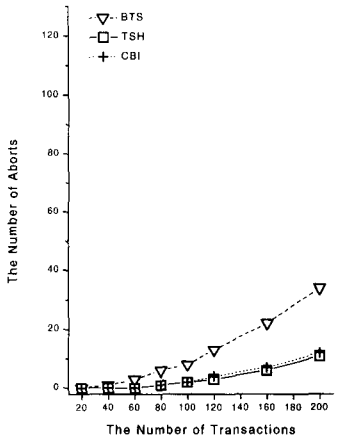
To prove that CBI is correct, we have to prove that all histories representing executions that could be produced by it are serializable. Although we don't prove formally the correctness of our protocol, we describe an idea based on serialization graph. The idea is to associate time-stamps with transactions in such a manner that each edge in the serialization graph is from a transaction with a smaller time-stamp to a transaction with a larger time-stamp. The assignment of a transaction time-stamp in the valid interval (i.e., between the lower bound and the upper bound) of all read data items to a committing transaction guarantees that each edge from the committing transaction is to a transaction with a larger time-stamp. Requiring the transaction time-stamp to be larger than the read time-stamp of updated data implies that no transaction with a larger time-stamp has an edge into the committing transaction. This ensures that the serialization graph is acyclic. By the *Serializability Theorem* in [5], all histories produced by CBI are serializable.

4. Simulation Study

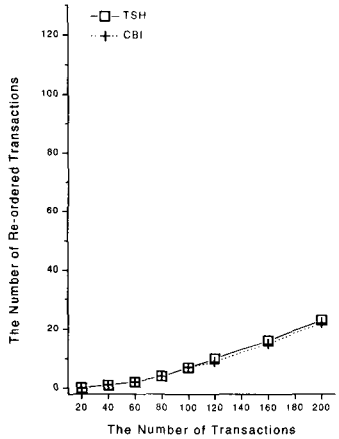
In this section, we compare the proposed schemes with Basic Time-stamp Certification(BTS) and TSH. Table 8 shows the relevant system parameters, their meanings and values. All the simulations were done

(Table 7) Parameters

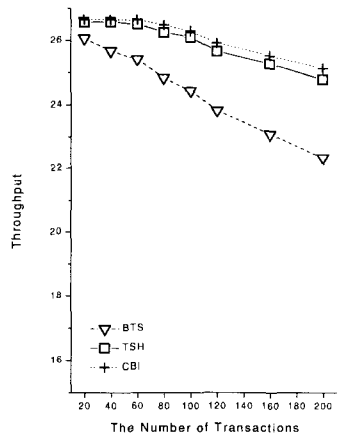
Parameter	Meaning	Values
Data_Size	Number of data items	1000, 5000
Tr_Size	Mean size of transaction	10
Max_Tr	Size of largest of transaction	14
Min_Tr	Size of smallest transaction	6
Write_Prob	Probability(Write X | Read X)	20, 40, 60, 80, 100%
Ex_IT	Mean time between operations.	200
Ex_IO	Mean time between transactions	200
Restart_Delay	Transaction restart delay	100
No_Tr	Number of transactions	20, 40, 60, 80, 100, 120, 160, 200
Time_Read	Read operation delay	30
Time_Write	Write operation delay	30
Time_Netdelay	Network delay	100



(Figure 7)



(Figure 8)

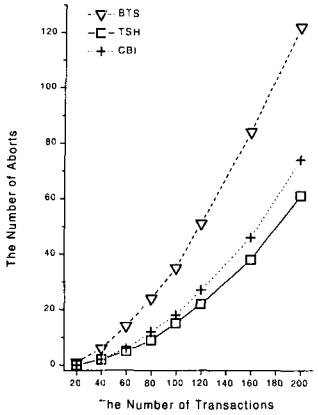


(Figure 9)

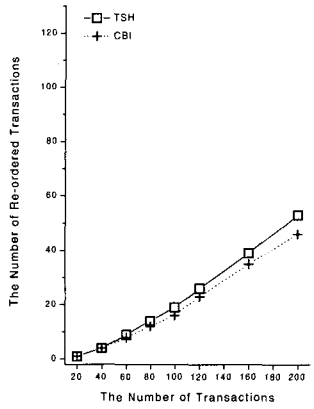
using a queuing system model. Our simulation assumes that all write operations are update operations. We use 6 time-stamps for TSH(1 for read time-stamp, 5 for write time-stamp history) following the simulation study in [3]. In this abstraction, we do not describe our simulation and its parameters in detail, because of page limitation.

