
Resolving Executing–Committing Conflicts in Distributed Real-time Database Systems

KAM-YIU LAM¹, CHUNG-LEUNG PANG¹, SANG H. SON²
AND JIANNONG CAO³

¹*Department of Computer Science, City University of Hong Kong, 83 Tat Chee Avenue, Kowloon, Hong Kong*

²*Department of Computer Science, University of Virginia, Charlottesville, VA 22903, USA*

³*Department of Computing, Hong Kong Polytechnic University, Hung Hom, Kowloon, Hong Kong*
Email: cskylam@cityu.edu.hk

In a distributed real-time database system (DRTDBS), a commit protocol is required to ensure transaction failure atomicity. If data conflicts occur between executing and committing transactions, the performance of the system may be greatly affected. In this paper, we propose a new protocol, called deadline-driven conflict resolution (DDCR), which integrates concurrency control and transaction commitment management for resolving executing and committing data conflicts amongst firm real-time transactions. With the DDCR, a higher degree of concurrency can be achieved, as many data conflicts of such kind can be alleviated, and executing transactions can access data items which are being held by committing transactions in conflicting modes. Also, the impact of temporary failures which occurred during the commitment of a transaction on other transactions, and the dependencies created due to sharing of data items is much reduced by reversing the dependencies between the transactions. A simulation model has been developed and extensive simulation experiments have been performed to compare the performance of the DDCR with other protocols such as the Opt [1], the Healthy-Opt [2], and the base protocol, which use priority inheritance and blocking to resolve the data conflicts. The simulation results show that the DDCR can significantly improve the system performance under different workload and workload distributions. Its performance is consistently better than the base protocol and the Opt protocols in both main-memory resident and disk-resident DRTDBS.

Received January 28, 1999; revised October 12, 1999

1. INTRODUCTION

In a *distributed real-time database system* (DRTDBS), transactions are associated with deadlines. Meeting the deadlines is one of the most important performance objectives. The lifetime of a transaction is divided into two stages: the *execution stage* and the *commitment stage*. In the execution stage, the operations of a transaction are processed at different sites of the system, while in the commitment stage, a commit protocol is executed to ensure failure atomicity [3]. The transactions in the execution stage are called *executing transactions* and the transactions in the commitment stage are called *committing transactions*.

There are several important factors contributing to the difficulty in meeting the transaction deadlines in a DRTDBS. One of the most important factors is data conflicts amongst transactions [4]. One kind of conflict occurs among executing transactions, referred to as *executing–executing conflicts*. Most of the proposed *real-time concurrency control protocols* (RT-CCPs) for DRTDBSs [5, 6, 7, 8] focus on resolving this kind of conflict. Another kind of

conflict involves committing transactions. When a commit protocol works with a concurrency control protocol, data conflicts among executing and committing transactions may occur. Traditionally, they are resolved by blocking, e.g. the lock-requesting transaction will be blocked until the lock-holding transaction releases the lock. However, blocking transactions may seriously affect the performance of a DRTDBS, especially when failures occur during the commitment phase. Due to the delay caused by the failures, the blocked transactions may have a high probability of missing their deadlines. Nevertheless, existing real-time concurrency control protocols are designed to address the executing–executing conflicts and they are not suitable for resolving the data conflicts involving committing transactions.

This paper studies the problems of data conflicts between executing and committing transactions in a DRTDBS, hereafter referred to as *executing–committing conflicts*. It is assumed that the transactions are firm real-time. Although missing the deadline of a firm real-time transaction is not catastrophic, it may seriously affect the

usefulness of completing the transaction. Data conflicts between committing and executing transactions are not uncommon compared with data conflicts between executing transactions. For example, in the two-phase locking protocol (2PL) [3], if an executing transaction requests a data item which is being locked by another transaction in a conflicting mode, the lock request will be denied and the executing transaction will be blocked until the lock is released. According to the strict two-phase locking principle [3], the lock on a data item cannot be released until a transaction completes the commitment stage. Because of this, a committing transaction usually holds more locks than an executing transaction, leading to lock conflicts with executing transactions. There are no data conflicts among committing transactions since committing transactions do not make any further data request.

An executing transaction can be blocked by a committing transaction for a long time due to a long delay in completing the commit procedure. As an example, let us look at the two-phase commit protocol (2PC) [9], which is one of the most commonly used commit protocols for traditional distributed database systems [3, 9], and was also widely used in the studies of DRTDBSs [1, 5, 6, 10, 11, 12]. In the 2PC, the processes of a transaction at different sites are divided into two groups. One of the processes is the coordinator and the others are the participants [3]. The following factors can cause a long delay in the execution of the 2PC:

- *Uneven distribution of transactions over the sites in the system and variable local CPU processing power.* The response time of the participants will be very different if the distribution of the transactions and the priorities of the transactions in the system are not even. The time required to reach the final commit decision in the 2PC depends on the longest response time of the participants as the final commit decision can only be made after the coordinator has received all the responses from its participants.
- *Unpredictable communication delays.* Since the 2PC requires at least two rounds of message communications between the coordinator and the participants, its performance is highly dependent on the performance of the underlying network. Even with the support of a real-time network, the communication delays are still unpredictable due to loss of messages or failures of communication links.
- *Failure of coordinator and participants.* Different kinds of failures may occur in the coordinator and in the participants during the execution of the commit protocol. Although the 2PC is resilient to these failures [3], the resolution methods are usually based on time-out. However, it is not easy to determine a suitable time-out period for resolving the failures. A well-chosen time-out interval is important to the performance of a real-time system. Otherwise, an executing transaction, which is blocked by a committing transaction, can be blocked for a very long time before the system detects the failure.

The above factors not only affect the performance of the transactions in the execution stage, but also they have a serious impact on the performance of committing transactions because these transactions are close to their completions and some of their participants might be committing. The blocked executing transactions, which have data conflicts with committing transactions, may also be affected. They can be blocked for a long time. As a result, more data conflicts may be induced as the blocked transactions are holding some locks. Consequently, it may result in thrashing due to data contention [3]. The importance of data conflicts involving committing transactions on the performance of a DRTDBS has also been reported by other studies [2].

The potential long delay in transaction commitment can make the executing–committing conflict an important problem for the performance of a DRTDBS. However, to our best knowledge, the executing–committing conflict problem has not received adequate attention in the past. Proposals started to appear in the literature only recently. Although several RT-CCPs [5, 7, 13, 14, 15, 16] have been proposed, they are mainly for solving the executing–executing conflicts. The previous studies are [1, 2, 17, 18]. In [17] and [18], compensation approaches are suggested in which the participants of a transaction are allowed to commit unilaterally, with an attempt to reduce the dependencies amongst the participants of a transaction and to improve the transaction response time. If it is found that the participants' decisions are inconsistent, a compensation transaction will be invoked to rectify the problem. The major deficiency of these approaches is that the durability property of the transaction is violated [9]. This can be a serious problem for many real-time database applications because many 'actions' are irreversible (e.g. firing a missile).

In [1], an optimistic protocol is proposed to improve system concurrency. However, the protocol creates the dependency problem. If a transaction depends on other transactions, it is not allowed to commit and has to be blocked until the transactions on which it depends have committed. A blocked committing transaction may induce a chain of dependencies as other executing transactions may have data conflicts with it. Enhancements have been made to the protocol to solve the dependency problem [2]. Two variants, called Shadow-Opt and Healthy-Opt, are proposed to reduce the probability of having a harmful effect from the dependencies. Even with these two variants it still does not solve the problem completely. The committing transactions may still have a high probability of being blocked until they miss their deadlines if their deadlines are closer than their dependent transactions, or their dependent transactions have encountered failures in their commitment. More detailed discussions on this related work can be found in Section 3.

Owing to the unique characteristics of committing transactions and the commit procedure, new approaches and protocols are required to resolve the executing–committing conflicts. In this paper we propose a new protocol, called *deadline driven conflict-resolution* (DDCR), which integrates concurrency control and transaction commitment

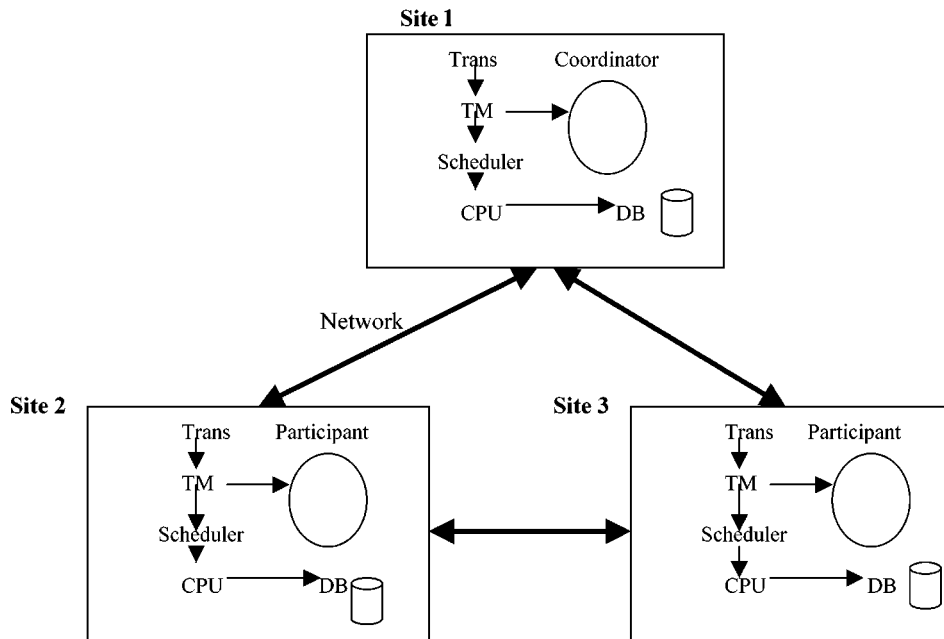


FIGURE 1. High-level architecture of the system.

in resolving executing–committing conflicts, while at the same time maintaining the schedules to be serializable. The design of the protocol and its correctness are presented. We have conducted extensive simulation experiments to study the performance of the proposed protocol. As shown in the simulation results, the DDCR not only improves the system performance under the no-failure situation in both the disk-resident database and main-memory database, but also gives a good performance even under failure situations.

The rest of the paper is organized as follows. Section 2 describes a distributed real-time database system (DRTDBS). Section 3 discusses different approaches for resolving the executing–committing conflicts. Section 4 presents our protocol and discusses its correctness. Section 5 reports on the performance evaluation of the proposed protocol. The paper is concluded in Section 6.

2. SYSTEM MODEL

In a DRTDBS, the *global database* is partitioned into a collection of *local databases*, which are distributed over the sites in the system as shown in Figure 1. Since the performance of a DRTDBS with a sequential transaction model has been found to be similar to a parallel operations transaction model [8], in order not to complicate the transaction model we have chosen the sequential transaction model. In this model, a transaction is defined as a sequence of operations. Each transaction is assigned a deadline based on the application requirements. It is assumed that the transactions are firm real-time and have the same criticality level. A transaction with an expired deadline will be aborted immediately. Each transaction is assigned a priority based on its deadline according to the earliest deadline first (EDF)

scheduling [19]. The scheduling of the CPU is based on the transaction priorities.