The first experiment examined the throughput of three schemes for a variety of the number of trans-actions, assuming low-to-moderate contention (*Data_Size* is 5000). The figure 7 shows the number of aborts when Write_ Prob is 20%. With the given setting, the number of aborts increases in all schemes as the number of transaction increases, because the conflict probability is increased. For BTS, the increased the number of aborts manifests itself, since it can not re-order conflicting transactions. The number of aborts of TSH and that of CBI are the same at 100 trans- actions and below. For more than 100 transactions, the number of aborts of CBI is slightly higher than that of TSH in our simulation results, because of spurious aborts due to cross read.

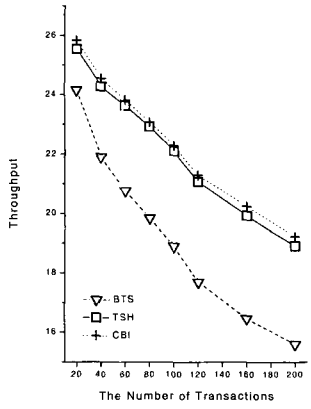
The Figure 8 shows the number of re-ordered trans-actions. In both schemes, the number of re- ordered transactions is increased as the number of transactions



(Figure 10)



(Figure 11)



(Figure 12)

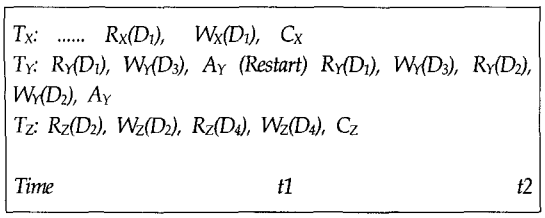
is increased, because the read-write conflict probability is increased. For any number of transactions, the difference between the number of re-ordered transactions in CBI and that in TSH(Figure 8) and the difference between the number of aborted transactions in CBI and that in TSH(Figure 7) are the same.

The Figure 9 shows the throughput of all schemes. Our scheme outperforms not only BTS and but also TSH. Only Figure 7 and 8 considered, the performance of CBI seems to be lower than that of TSH. However, the benefit of CBI scheme is not represented in Figure 7 and 8. Compared with two schemes, our scheme has a mechanism to reduce unnecessary operations due to write-write conflicts. As the conflict probability increases, the benefit of the reduction is also increased. In addition, our scheme reduces validation time compared with TSH. In CBI, a transaction is committed without validation if its time-stamp T^U is the initial value. In addition, to re-order a transaction, the server accesses $|S^W|$ data. By these reasons, the gap between TSH and CBI is increased as the number of transactions increases.

The next experiment examines the throughput of three schemes assuming high-to-moderate contention. For the experiment, we set Data_Size to 1000. While OCC schemes are not suitable in high contention environments, we use this setting to gain a better understanding of three schemes. Figure 10 shows the number of aborts with a variety of the number of transactions (Write_Prob is 20%). The number of aborts in three schemes increases rapidly as the number of transactions increases, because the conflict probability is higher than that of the first experiment(Figure 7). In addition, the difference between the number of aborts in CBI and that in TSH is increased compared with Figure 7, because the number of unnecessary aborts is increased.

The Figure 11 shows the number of re-ordered transactions in two schemes. The difference between the number of re-ordered transactions in CBI and that of TSH is also increased compared with Figure 8 by the same reason. However, the gap between the number of re-ordered transactions in CBI and that in TSH(Figure 11) becomes different from the gap between the number of aborted transactions in CBI and that in TSH(Figure 10). The reason is that CBI may cause re-aborted transactions.