The operations of a transaction are processed one by one in the execution stage. The processing of an operation requires the use of the CPU and the accesses to data items located at either the local site (the site where the transaction is generated) or at a remote site. An operation that requires accessing a data item at a remote site is referred to as a remote operation. It will be forwarded to that site for processing, where a new process, if none exists, will be created for the transaction. Processes created for a transaction can communicate with each other over a communication network. However, the processes belonging to different transactions cannot communicate with each other directly.

We assume that *high-priority two-phase locking* (H2PL) is used for resolving the executing–executing conflicts [20]. In H2PL, if a higher priority transaction requests a lock which is held by a lower priority transaction, the higher priority transaction will be granted the lock and the lower priority transaction will be restarted. In this way, the problem of priority inversion is solved. The reasons for choosing H2PL in our study instead of the optimistic concurrency control (OCC) protocols [21] are:

- (1) Lock-based concurrency control protocols are much more common than OCC protocols for distributed database systems.
- (2) The performance of a validation test, which is required in OCC protocols, in a distributed environment can be very complex and may require substantial overheads.
- (3) Most of the studies in concurrency control for DRTDBSs use lock-based protocols. The performance characteristics of lock-based protocols in DRTDBSs are clearer than the OCC protocols.

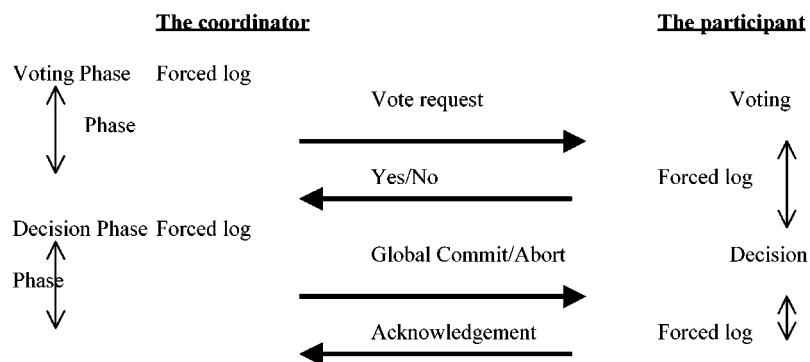


FIGURE 2. Two-phase commit protocol (2PC).

The lock scheduler at a site performs the management of locks at that site. An operation of a transaction is ready for processing only after it obtains the required locks in the appropriate (read or write) modes, otherwise it will be blocked. It is assumed that the new values from write operations are not written immediately into the database during the execution stage. Instead, a deferred update approach, which is similar to the one used in the optimistic approach [9], is used. The new values are written into a temporary workspace allocated to the transaction. Permanent database updates are performed in the commitment stage of the transaction.

When all the operations of a transaction have been processed, the transaction enters the commitment stage in which an atomic commit protocol is performed. We assume that the 2PC [9] is used because of its popularity and simplicity, as well as its wide usage in the study of DRTDBSs [1, 5, 6, 7, 8, 10]. Note that our protocol can also be applied to other blocking-based commit protocols such as pre-assumed commit and pre-assumed abort [9].

With the 2PC, the process located at the originating site of a transaction will be designated as the *coordinator*. The other processes are called *participants*. The commitment stage is divided into two phases, the *voting phase* and the *decision phase*, as shown in Figure 2.

The voting phase starts at the coordinator. After it has written a prepare record in the stable storage, the coordinator sends vote requests to the participants. A participant enters the voting phase after receiving the request from the coordinator. It responds to the vote request by replying 'Yes' or 'No' to indicate its decision for commit or abort respectively. Before it sends out the response, it also writes a forced log into the stable storage to record its decision, which will be used by the participant in recovery in case of failure. If the response is 'Yes', the participant enters the uncertainty period and waits for the final decision from the coordinator. When the coordinator has received all the responses, it enters the decision phase in which a final global commit or abort decision will be made. It sends the decision to the participants after performing a forced logging of the final decision. When a participant receives the final decision, it enters the decision phase. The decision phase is completed

when the coordinator receives all the acknowledgements of the final decision from the participants.

3. RESOLVING EXECUTING–COMMITTING CONFLICTS

3.1. The basic approaches

In case of an executing–committing conflict where the executing transaction (lock requester) is a higher priority transaction, the problem called *priority inversion with a committing transaction* occurs. Basically, there are two approaches to resolve the problem:

- (1) *restarting*: use a method similar to that of resolving the executing–executing conflicts—to restart the lower priority committing transaction; and
- (2) *blocking with higher priority* [22]: block the higher priority executing transaction and assign the committing transaction a priority higher than all other executing transactions.

The first approach is highly undesirable for the following reasons. Firstly, restarting a committing transaction wastes a lot of resources. This is because, as required by the failure atomicity property, the transaction has to be restarted from its beginning. The restarted transaction will have a high probability of missing its deadline as it has already spent a significant amount of time on processing. Secondly, if the criticality levels of the conflicting transactions are the same, it is not desirable to restart a committing transaction simply because it has a lock conflict with a transaction which has a closer deadline. The restart is beneficial only when the higher priority executing transaction is likely to miss its deadline if it is blocked, and the committing transaction has sufficient time to restart from its beginning. Otherwise, it may be *fruitless* as the higher priority transaction may later miss its deadline. Thirdly, if the committing transaction is in the decision phase, it cannot be restarted arbitrarily, because some of the participants of the committing transaction may have already committed. Restarting them may violate the durability property [9].

The second approach can solve part of the problem. Raising the priority of the committing transaction can

speed up the commit procedure without seriously affecting the scheduling as the committing transaction is close to completion. However, the blocking time of the executing transaction is still unpredictable due to the possibility of failures in the committing transaction. Blocking the executing transaction will make its fate *completely dependent* on the performance of the committing transaction. The executing transaction may be blocked until its deadline is expired. This is likely to happen if the deadline of the executing transaction is much closer than the committing transaction, or the committing transaction has encountered some failures during its commitment. Also, due to communication delay, it may take a long time to raise up the priority of the committing transaction [2]. In this case, raising the priority of the committing transaction may not effectively reduce the time required to complete the commitment stage.

3.2. The optimistic approaches

As mentioned in the related work, several optimistic protocols have been suggested to resolve the executing–committing conflicts in [1, 2]. In [1], a commit protocol called Optimistic, *Opt*, is proposed. The *Opt* protocol allows executing transactions to borrow data items being held by committing transactions. In their simulation studies, it was shown that the *Opt* protocol could significantly improve the system performance under failure-free situations due to a higher degree of concurrency. However, sharing of data items in conflicting modes, as used in the *Opt* protocol, creates dependencies among the conflicting transactions and constrains their commit orders. Due to the dependencies, if the lender aborts, the borrower may also have to be aborted and the borrower is not allowed to commit if its lender has not committed.

Basically there are two kinds of dependencies, *commit dependency* and *abort dependency* [23]. If a transaction t_1 writes a data item after another transaction t_2 , a commit dependency is created from t_1 to t_2 . t_1 is not allowed to commit until t_2 is committed. If t_1 reads an uncommitted data item written by t_2 , an abort dependency is created from t_1 to t_2 . t_1 has to commit after t_2 and the abort of t_2 will cause t_1 to abort.

The dependencies are required to maintain the ACID properties of transactions [3]. If t_1 reads an uncommitted data item from t_2 , the processing result of t_1 thus depends on t_2 . If t_2 has to be aborted later, all its effects on the database system and other transactions have to be removed as required by the failure atomicity property. Thus, t_1 also has to be aborted. However, if t_1 is allowed to commit before t_2 , it is impossible to abort t_1 and the atomicity property of t_2 cannot be ensured.

In [2], the *Opt* protocol is enhanced with an attempt to reduce the impact of the dependencies. Two variants of the *Opt*, called Shadow-*Opt* and Healthy-*Opt*, were suggested. In Healthy-*Opt*, a health factor H_t is associated with each transaction and a transaction is allowed to lend its data items to other transactions if its health

factor is greater than a threshold value. Otherwise, the lock requesting transaction will be blocked. In [2], the health factor is chosen to be the ratio of the time left for the transaction over the minimum time required for completing the commit procedure. Although the Healthy-*Opt* can reduce the number of dependencies by blocking transactions, the performance of the system will be highly dependent on the chosen threshold value. Also, while the remaining time and expected commitment time can indicate how much time a borrower can wait for its lender, they are not good indicators for determining the status of the lender.

In the Shadow-*Opt* protocol, a shadow transaction will be created whenever a transaction borrows a data item. The original incarnation of the transaction is blocked at the point of borrowing. If the lending transaction finally commits, the borrowing transaction continues its execution and the shadow transaction will be discarded. Otherwise, if the lender aborts, the borrowing transaction is aborted and the shadow transaction, which is blocking, is activated. By the creation of two versions for a transaction, it can reduce the problem of abort dependency, as the borrower does not need to restart from its beginning in case the lender has to be aborted.

In [2], a detailed simulation study was performed to compare the performance of the *Opt*, Shadow-*Opt* and Healthy-*Opt*. It has been found that the performance of the Healthy-*Opt* is better than the *Opt* and Shadow-*Opt* under a resource contention-free environment, e.g. with an infinite number of CPUs and disks. However, when the resource contention in the system is significant, the performance of the three variants is similar [2].

4. DEADLINE-DRIVEN CONFLICT RESOLUTION (DDCR)

4.1. Outline of our protocol

In this sub-section, we introduce the DDCR protocol, which consists of two main parts:

- (1) an *optimistic approach* to resolve executing–committing conflicts to increase concurrency; and
- (2) a *deadline-driven approach* to reverse the dependencies between the transactions when a transaction, which is commit dependent on a committing transaction, wants to enter its decision phase and its deadline is approaching. Reversing a dependency means that transaction t_i is originally dependent on transaction t_j , but after the reversal of the dependency, t_j is now dependent on t_i .

As we have discussed in Section 3.1, restarting a transaction to solve the data conflicts is not desirable but blocking higher priority transactions also presents problems. A better way to solve the problems is to allow transactions to share data items even in conflicting modes. The optimistic approach in the DDCR is based on this principle. In the DDCR, concurrent access to data items is allowed

by assuming that most of the conflicts will not cause the schedule to be non-serializable, and the committing transaction will complete before its conflicting executing transaction (an optimistic approach). When concurrent access of data items by different transactions in conflicting modes is allowed, dependencies amongst the conflicting transactions are defined. The final schedule is ensured to be serializable by performing a validation test before a transaction is allowed to enter its decision phase. In the validation test, the dependencies of the transaction with other transactions will be checked so that no cyclic dependency among the transactions in the decision phase will occur. Cyclic dependency may result in a non-serializable schedule.

In the 2PC, the actual commit procedure is performed in the second phase, the decision phase. Thus, we may define the ordering of transaction commitment as the ordering of the transactions entering the decision phase. Suppose transaction t_2 is commit dependent on transaction t_1 . If we make no assumption on the time required for committing a transaction, it is possible that t_2 reaches the decision phase before t_1 enters (or finishes) its decision phase. Due to the dependency, t_2 cannot enter the decision phase and has to be blocked until t_1 has finished its decision phase (either committed or aborted). If the deadline of t_2 is tighter than that of t_1 or the deadline of t_2 is very close to expiration, it is possible that t_2 may miss its deadline due to the blocking.