Consider the following example; suppose that transactions X, Y and Z perform read and write operations as in Figure 13. In this example, transaction Y is aborted during its execution phase by transaction X, because it updates data D_1 that was updated by trans-



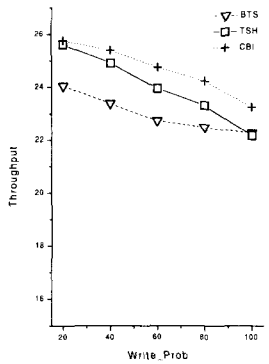
(Figure 13) Time Line of Operations of Transactions X, Y and Z

action X. After transaction Y is re-started, transaction *Y* is also re-aborted at time t_2 by transaction *Z* since it updated the data D_2. This kind of abort can happen in CBI and BOCC due to their mechanism to abort the transactions during their execution phase. However, transaction *Y* can be aborted once in BTS and TSH, because abort always occurs at the end of the transaction. However, this kind of re-abort is rare and only happens in high conflicting situation.

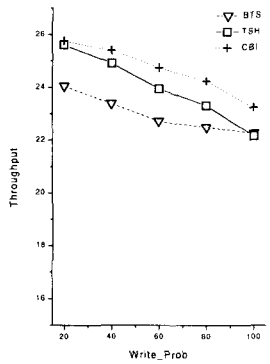
In Figure 12, the throughput of all schemes is decreased dramatically as the number of transactions increases. However, our scheme also outperforms not only BTS and but also TSH. As the figure shows, the gap between TSH and CBI is not increased even if the number of transactions increases. The reason is that the benefit of reducing unnecessary operations and validation time loses by the effect of unnecessary aborts and re-aborts.

The third experiment examines the throughput of three schemes varying the write probability from 20% to 100%. The conflict probability is increased as the write probability increases. In addition, the write-write conflict probability is also increase, because we force that all write operations is update operations in our simulation. Hence, this experiment shows the benefit of aborting the write-write conflicting transactions clearly.

Figure 14 shows the result of the experiment when *Data_Size* is 5000 and *No_Tr* is 100. The throughput in all schemes is decreased as the write probability increases. However, compared with other schemes,



(Figure 14)



(Figure 15)

the throughput of TSH is decreased rapidly, because the number of re-ordered transactions is decreased since write-write conflicts are increased. At 100% write probability, all conflicts are write-write conflicts. Hence, the performance of TSH is less than that of BTS, because the validation overhead of TSH is higher than that of BTS. The gap between the throughput of CBI and TSH is increased as the write probability increases. The reason is that the benefit of aborting the writing-writing transactions is increased even if the number of re-ordered transactions is decreased. Hence, the gap at write probability 100% shows the whole benefit of reducing unnecessary operations and validation time.

The last experiment examines the throughput of CBI/2 and that of CBI. We use the same values of parameters of the first experiment. In this experiment, the number of re-ordered transactions of TSH and that of CBI/2 are the same. Also, the number of aborts of TSH and that of CBI/2 are the same (not shown). It means that CBI /2 can reduce unnecessary aborts due to cross read. However, in high contention environments, the number of aborts of CBI/2 and that of TSH may differ, because CBI/2 can make unnecessary aborts due to re-aborts.

Figure 15 shows the throughput result of the experiment. To show the result clearly, we magnate

the Y-axis 3 times. Even if the storage for time-stamp of CBI/2 is increased 2 times, its throughput is the same as that of CBI at 100 transactions and below. For more than 100 transactions, the throughput of CBI/2 is little higher than that of CBI because it reduces unnecessary aborts due to cross read.

5. Conclusion

The performance of OCC is very sensitive to the transaction abort rate. Even if the abort probability can be reduced by re-ordering, most of re-ordering schemes have space overhead in maintaining a graph or histories. Already mentioned in [12], space overhead for time-stamp can be significant for small object. In this paper, we proposed an efficient re-ordering scheme based on a broadcast mechanism. Our scheme, called CBI, can reduce the abort probability without space overhead in maintaining a graph or histories.

In our simulation study, we showed that CBI reduced the abort rate and unnecessary operations. Additionally, we showed that CBI outperforms not only BTS but also TSH with low space overhead. In CBI, there are spurious aborts due to cross read. Hence, we also proposed an alternative scheme called CBI/2 to improve the performance. However, through several simulation experiments, we find out that CBI/2 does not outperform CBI significantly even if storage overhead is increased twice. However, CBI/2 is a useful scheme, because it dose not restrict operation type.

In our schemes, information for re-ordering is distributed among clients based on broadcast invalidation. Hence, the server maintains only one(CBI) or two (CBI/2) time-stamps for indirect conflict check. In addition, our schemes can abort a transaction during the execution phase when it accesses a write-write conflicting data item. In our scheme, each client maintains additional information. However, the size of set S^I is the same as that of set S^R in the worst case. In addition, maintaining S^I is optional. Hence, overhead due to additional information is not significant.