In the DDCR, the blocking problem due to commit dependency is resolved by letting the dependent transaction take a more ‘active’ role in resolving the data conflict if its deadline is approaching (a deadline-driven approach). An executing transaction estimates whether the committing transaction has any serious problem in its commitment. A *serious problem* is defined as a problem which can cause the abort of a transaction, e.g. the continuous loss of messages and temporary link failure or site failure. If a committing transaction t_i is determined to have a serious problem (by its dependent transaction) and the dependency is a commit dependency, the dependent transaction t_j may enter its decision phase and reverse the commit dependency by assuming that the committing transaction t_i will be aborted later. Reversing the commit dependency between transactions t_i and t_j allows t_j to proceed without waiting for t_i .

The main purpose of reversing the dependency is to save the executing transaction from missing its deadline. If the dependency is not reversed, it is likely that both transactions will miss their deadlines. The cost of reversing the dependency is: (1) the committing transaction may have to be aborted later when it is found that the schedule will be non-serializable if it is allowed to commit; and (2) the committing transaction may have to be blocked when it wants to enter its decision phase as it is now dependent on another transaction. Therefore, the dependency should be reversed only when t_i is likely to be aborted. Note that abort dependency cannot be reversed.

4.2. Resolving conflicts and dependencies in DDCR

In the DDCR, in order to allow concurrent accesses of data items in conflicting modes without affecting database consistency, three copies of the data may be temporarily created for a data item in the database. Note that this is different from the multi-version database [9] as the multiple data copies will be removed once the transaction, which has written the data item, has committed or aborted.

The three data copies are called *before-value* (BV), *after-value* (AV) and *further-value* (FV). For a data item x , BV(x) stores the latest committed value of x . If a transaction t_i has written the data item x , an after-value AV(x) will be created to store the pre-commit value of x from t_i . The value in AV(x) will be copied to BV(x) upon the successful commitment of t_i and AV(x) will be removed. FV is used to resolve the write/write conflict. If a transaction t_i wants to write on the data item x , which has been written by another transaction t_j , FV(x) will be created to store the new value from the write operation of t_i . After the successful commitment of t_j , AV(x) will be copied to BV(x) and FV(x) will be copied to AV(x). Then FV(x) will be removed.

In an executing–committing conflict, for most cases there exists only one executing transaction. If there is more than one executing transaction, it is likely that there also exists an executing–executing conflict which will be resolved according to the adopted concurrency control protocol, e.g. H2PL. Thus, defining one FV value for each data item will be sufficient for resolving most of the executing–committing conflicts.

In each site, for a transaction t_i , two dependency sets, *before-set* (BS $_i$) and *after-set* (AS $_i$), are maintained for each transaction process $t_{i,k}$ at the site k . BS $_i$ is a set of tuples containing the information about the transactions on which t_i depends. AS $_i$ is a set of tuples containing the information about the transactions which depend on t_i . Each tuple in BS $_i$ and AS $_i$ consists of three attributes: (id, mode, status), where:

- id: is the ID of the transaction which depends on t_i , or t_i depends on.
- mode: defines the type of dependency, e.g. abort or commit.
- status: defines the status (e.g. executing, in the voting phase or in the commit phase) of the transaction which depends on t_i , or t_i depends on.

Conflict resolution in the DDCR is divided into two parts: (a) resolving conflicts at the conflict time; and (b) reversing the commit dependency when a transaction, which depends on a committing transaction, wants to enter the decision phase and its deadline is approaching. The two parts are described in Sections 4.2.1 and 4.2.2 respectively.

4.2.1. Resolving conflicts at the conflict time

At the conflict time, the lock conflict is resolved as follows:

Let t_1 be a transaction with identifier id $_1$ and holding a lock on data item x , and let t_2 be a transaction with id $_2$

TABLE 1. Resolving conflicts at the conflict time.

Situation 1: t_1 is in execution stage	Situation 2: t_1 is in commitment stage		Situation 3: t_1 is restarting/aborting
Executing-executing conflict	Read-write conflict	Write-read conflict	Write-write conflict
			Blocking

requesting the same data item x . As illustrated in Table 1, three situations may happen depending on the status of t_1 .

Situation 1: t_1 is executing (an executing-executing conflict).

The conflict resolution method follows the rules set by the H2PL.

Situation 2: t_1 is committing (an executing-committing conflict).

There are three possible cases of conflict.

Case 1: Read-write conflict

If t_2 requests a write lock while t_1 is holding a read lock, then t_2 writes to $AV(x)$ and proceeds. A commit dependency is then defined from t_2 to t_1 . The transaction id of t_1 , id_1 , is added to the before-set of t_2 , BS_2 . The transaction id of t_2 , id_2 , is added to the after-set of t_1 , AS_1 . If t_2 finally commits, $AV(x)$ will be copied to $BV(x)$. Otherwise $AV(x)$ is discarded when t_2 is aborted.

Case 2: Write-write conflict

If both locks are write locks, then t_2 writes to the further-value $FV(x)$. A commit dependency is created from t_2 on t_1 . The transaction id of t_1 , id_1 , is added into BS_2 , and the transaction id of t_2 , id_2 , is added into AS_1 . Then, t_2 can continue its execution without being blocked by the data conflict. After the commit or abort of t_1 , the value in $FV(x)$ will be copied to $AV(x)$. If t_2 is aborted, $FV(x)$ from t_2 will be discarded.

Case 3: Write-read conflict

If t_2 requests a read lock while t_1 is holding a write lock, there are three alternatives to resolve the conflict.

- (1) t_2 reads from $AV(x)$ and proceeds. Since t_2 reads an uncommitted data item from t_1 , an abort dependency is created from t_2 to t_1 . id_1 is added into BS_2 , and id_2 is added into AS_1 . If t_1 aborts, t_2 also has to be aborted. Also, t_2 is not allowed to enter the decision phase before t_1 finishes the decision phase.
- (2) t_2 is blocked for a certain period of time hoping that within the period t_1 releases the lock on data item x .
- (3) t_2 reads from $BV(x)$ the committed value of the data item before t_1 writes the new value. This creates a commit dependency from t_1 to t_2 . id_2 is added into BS_1 , and id_1 is added into AS_2 .

The choice among these three alternatives depends on the commit status of t_1 and the deadline of t_2 . Alternative (1) will give a higher concurrency and a better performance if t_1 can complete its commitment within a short period of

time. However, it creates an abort dependency from t_2 to t_1 . The fate of t_2 will completely depend on t_1 . If t_1 has a serious problem in its commitment and will be aborted later, it will cause the abort of t_2 . Choosing alternative (3) creates a commit dependency from t_1 to t_2 . So, t_1 cannot enter the decision phase before t_2 until t_2 completes the decision phase. It should be chosen only if the participant of t_1 at the conflicting site is still in the voting phase, and is likely to be aborted later because of a serious problem in its commitment (which may occur in the participant at the conflicting site, in other participants, or in the coordinator). Once a transaction has entered the decision phase, it is not possible to choose alternative (3) as the dependency cannot be reversed. Alternative (2) can be used as a compromise between alternative (1) and alternative (3) if t_2 can afford to be blocked for a certain period of time. However, choosing alternative (2) will decrease the concurrency. It will only be chosen in the second part of the DDCR, when a dependent transaction wants to enter the decision phase.

Given the above analysis, the details in resolving the write-read conflict are summarized in Figure 3.

T_{thre} is a threshold value for the voting phase and is a tunable system parameter. If the time already spent by a transaction in the voting phase is longer than the threshold value, it will be assumed by its dependent transaction that a serious problem has occurred during its commitment and the transaction will be aborted later.

Situation 3: t_1 is restarting or aborting.

t_2 is blocked until t_1 releases the lock on data item x . If a transaction has to be restarted or aborted, all its locks will be released and undo operations will be performed to restore the database state to the state immediately before its execution. During this period, no other transactions are allowed to use its updated data items as the values are inconsistent. Although this will create the priority inversion problem, the blocking will not be long as it only needs to wait for the completion of the undo operations for the transaction process at the conflicting site. Also, the duration can be made shorter by giving the highest priority to the undo operations.

So only resolving the write-read conflicts using alternative (2) will create transaction blocking. In that case, if a transaction has been blocked for a long time, it will check again so as to reduce the probability of being blocked to miss the deadline. For the other cases, concurrent access of data items is allowed. The cost is that we have to define dependencies among the conflicting transactions. The dependency may create problems and constrains their

At the conflict time (t_2 is executing and t_1 is in the commitment stage):

```

if  $t_1$  is in the voting phase
  if  $t_2$  is already abort dependent to  $t_1$ 
    choose alternative (1)
    update the dependency sets of  $t_1$  and  $t_2$  at the conflicting site
  else
    calculate the time already spent by  $t_1$  in the voting phase,  $T_v$ 
    if  $T_v$  is greater than  $T_{thre}$ 
      choose alternative (3)
      ( $t_2$  assumes that  $t_1$  has a serious problem in its voting phase and
      will be aborted later.)
      update the dependency sets of  $t_1$  and  $t_2$  at the conflicting site
    else
      choose alternative (1)
      update the dependency sets of  $t_1$  and  $t_2$  at the conflicting site
    endif
  endif
else
  choose alternative (1)
  ( $t_1$  has entered the decision phase. It is not possible to reverse the dependency)
  update the dependency sets of  $t_1$  and  $t_2$  at the conflicting site
endif

```

FIGURE 3.

commit order when the executing transaction wants to commit. So, in the following section we will discuss how the dependency problem can be resolved.

4.2.2. Reversing the commit dependencies at the commitment time

When a transaction t_2 wants to enter the decision phase, its before-sets and after-sets, BS_2 , and AS_2 , will be checked. The checking is done by the coordinator of t_2 . A participant can send the information in its before-set and after-set to its coordinator at the same time as it sends the vote response. If BS_2 is not empty and the dependency is a commit dependency, t_2 may reverse the dependency so that it can proceed.

The following pseudo-codes summarize the conflict resolution procedures when a transaction t_2 wants to enter its decision phase but it is commit dependent on a committing transaction t_1 .

Before the start of the decision phase of t_2 ,

```

if  $t_1$  is in the voting phase
  if  $t_2$  is abort dependent on  $t_1$ 
    block  $t_2$  until  $t_1$  is committed or aborted
  else
    calculate the time already spent by  $t_1$  in the
    voting phase,  $T_v$ 
    if  $T_v$  is smaller than  $T_{thre}$ 
      calculate the waiting time of  $t_2$ ,  $T_w$ 
      block  $t_2$  for the time  $T_w$  (alternative 2)
    endif
    put  $id_1$  into  $AS_2$  and put  $id_2$  into  $BS_1$ 
    (reverse the commit dependency by assuming
    that  $t_1$  will be aborted later)
     $t_2$  enters the decision phase
  endif
else
  block  $t_2$  until  $t_1$  is committed or aborted
endif

```

The waiting time, T_w , of transaction t can be calculated as the following:

$$T_w = \text{deadline of } t - \text{mean time to complete } t.$$

If the committing transaction t_1 enters the decision phase before the waiting time of t_2 has expired, t_1 will inform t_2 so that t_2 can immediately proceed with alternative (1).

Note that if a transaction is abort dependent on another transaction, it cannot reverse the dependency. Also, if a transaction is already in the decision stage, it is too late to reverse the dependency with that transaction as the actual commit procedure has started. Once the dependency order is reversed, it will be recorded in the before-sets and after-sets of the affected transaction processes at the conflicting site, and in the coordinators of the transactions. If a transaction has to be aborted due to whatever reason, all its dependencies in the dependency sets of other transactions will be cleared.

4.3. Ensuring serializability

A schedule is serializable if it is *equivalent* to a serial schedule of the same set of transactions [3]. Serializability can be checked by using a serialization graph in which the serialization orders of the transactions are defined. If the serialization graph is acyclic, then the schedule is serializable. In the DDCR, the serialization orders of the committed transactions in a schedule can be determined based on their before-sets and after-sets. If there exists a cyclic dependency among the transactions by searching their before-sets or after-sets, the schedule is non-serializable.

The checking for serializability in the DDCR consists of two parts. Firstly, it is done by the participant at its own

site. Secondly, it is done by the coordinator of the transaction before the transaction is allowed to enter the decision stage. In the DDCR, once an abort dependency is defined for a transaction t_i to t_j , an opposite dependency is not allowed. If t_i is commit dependent on t_j , an opposite dependency is not allowed as it reverses the dependency. Thus, a cyclic dependency between two transactions is impossible. Note that the checking is only required at the conflicting site. If two transactions are in conflict, they must have a process at that site. Before a transaction is allowed to enter the decision phase, its participants will check the before-sets and after-sets of all the transaction processes at their sites. If it is found that there exists a cyclic dependency with other transactions, the transaction is not allowed to enter the decision and has to be aborted. In this way, local serializability can be ensured. After passing the local serializability test, the participants send their vote responses to their coordinator, attached with their serialization orders with respect to other transactions. Based on this information, the coordinator can determine the transaction's global serialization orders with other transactions. If its commitment will result in a non-serializable schedule, it will be aborted. Otherwise, it will make the commit decision accordingly. In this way, global serializability can be ensured.

Although it is possible that a transaction may have to be aborted due to cyclic dependencies with other transactions, as found by the participants and the coordinators, this probability is low. This is because t_j is committing and t_i is executing, alternative (1) (in Section 4.2.1) will be chosen to resolve the executing-committing conflicts and their dependency will be $t_j \rightarrow t_i$ in most of the cases. Thus, the serialization orders of the transactions are usually defined according to their orders of entering into the voting phase. The only exception is when the committing transaction has a problem in its commitment. In this case, the dependency may be reversed or the executing transaction may decide to choose alternative (3) (in Section 4.2.1). Since the committing transaction is likely to have a problem in its commitment, it is unlikely that it wants to reverse the dependency later.

4.4. Implementation considerations and advantages

An important issue of the DDCR is to reduce the number of blocking due to executing-committing conflicts. Of course, the implementation of the protocol will create additional workload for the system. The main overheads are in:

- (1) maintenance of the different versions of a data item if more than one transaction want to access it at the same time;
- (2) maintenance of the dependencies of the transactions as defined in the dependency sets; and
- (3) checking of the dependency sets of a transaction when it wants to enter its decision phase.

In the DDCR, most of the data items have only one version. Multiple versions will be created for a data item only when more than one transaction wants to access it at

the same time, and the multiple versions will be destroyed once the transaction writing the data item has committed or aborted. Thus, the overhead to maintain the multiple versions should be small when the probability of data conflict is low. If the probability of data conflict is high, the benefit from higher concurrency should be much greater than the cost required to maintain the multiple data versions. Furthermore, the DDCR increases the concurrency of the executing transactions, but more importantly, the protocol increases the concurrency of committing transactions. Therefore, the committing transactions can complete and release their locks earlier. This can significantly reduce the degree of data conflicts amongst the transactions and the cost for resolving data conflicts. Note that in the H2PL, a restart approach is used to resolve the problem of priority inversion. Restarting transactions may greatly increase the system workload. Lowering the degree of data conflict can significantly alleviate the system workload and improve the system performance. Thus, the proposed protocol DDCR will also be useful for an overloaded system.

Similarly, the cost for maintaining the dependency sets also depends on the number of data conflicts. The dependencies between the transactions can easily be implemented in the transaction table, in which attributes are added to indicate the dependencies of the transactions. Note that similar overhead is required in the Opt protocols [1, 2], in which it also has to maintain the dependency relationships among the transactions until a transaction is committed.

The checking of the dependencies for ensuring serializability of execution creates additional messages for communication between the processes at different sites. However, the message cost is getting lower in most distributed systems as the speed and capacity of the network is increasing at a high rate. The main concern in the performance for the checking is the time duration to complete the checking. One way to reduce the time duration is to integrate it with the 2PC. For example, the dependency sets of a transaction at a site can be sent with the vote response to the coordinator.

Comparing the DDCR with the blocking with higher priority approach, the probability of blocking due to data conflict is lower in the DDCR as concurrent access of data items is allowed by the optimistic part of the protocol. Furthermore, the probability of a blocked transaction missing its deadline will be lowered. If the blocked transaction has been blocked for a long time or its dependent transaction has been in the commitment stage for a long time, the dependency can be reversed so that the blocked transaction can proceed. Comparing the DDCR with the transaction restart approach, the number of transaction restarts in the DDCR should be much lower, as transaction restart is used only for resolving those executing-committing conflicts where the dependencies among the transactions cannot be resolved to be acyclic. The system concurrency with the DDCR is also higher than the Opt protocols as the dependencies among transactions can be reversed if the committing transaction has already spent a long time in its commit stage. Although this may cause

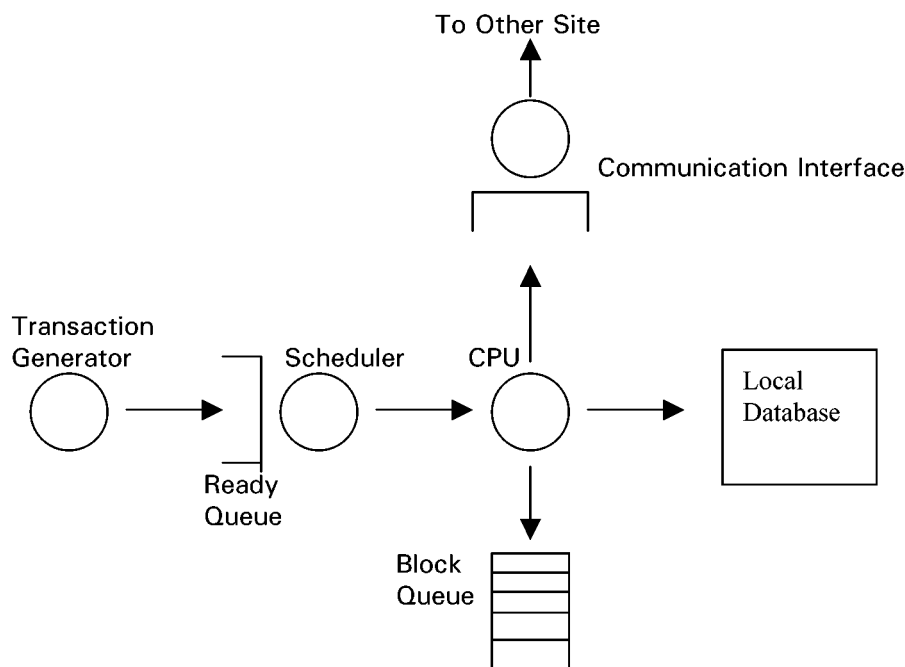


FIGURE 4. Simulation model of a site in the DRTDBS.

the committing transaction to be aborted, it will not degrade the system performance significantly since the committing transaction is likely to be aborted due to its own problems in commitment.

5. PERFORMANCE EVALUATION

5.1. Simulation model and parameters

The simulation model follows the DRTDBS described in Section 2. The model consists of a collection of sites which are fully connected by a point-to-point network. Each site is defined with a unique site ID and contains a local database system which consists of a transaction generator, a scheduler, a CPU, a ready queue, a local database, a communication interface and a block queue. We have developed two DRTDBS models. The first one is a main-memory resident DRTDBS (MDRTDBS) so as to eliminate the impact of different disk scheduling algorithms on the performance of the protocols [20]. It is illustrated in Figure 4. Since the use of main memory databases is not so common in commercial database systems, we also have developed a second model which is a disk resident DRTDBS (DDRTDBS). In the DDRTDBS model, the scheduling of the disk is prioritized and is non-preemptive.

In both models, a site connects to a network via a communication interface which is responsible for sending and receiving messages. Each communication link is modelled as a delay center. In order to eliminate the impact of different network configurations on the system performance, it is assumed that the sites are fully connected.

The transaction generator at each site is responsible for generating transactions with an arrival rate (AR) following a Poisson distribution. Transactions wait in the ready queue

for the CPU according to their priorities. The scheduler selects the transaction with the highest priority in the ready queue to use the CPU, and is responsible for the management of locks at the site. When a transaction requests a lock on a data item and the lock is being held in a conflicting mode by another transaction, several cases may occur. If the lock holding transaction is in the executing stage and has a higher priority, the lock requesting transaction will be put in the block queue until the lock is released. If the priority of the lock requesting transaction is higher, the lock holding transaction will be restarted and the lock will be released as required in the H2PL. On the other hand, if the lock holding transaction is in the committing stage, the method used for resolving the conflict depends on the adopted conflict resolution method. We have implemented four protocols for resolving the executing-committing data conflict: (1) blocking with higher priority (we call it the base protocol); (2) the Optimistic protocol (Opt) [1]; (3) the Healthy Optimistic (HOpt) [2], and (4) the DDCR. In the HOpt, the definition of the healthy factor follows the one used in [2]. It is defined as the remaining execution time over the mean time for completing the commitment stage at the time when a transaction enters its commitment stage. If the healthy factor is greater than 1, the executing transaction will be allowed to borrow the value from the committing transaction. In [2], it was found that the best range for the healthy factor is between 1 to 2 and its performance with this range is similar.

5.1.1. Transaction deadline assignment

Following the assumptions described in previous works [5, 6, 8, 12, 13, 15, 20, 24], the transaction deadlines are assumed to be proportional to their expected execution

time¹ and are defined as follows:

$$\text{Deadline} = T_{\text{ar}} + (T_{\text{stage1}} + T_{\text{stage2}}) * \text{slack factor}.$$

The slack factor (SF) is a random variable uniformly distributed between two bounds. T_{ar} is the arrival time of the transaction. T_{stage1} is the time required for the execution stage. T_{stage2} is the time required for the commitment stage.

T_{stage1} is given by:

$$T_{\text{stage1}} = \text{processing delay} + \text{propagation delay}$$

where the processing delay is the mean time required to process the operations in a transaction and the propagation delay is the communication overhead for accessing data items at remote sites.