Another issue of re-ordering scheme is time overhead. SGT has overhead in maintaining the serialization graph and checking for cycle. In TSH, when a transaction requests a commit, the server has to access $|S^R| + (|S^W| \times 2)$ data items for re-ordering. In our scheme, transactions are committed without validation if their upper bound time-stamp are the initial value. Furthermore, the server has to access |SW| data items for re-ordering.

References

[1] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control," ACM Trans. Database Syst., vol. 6, no. 2, pp. 213~226, June 1981.

[2] J. T. Robinson, "Experiments with Transaction Processing on Multiprocessor," IBM Res. Rep. RC9725, Yorktown heights, NY, Dec. 1982.

[3] P. S. Yu, H. Heiss, and D. M. Dias, "Modeling and Analysis of a Time-Stamp History Based Certification Protocol for Concurrency Control", IEEE Trans. Know. and Data Eng., vol. 3. no. 4, pp. 525~537, Dec. 1991.

[4] R. J. T. Morris and W. S. Wong, "Performance Analysis of Locking and Optimistic Concurrecy Control Algorithms," Perform. Eval., vol. 5, pp. 105~118, 1985.

[5] P. A. Berstein, V. Hadzilacos, and N. Goodman, Concurrency Control and Recovery in Database Systems, Reading, MA: Addison-Wesley, 1987.

[6] S. H. Cho, K. Y. Bae and C. Hwang, "Write Notification for Certification Protocol Based on Time-Stamp History," 9th International Workshop on DEXA, pp. 919~924, Aug. 1998.

[7] M. Frans Kaashoek and A. S. Tanenbaum, "Group Communication in the Amoeba Distributed Operating Systems," Proceedings of the 11th ICDCS, PP. 222~230, 1991.

[8] K. P. Birman and R. van Renesse, "Reliable Distributed Computing with the ISIS Toolkit," IEEE Press, 1994.

[9] Y. Amir, D. Dolev, S. Kramer, and D. Malki, "Transis: a communication sub-system for high availability," Proceedings of the 22th International Symposium on Fault-Tolerant Computing, pp. 76~84, 1992.

[10] L. E. Moser, Y. Amir, P. M. Mellar-Smith and D. A. Agarwal, "Extended virtual synchrony," Proceedings of the 14th ICDCS, pp. 56~65, 1994.

[11] L. E. Moser, P. M. Mellar-Smith, D. A. Agarwal, R. K. Budhia and C. A. Lingley-Papadopoulos, "Totem: A Fault-Tolerant Multicast Group Communication Sytem," Communication of the ACM, 39(4), pp. 54~63, 1996.

[12] A. Adya, R. Gruber, B. Liskov and U. Maheshwari, "Efficient Optimistic Concurrency Control Using Loosely Synchronized Clock," ACM SIGMOD, 1995.

[13] A. S. Tanenbaum and M. V. Steen, "Distributed Systems Principles and Paradigms," Prentice Hall Press, 2002.

◐ 저 자 소 개 ◑

조 성 호

1994년 한국외국어대학교 컴퓨터공학과(공학사)
1997년 고려대학교 전산과학과(이학석사)
2000년 고려대학교 컴퓨터학과(이학박사)
2000년~2001년 : (주)MPSCOM 기술개발 이사
2001년~2002년 천안대학교 정보통신학부 교수
2002년~현재 : 한신대학교 정보통신학부 교수
관심분야 : 분산 시스템, e러닝 및 웹 시스템 개발, 이동 컴퓨팅
E-mail : zoch@hanshin.ac.kr

강 민 구

1986년 연세대학교 전자공학과(공학사)
1989년 연세대학교 전자공학과(공학석사)
1994년 연세대학교 전자공학과(공학박사)
1985년~1987년 삼성전자 연구원
1997년~1998년 일본 오사카 대학 객원연구원(Post Doc.)
1994년~2000년 호남대학교 정보통신공학부 조교수
2000년~현재 : 한신대학교 정보통신학과 부교수
관심분야 : 이동통신시스템, 무선인터넷 응용
E-mail : kangmg@hanshin.ac.kr