- The processing delay is defined as:

$$\begin{aligned} \text{processing delay} &= N_{\text{oper}} * P_{\text{write}} * T_{\text{write}} \\ &+ (1 - P_{\text{write}}) * N_{\text{oper}} * T_{\text{read}} \end{aligned}$$

- The propagation delay is defined as:

$$\text{propagation delay} = N_{\text{oper}} * P_{\text{remote}} * T_{\text{com}}$$

where N_{oper} is the number of operations in a transaction, P_{write} is the probability of a write operation, P_{remote} is the probability of a remote operation, T_{write} is the mean CPU time required to process a write operation, T_{read} is the mean CPU time required to process a read operation, and T_{com} is the point-to-point communication delay.

T_{stage2} is calculated as follows:

$$T_{\text{stage2}} = T_{\text{vote_time}} + T_{\text{commit_time}}.$$

$T_{\text{vote_time}}$ is the mean time required for the voting phase. It includes the communication delay for sending the vote requests to the participants, the time to check for commitment, the time for writing the vote response into the stable storage, and the time for sending the commit responses to the coordinator. $T_{\text{commit_time}}$ is the mean time required for the decision phase. It includes the time for sending the final commit decision to the participants, the time for writing the final decision into the stable storage, the time for releasing the locks, the time for permanently updating the data items for write operations, and the time for the sending of final acknowledgements to the coordinator.

5.1.2. Modelling of commit delays

In order to model temporary failures at transaction commitment, we define two variables, T_{failure} and P_{failure} . T_{failure} is the additional time required for delivering a message to a remote site due to the occurrences of different kinds of temporary failure. The failures may be the loss of messages or breakdowns of the communication links. T_{failure} is defined to be a random variable which is normally distributed. P_{failure} defines the probability of failure and its value is Bernoulli distributed. Note that T_{failure} is not included in the calculation of transaction deadlines.

¹The time required for execution without any queueing delay, blocking or restart due to data conflict.

5.2. Performance measures

The primary performance measure used is the missing ratio (MR), which is defined as:

$$\text{MR} = \frac{\text{Number of transactions missing their deadlines}}{\text{Total number of transactions processed}}.$$

In order to understand the probability of lock conflicts, we measure the mean block queue length and conflict probability. The mean block queue length indicates the mean number of executing transactions which are blocked due to lock conflicts with other executing transactions. If the probability of lock conflict is higher, the length of the block queue will be longer. The conflict probability is defined as:

$$\text{Conflict probability} = \frac{\text{Number of lock conflicts}}{\text{Number of lock requests}}.$$

Other measures are the executing-executing (E-E) conflict proportion, the executing-committing (E-C) conflict proportion, the follow rate and reverse rate. They are important to understand the behaviour of the protocols. The E-E conflict proportion and the E-C conflict proportion are used to indicate the proportion of each type of conflict. The E-E conflict proportion is defined as:

$$\begin{aligned} \text{E-E conflict proportion} \\ &= \frac{\text{Number of executing-executing conflicts}}{\text{Total number of lock conflicts}}. \end{aligned}$$

The E-C conflict proportion is defined as:

$$\begin{aligned} \text{E-C conflict proportion} \\ &= \frac{\text{Number of executing-committing conflicts}}{\text{Total number of lock conflicts}}. \end{aligned}$$

In the DDCR, Opt and HOpt, some of the executing-committing conflicts are resolved by the creation of dependencies. The conflicts are also counted in the calculation of the E-C conflict proportion so that their values for the DDCR, Opt and HOpt, can be compared with those for the base protocol. The reverse rate is defined only for the DDCR. It is defined as:

$$\text{Reverse rate} = \frac{\text{Number of dependencies reversed}}{\text{Number of committed transactions}}.$$

The follow rate is defined only for the DDCR, Opt and HOpt. It is defined as:

$$\text{Follow rate} = \frac{\text{Number of dependencies created as a result of execution-committing conflicts}}{\text{Number of committed transactions}}.$$

5.3. Simulation experiments and results

5.3.1. Baseline setting

The baseline setting of the values for the parameters is shown in Table 2.

TABLE 2. Baseline values for the model parameters.

N_{site}	Number of sites	4
Site ID	Site IDs	0, 1, 2, 3
AR	Arrival rate	5 transactions/s
$S_{\text{data_obj}}$	Number of data items in each site	200 (data items per site)
T_{read}	Time to read a data item s	10 ms (constant) for disk database and 1 ms for memory resident database
T_{write}	Time to update a data item s	10 ms (constant) for disk database and 1 ms for memory resident database
T_{com}	Communication delay	10 ms (constant)
$P_{\text{site}(i)}$	Site i data request probability	0.25 (where $i = 0, 1, 2, 3$)
N_{oper}	Number of operations in a transaction	2–5 operations (uniform distribution)
SF	Slack factor	1–4 (uniform distribution)
P_{write}	Write operation probability	50% (Bernoulli distribution)
T_{serial}	CPU time for checking serialization order	1 ms (constant)
$T_{\text{vote_time}}$	Time for completing the voting phase	40 ms (normal distribution)
T_{failure}	Time for sending responses	250 ms (normal distribution)
P_{failure}	Probability of failure	0.3 (Bernoulli distribution)
$T_{\text{commit_time}}$	Time for completing the decision phase	40 ms (normal distribution)
T_{thre}	Threshold value for DDCR	1.2

T_{serial} is only required in the DDCR. It is the CPU time required to check the before-set and after-set of a transaction when a transaction wants to enter the decision phase or when it wants to commit.

Simulation results are obtained by defining four sites ($N_{\text{site}} = 4$) in the model with site IDs of 0, 1, 2 and 3. Each site contains 200 data items ($S_{\text{data_obj}}$). The reason for using a small database is to create a high data contention environment so that the impact of concurrency control on the system performance will be more significant. A small database also allows us to study the effect of hot spots, where a small part of the database is accessed frequently by most of the transactions. The relatively long time for transaction commitment and voting is due to the writing of transaction logs into the stable storage.

It is assumed that each operation accesses one data item. In the baseline setting, the data items required by the operations are assumed to be evenly distributed over the sites in the system. In the third set of experiments (Section 5.3.5), the performance of the protocols under a skewed workload is tested, in which the probability of references of the data items at certain site will be different from the other sites.

The simulation length is set to be 1000 s. Each run of the simulation produces the statistics of processing of about 20,000 transactions at an arrival rate of 5 transactions/s. In the experiments, using both the main memory DRTDBS model (MDRTDBS) and the disk resident DRTDBS model (DDRTDBS), we study the performance of the DDCR and compare it with the base protocol and the optimistic protocols (Opt and HOpt) under a failure-free situation. In the last set of experiments (Section 5.3.8), we compare the performance of the four protocols under temporary failure situations.

5.3.2. Determination of the value for T_{thre}

The variable T_{thre} defines the criteria for selecting the various alternatives to resolve the executing–committing conflicts in the DDCR. Its value determines the threshold time for reversing a dependency. If the time already spent by the committing transaction in the voting phase is longer than the value calculated from the following formula,

$$T_{\text{thre}} \times T_{\text{vote_time}}$$

the executing transaction will try to reverse the commit dependency. If the value for T_{thre} is set to be too large, the executing transactions will always be commit dependent on the committing transactions. In this case, if the committing transactions have any problems in the commitment stage, the executing transactions will be affected. On the contrary, if the value is set to be too small, a transaction may always be blocked by its dependent transaction from entering into its decision phase.

Figure 5 shows the impact of different values of T_{thre} on the MR. For other parameter values, the values listed in the baseline setting are used. As shown in Figure 5, the MR decreases as T_{thre} increases from 1 up to a point somewhere between 1.2 and 1.4. After that point MR rises. This is consistent with our expectation. Since the best performance is obtained when $T_{\text{thre}} \approx 1.2$, the remaining experiments on the DDCR will use this value for T_{thre} .

5.3.3. Impact of arrival rate

Figures 6 to 12 show the impact of arrival rates (workload) on the performance of the four protocols in the MDRTDBS model. As anticipated, the MRs for the protocols increase with arrival rate (see Figure 6). At heavier workload, the probability of deadline missing is higher due to a higher

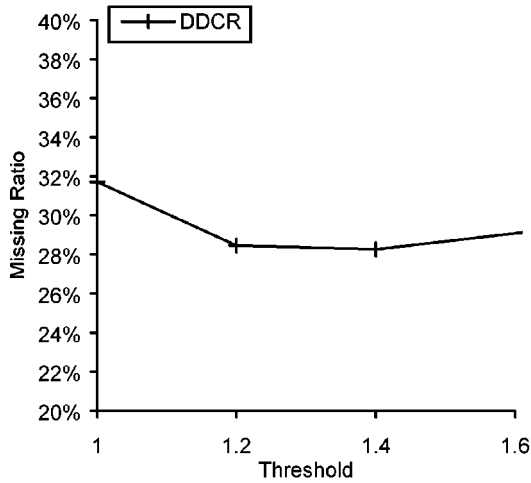


FIGURE 5. The value of T_{thre} on missing ratio (disk).

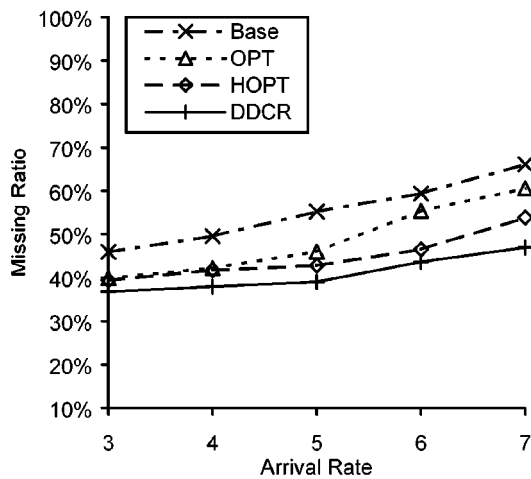


FIGURE 6. Arrival rate on missing ratio (memory).

probability of lock conflict and longer queuing delay for using the resources in the system.

As depicted in Figure 6, the DDCR and the optimistic protocols (Opt and HOpt) have much smaller MRs than the base protocol. The differences become larger at higher arrival rates, especially between the DDCR and the base protocol. At an arrival rate of 7 transactions/s, the MRs for the DDCR and Opt are 47% and 60% respectively. These are about 15% and 5% lower than the MR for the base protocol. The performance improvement achieved by the optimistic protocols over the base protocol is consistent with the results in [1, 2].

The better performance of the DDCR and optimistic protocols is due to the better approaches used for resolving the executing–committing conflicts. Blocking with priority inheritance is the only method used in the base protocol to resolve the conflicts. The blocked transactions may miss their deadlines while they are waiting for their locks. Even worse, as the blocked transactions are holding some locks, they may induce more lock conflicts. As can be observed in Figure 7, the conflict probability for the base

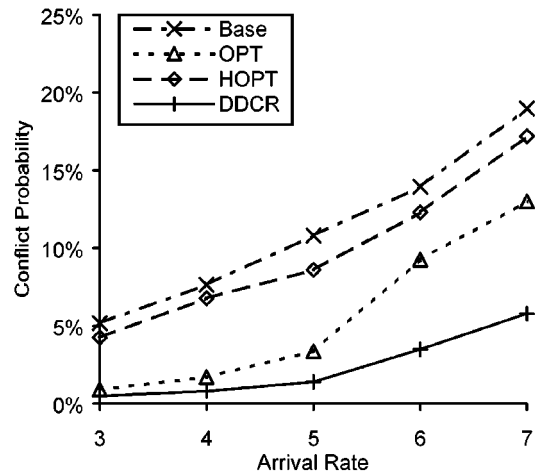


FIGURE 7. Arrival rate on conflict probability (memory).

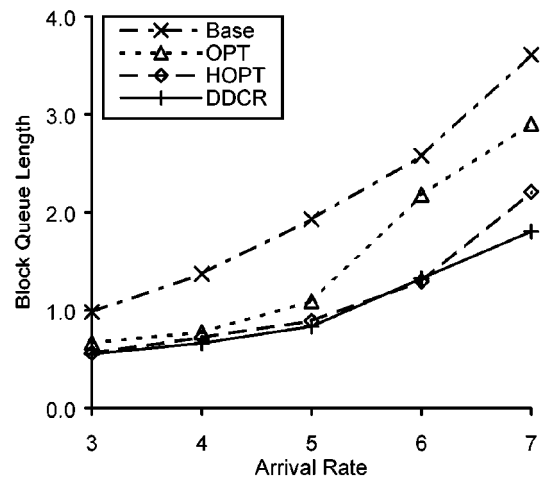


FIGURE 8. Arrival rate on mean block queue length (memory).

protocol is much higher than that for the DDCR and the optimistic protocols. At an arrival rate of 7 transactions/s, the conflict probability for the DDCR and Opt are about 14% and 6% lower compared with the base protocol. (The conflict probability for HOpt is higher than for Opt, due to its blocking nature. If the healthy factor is false, HOpt will resolve the executing–committing conflicts by blocking.) Greater lock conflict probability in the base protocol can also be observed in Figure 8, in which the mean block queue length for the base protocol is the longest.

A higher degree of concurrency can be achieved with the DDCR and the optimistic protocols as some executing–committing conflicts will not necessarily result in blocking and concurrent accesses of locks in conflicting modes are allowed. As shown in Figure 9, the follow rates for the DDCR and the optimistic protocols increase with arrival rate. This indicates that, at heavier workload, more executing–committing conflicts are resolved by the creation of dependencies among the conflicting transactions.

Figures 10 and 11 show the impact of the arrival rate on the E–E and E–C conflict proportions respectively. It

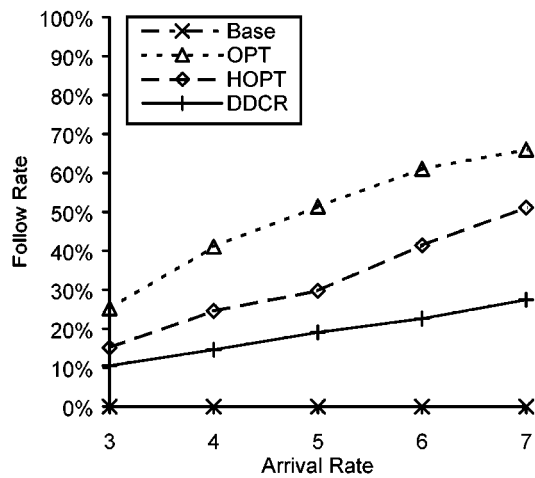


FIGURE 9. Arrival rate on follow rate (memory).

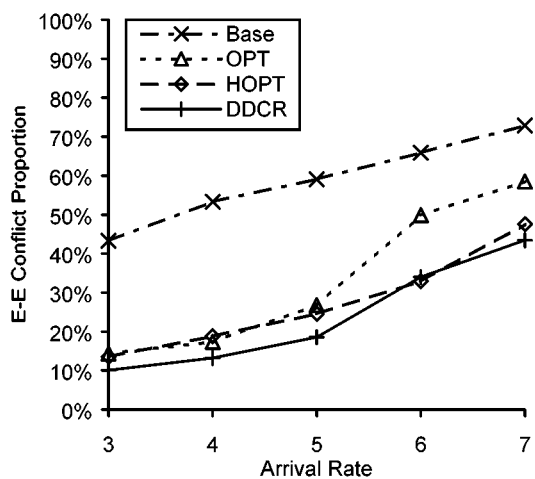


FIGURE 10. Arrival rate on E-E conflict proportion (memory).

is interesting to see that at a light workload, the E-C conflict proportion is greater than the E-E conflict proportion, while the reverse is true at heavy workload. It is because at light workload, most of the transactions can reach their commitment stage without encountering any lock conflict. Lock conflicts are more likely to happen with committing transactions which are holding more locks. At heavy workload, the probability of chain of blocking is higher. Therefore, the execution stage becomes much longer than that at light workload and the probability of lock conflict among the executing transactions is higher than with committing transactions. The observation from Figure 10 that the E-E conflict proportion for the base protocol is much higher than that for the DDCR and the optimistic protocols further supports the argument that the blocking method used in the base protocol for resolving the executing-committing transactions induces more executing-executing conflicts.

When we compare the MRs for the DDCR with that of the optimistic protocols in Figure 6, we can observe that the MR for the DDCR is consistently smaller than that for the Opt and the HOpt. In the Opt and HOpt, a transaction cannot

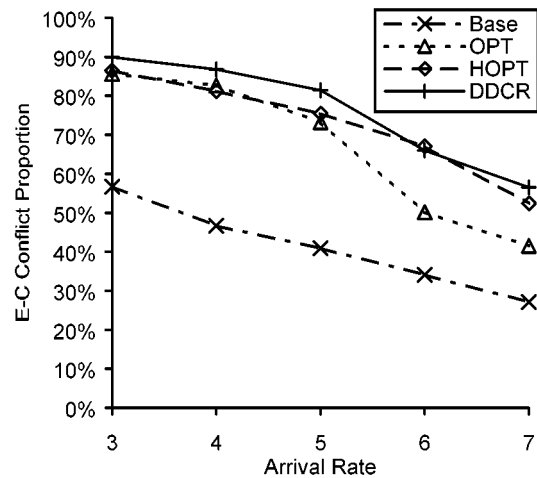


FIGURE 11. Arrival rate on E-C conflict proportion (memory).

start its decision phase if its dependent transaction has not committed (or aborted). This introduces a blocking delay in transaction commitment, where the blocked transaction may miss its deadline while it is waiting to enter into its decision phase. The greater number of dependencies created in the Opt and HOpt compared with the DDCR can be observed in Figure 9, in which the follow rate for the Opt and HOpt is much higher than the DDCR. Although the number of dependencies is smaller in the HOpt compared with the Opt, the number of blockings for resolving the executing-committing conflicts is higher. Blocking of transactions affects the concurrency of the system and makes the transactions have a higher probability of missing their deadlines.

In the DDCR, the blocking problem caused by the commit dependencies is resolved partly by reversing the dependencies if their dependent transactions have spent too much time in their voting phases. The reversal reduces the probability of blocking and the probability of formation of dependency chains among the committing transactions. These are highly desirable. This also explains why the follow rates for the Opt and HOpt are much higher than that for the DDCR as shown in Figure 9. As shown in Figure 12, the larger reverse rates at higher arrival rates indicate that more dependencies are reversed at heavier workload, leading to a smaller number of transactions being blocked due to commit dependencies. As depicted in Figure 12, about 16% of the dependencies are resolved by reversing the dependency in the DDCR at an arrival rate of 7 transactions/s. The fact that the total values of the follow rate and the reverse rate for the DDCR are smaller than the follow rate for the Opt and HOpt indicates the chains of dependency problem in the Opt and HOpt. The results also explain why the mean block queue length for the DDCR is much smaller than the Opt as shown in Figure 8.

5.3.4. The impact of locality ratio

Figure 13 shows the MRs for the DDCR, optimistic protocols and the base protocol at different locality ratios

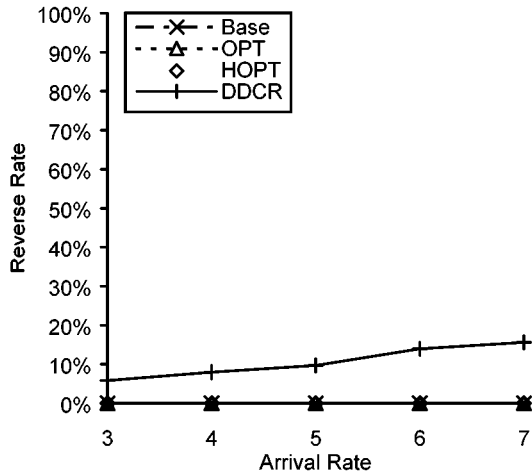


FIGURE 12. Arrival rate on reverse rate (memory).

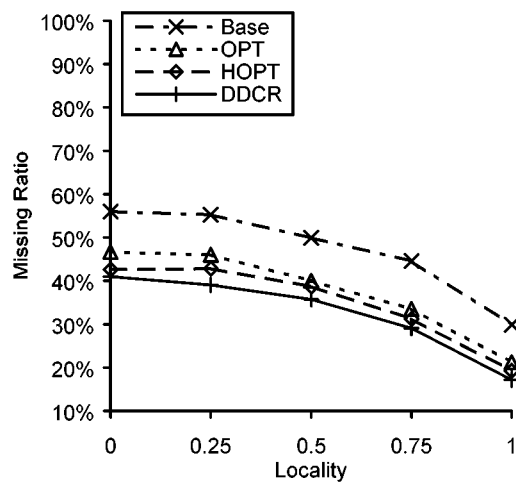


FIGURE 13. Locality on missing ratio (memory).

in the MDRTDBS model. The locality ratio defines the proportion of local operations over the total number of operations in a transaction. In the baseline setting, it is defined to be 0.25. It means that the required data items of the operations are evenly distributed among all the four database sites in the system. If the ratio is increased, more local operations will be assigned to a transaction. When the locality ratio is set to be one, a transaction will become a local transaction, i.e. all the operations of a transaction will be processed at the originating site of the transaction. Note that when the locality is varied, the expected execution time of the transactions will also be changed. However, the deadline constraints of the transactions will remain similar as the slack factor for the transactions is kept constant.

As shown in Figure 13, although the deadline constraint remains similar, the MRs for the four protocols decrease when the locality ratio is increased from 0 (all remote operations) to 1 (all local operations). This is due to smaller overheads incurred in resolving lock conflicts at higher locality ratios. Restarting a transaction with a larger number of remote operations will be more expensive than restarting

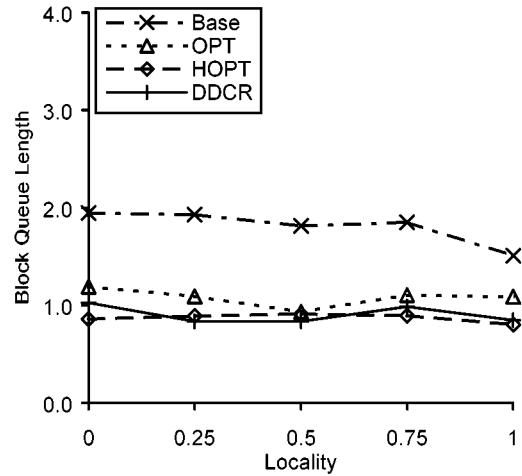


FIGURE 14. Locality on mean block queue length (memory).

a transaction with only local operations as remote operations require more resources for processing. Also, a restarted transaction will have a much higher probability of missing its deadline if it has more remote operations.

Consistent with the results in the previous set of experiments, the MRs for the DDCR and the optimistic protocols are smaller than the base protocol. As observed in Figure 13, the MRs for the DDCR and Opt are, on average, 15% and 10% lower than the base protocol respectively. Again, this is due to the lower conflict probability and higher degree of concurrency achieved by the DDCR and the optimistic protocols. The smaller mean block queue length for the DDCR and the optimistic protocols can be observed in Figure 14.

When the performance of the DDCR is compared with the optimistic protocols, the DDCR is consistently better than the optimistic protocols, although the differences in the MRs shrinks gradually as the locality ratio increases. This is because the benefits gained by reversing the commit dependencies decrease as the cost for resolving the executing-committing conflict becomes smaller. Nevertheless, even at a locality ratio of 1, the DDCR is still able to provide an improvement of about 5% better than Opt and HOpt in terms of MR as depicted in Figure 13. The performance of the Opt and HOpt is similar, and the HOpt gives a marginally better performance than the Opt.

5.3.5. The impact of skewed workload

In a DRTDBS, it is common to have transactions whose required data items may not be evenly distributed among the sites in the system. The data items at one of the sites may have a higher (or lower) probability to be accessed by the transactions than the data items located at the other sites. We call this site the primary site. The workload and lock conflict probability will be higher (or lower) at the primary site.

In this set of experiments, the performance of the four protocols is evaluated with site 0 as the primary site. The parameter $P_{\text{site}(0)}$ is used to define the proportion of

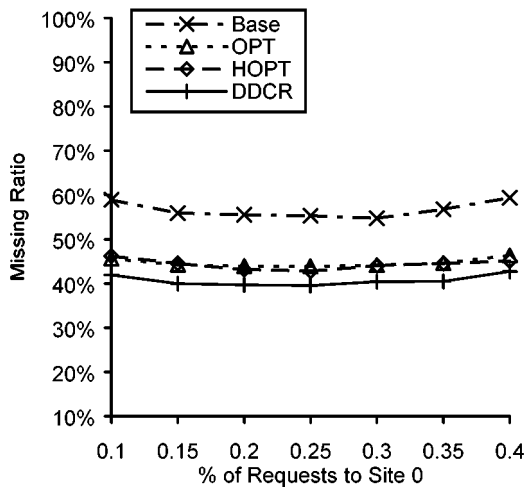


FIGURE 15. Percentage of data requests to site 0 on missing ratio (memory).

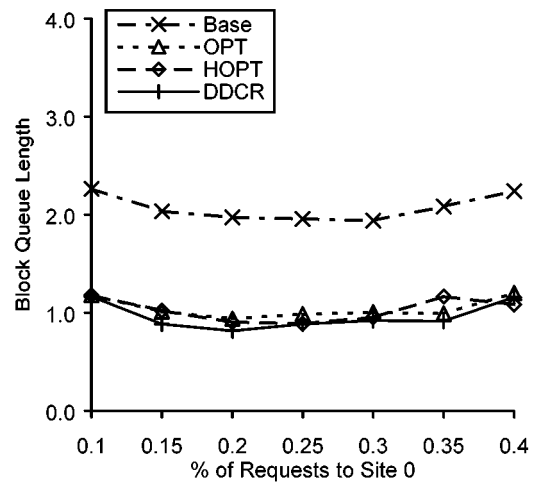


FIGURE 16. Percentage of data requests to site 0 on mean block queue length (memory).

operations with required data items located at site 0. When $P_{\text{site}(0)}$ is set to be 0, no transaction will access the data items at site 0. On the contrary, when $P_{\text{site}(0)}$ is defined to be 1, all operations of the transactions will access the data items at site 0. The distribution of the required data items of the operations to the other sites, site 1 to site 3, is spread evenly.

As the value of $P_{\text{site}(0)}$ will affect the distribution of workload in the system, it is expected that the performance of the system will be poorer if the distribution of workload in the system is not even. This is confirmed by Figure 15, which shows U-shape curves with the minimum point at $P_{\text{site}(0)} \approx 0.25$. When the value of $P_{\text{site}(0)}$ is smaller than 0.25, i.e. the workload at site 0 is lighter than at the other sites, the MRs for the four protocols will decrease when the value of $P_{\text{site}(0)}$ increases because the workload in the system becomes more even. When the value of $P_{\text{site}(0)}$ is greater than 0.25, e.g. the workload at site 0 is heavier, further increases in the value of $P_{\text{site}(0)}$ degrade the system performance as site 0 now becomes the bottleneck.

Consistent with the results in previous sub-sections, the performance of the DDCR and the optimistic protocols is much better than the base protocol, especially when the workload in the system is not even, e.g. the value of $P_{\text{site}(0)}$ is very small or very large. For example, when the value of $P_{\text{site}(0)}$ is 0.1, the MRs for the DDCR and Opt are about 17% and 13% smaller than the MR for the base protocol. This is because with an uneven distribution of workload in the system, the commit protocol requires more time to complete. Thus, the impact of the executing–committing conflicts on the system performance will be greater. Figure 16 shows the mean block queue length for the four protocols. The base protocol has the longest mean block queue length. Comparing the DDCR with the optimistic protocols, the performance of the DDCR is still better than the Opt and HOPT with the differences in MR ranging from 3% to 5%.

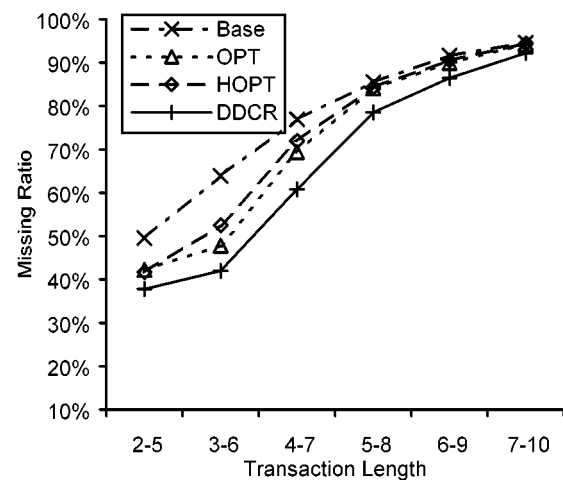


FIGURE 17. Transaction length on missing ratio (memory).

5.3.6. The impact of transaction length

Figure 17 shows the MRs for the protocols at different transaction lengths. Increasing the length of the transactions not only increases the probability of data conflict, it may also affect the performance of the DDCR as the DDCR assumes that if a transaction enters the commitment stage first, it should complete the commitment stage earlier. Otherwise, the committing transaction may be assumed to have problems in its commitment. Other transactions, which depend on it, may reverse the dependency. For a transaction with a longer length, it may take longer to complete its execution stage and commitment stage. So, the assumption made in the DDCR may not always be true.

As shown in Figure 17, although the DDCR gives the best performance even at a longer transaction length, the differences among the performance of the protocols decrease with an increase in transaction length. When the transactions are longer, the importance of the executing–committing conflicts relative to the executing–executing conflicts will

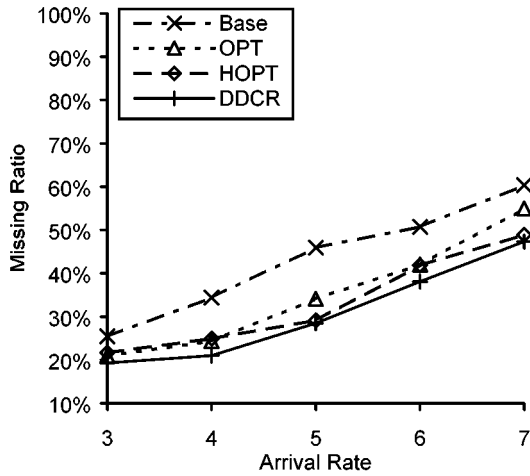


FIGURE 18. Arrival rate on missing ratio (disk).

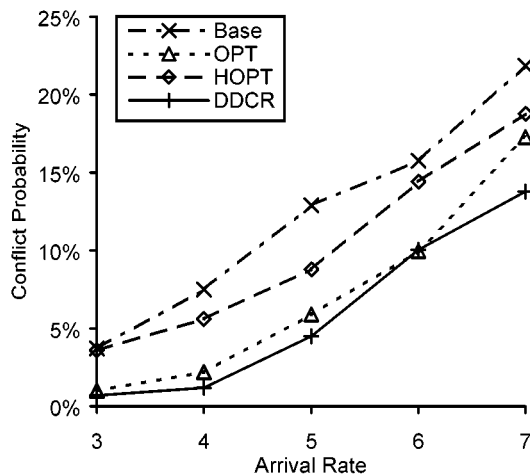


FIGURE 19. Arrival rate on conflict probability (disk).

become smaller as the time for the commitment stage will be relatively shorter compared to the execution stage. However, even at a longer transaction length, the DDCR still shows a significant improvement over the optimistic protocols and the base protocols.

5.3.7. The protocol performance in disk resident DRTDBS

In the previous sub-sections, we have studied the performance of the DDCR as compared with the optimistic protocols and the base protocol in a main memory DRTDBS (MDRTDBS). In this sub-section, we examine their performance in a disk-resident DRTDBS (DDRTDBS) model. The main difference between the DDRTDBS model and the MDRTDBS model is in the access of the data items. In the DDRTDBS, the accesses to the database are performed by the disk scheduler, which uses a non-preemptive policy.

Due to the non-preemptive scheduling, it is expected that the improvement in performance by the DDCR and the optimistic protocols as compared with the base protocol will be smaller as the transactions will have a higher probability of missing their deadlines. Figures 18 and 19

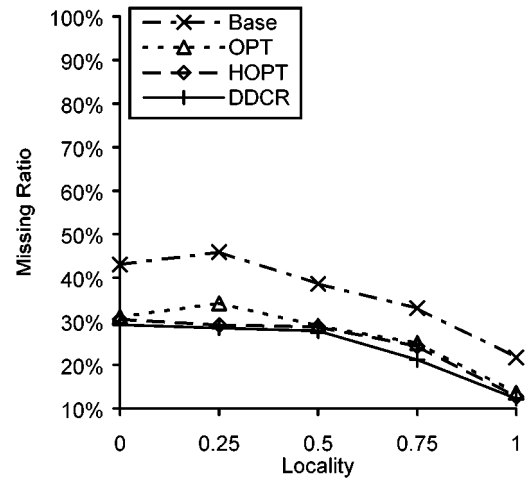


FIGURE 20. Locality on missing ratio (disk).

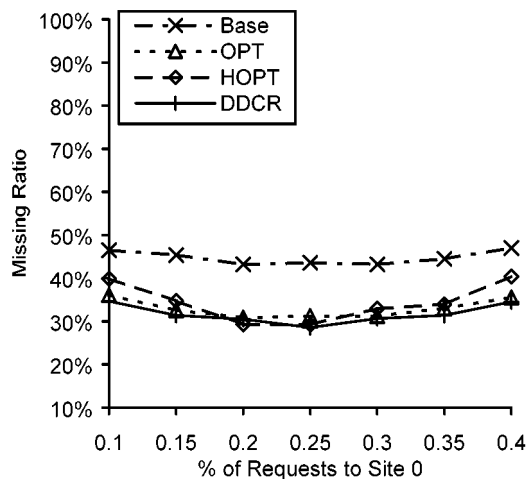


FIGURE 21. Percentage of data requests to site 0 on missing ratio (disk).

show the impacts of arrival rates on the performance of the four protocols in the DDRTDBS model. Consistent with the results for the MDRTDBS model, the MRs for the DDCR and the optimistic protocols are much smaller than the base protocol (see Figure 18). At arrival rate of 7 transactions/s, the MRs for the DDCR and Opt are about 12% and 6% smaller respectively than that for the base protocol. The conflict probability of the base protocol is always the highest, as shown in Figure 19.

The impact of the locality ratio and skewed workload on the performance of the protocols in the DDRTDBS model is shown in Figures 20 and 21 respectively. Similar to the results in the MDRTDBS model, the performance of the DDCR is still the best, although the improvement is smaller compared with the improvement obtained from the MDRTDBS model, due to the effect of the non-preemptive disk scheduling.

Figure 22 shows the impact of transaction length on the performance of the protocols at a DDRTDBS. Consistent with the results for the main-memory resident DRTDBS,

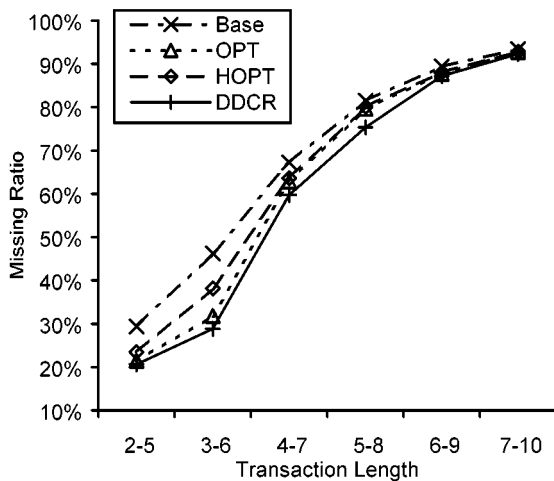


FIGURE 22. Transaction length on missing ratio (disk).

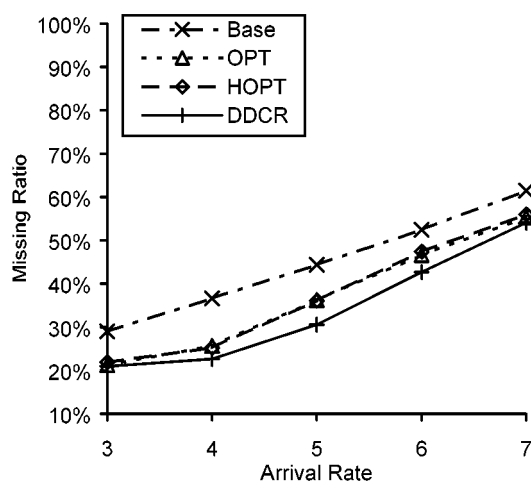


FIGURE 23. Arrival rate on missing ratio (disk/with failure).

the performance of the DDCR is better than the others and the base protocol gives the worse performance. When we compare their performance in Figure 22 with the results shown in Figure 17, it can be observed that the differences in their MRs are smaller. This is consistent with our expectation as the non-preemptive disk scheduling in the DDRTDBS will affect the effectiveness of the real-time protocols.

5.3.8. The impact of failure at commitment

In this set of experiments, we study the impact of temporary failure at commitment on the performance of the protocols. The probability of failure is defined by the variable P_{failure} .

The MRs for the DDCR, optimistic protocols and the base protocol in the DDRTDBS model at different arrival rates are shown in Figure 23. As we can see in Figure 23, the MRs for the DDCR are consistently smaller than the Opt and HOPT which are lower than the base protocol. The MR for the base protocol increases linearly with an increase in arrival rate. However, the impact of arrival rate on the DDCR and the

optimistic protocols is much smaller. When the arrival rate is 5 transactions/s, the MRs for the DDCR and Opt are about 14% and 8% smaller than the MR for the base protocol. The differences in the MRs between the four protocols become smaller at higher workload as more transactions will miss their deadlines due to the failures instead of due to the executing–committing conflicts. However, the MR for DDCR is still lower than the optimistic protocols and base protocol even when arrival rate is 7 transactions/s.

6. CONCLUSIONS

Concurrency control is an important issue in the design of DRTDBSs. Most of the proposed real-time concurrency control protocols focus on resolving data conflicts among executing transactions. However, the executing–committing conflict problem is also a very important issue for the performance of a DRTDBS, since communication delays and the possibility of failures during commitment can make the time to complete the commitment procedure very long. Under the traditional two-phase commit protocol, some transactions may be blocked to miss their deadlines due to conflicts with committing transactions. Furthermore, the blocked transactions may induce more lock conflicts as they are holding some locks. Consequently, the system may result in thrashing due to data contention. Although the optimistic protocols [1, 2] have been proposed with an attempt to increase the system concurrency, the dependency problem in the optimistic protocols can make a transaction miss its deadline while it is waiting for its dependent transaction to commit.

In this paper, we have proposed a new protocol, called deadline-driven conflict resolution, to resolve the executing–committing conflicts. With DDCR, most such conflicts will not cause the system performance to be degraded. An optimistic approach is used to increase the system concurrency. A deadline-driven approach is used to resolve the dependency problem by reversing the dependencies if the committing transaction has already spent a long time in its voting phase. The purpose is to reduce the impact of a committing transaction, especially when it encounters some temporary problems in its commitment, on other transactions which depend on it. Serializability of the schedule is ensured by checking the before-sets and after-sets when a transaction wants to enter the decision phase.

The performance of the DDCR has been compared with the optimistic protocols and the base protocol, which uses blocking with priority inheritance to resolve the executing–committing conflicts. The results of performance experiments have shown that significant improvement in both main memory DRTDBS and disk-resident DRTDBS can be obtained with the use of the DDCR as compared with the base protocol and the optimistic protocols, as a result of higher concurrency and less impact of commit dependency on the system performance. With the use of the DDCR, the resulting system will also be more resilient to the failures in transaction commitment.

REFERENCES

- [1] Gupta, R., Haritsa, J., Ramamritham, K. and Seshadri, S. (1996) Commit processing in distributed real-time database systems. In *Proc. Real-time Systems Symposium*, Washington DC. IEEE Computer Society Press, San Francisco.
- [2] Gupta, R., Haritsa, J. and Ramamritham, K. (1997) More optimistic about real-time distributed commit processing. In *Proc. 1997 Real-time Systems Symp.*
- [3] Bernstein, P. A., Hadzilacos, V. and Goodman, N. (1987) *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, Reading, MA.
- [4] Yu, P. S., Wu, K. L., Lin, K. J. and Son, S. H. (1994) On real-time databases: concurrency control and scheduling. *Proc. IEEE*, **82**, 140–57.
- [5] Lam, K. Y. and Hung, S. L. (1995) Concurrency control for time-constrained transactions in distributed database systems. *Comp. J.*, **38**, 704–715.
- [6] Son, S. H. and Zhang, F. (1995) Real-time replication control for distributed database systems: algorithms and their performance. In *Proc. 4th Int. Conf. on Database Systems for Advanced Applications*, pp. 214–221. World Scientific Press, Singapore.
- [7] Ulusoy, O. and Belford, G. G. (1993) Real-time transaction scheduling in database systems. *Inf. Syst.*, **18**, 559–580.
- [8] Ulusoy, O. (1994) Processing of real-time transactions in a replicated database system. *Distrib. Parallel Databases*, **2**, 405–436.
- [9] Gray, J. and Reuter, A. (1993) *Transaction Processing: Concept and Techniques*. Morgan Kaufmann Inc., USA.
- [10] Chen, Y. W. and Gruenwald, L. (1996) Effects of deadline propagation on scheduling nested transactions in distributed real-time database systems. *Inf. Syst.*, **21**, 103–124.
- [11] Lam, K. Y., Law, G. C. K. and Lee, C. S. (1997) Scheduling of triggered transactions in distributed real-time active databases. In *Proc. 2nd Int. Workshop on Active, Real-Time, and Temporal Database Systems (ARTDB-97) (Lecture Notes in Computer Science)*. Springer-Verlag, Berlin.
- [12] Ulusoy, O. (1995) A study of two transaction processing architectures for distributed real-time database systems. *J. Syst. Softw.*, **31**, 97–108.
- [13] Agrawal, D., Abbadi, E., Jeffers, R. and Lin, L. (1995) Ordered shared locks for real-time databases. *VLDB J.*, **4**, 87–126.
- [14] Hong, D., Johnson, T. and Chakravarthy, S. (1993) Real-time transactions scheduling: a cost-conscious approach. In *Proc. ACM Int. Conf. on Management of Data*. ACM Press, Washington DC.
- [15] Lee, J. (1994) Concurrency control algorithms for real-time database systems. PhD Thesis, Department of Computer Science, University of Virginia.
- [16] Ulusoy, O. and Buchmann, A. (1998) A real-time concurrency control protocol for main-memory database systems. *Inf. Syst.*, **23**, 109–125.
- [17] Soparkar, N., Levy, E., Korth, H. and Silberschatz, A. (1992) Adaptive commitment for real-time distributed transactions. *TR-92-15*, Department of Computer Science, University of Texas-Austin.
- [18] Yoon, Y., Cho, J. and Han, C. (1996) Real-time commit protocol for distributed RT database systems. In *Proc. Second IEEE Int. Conf. on Engineering of Complex Computer Systems*, Canada. IEEE Computer Society Press, Los Alamitos, CA.
- [19] Liu, C. L. and Layland, J. W. (1976) Scheduling algorithms for multi-programming in a hard-real-time environment. *J. ACM*, **20**, 41–64.
- [20] Abbott, R. and Garcia-Molina, H. (1992) Scheduling real-time transactions: a performance evaluation. *ACM Trans. Database Syst.*, **17**, 513–560.
- [21] Haritsa, J. R., Carey, M. J. and Livny, M. (1992) Data access scheduling in firm real-time database systems. *J. Real-Time Syst.*, **4**, 203–242.
- [22] Huang, J., Stankovic, J., Towsley, D., Ramamritham, K. and Purimetla, B. (1992) Priority inheritance in soft real-time databases. *Real-time Syst.*, **3**, 243–268.
- [23] Ramamritham, K. and Chrysanthos, P. K. (1996) A taxonomy of correctness criteria in database applications. *VLDB J.*, **5**, 85–97.
- [24] Lam, K., Lee, V. C. S., Lam, K. and Hung, S. (1996) Distributed real-time optimistic concurrency control protocol. In *Proc. 4th International Workshop on Parallel and Distributed Real-time Systems*, pp. 122–125. IEEE Computer Society Press, Hawaii.