A Token-Based Synchronization Scheme using Epsilon-Serializability and Its Performance for Real-Time Distributed Database Systems

Sang H. Son and Spiros Kouloumbis

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

ABSTRACT

Schedulers for real-time distributed replicated databases must satisfy timing constraints of transactions and preserve data consistency. In this paper, we present a replication control algorithm, which integrates real-time scheduling and replication control. The algorithm adopts a token-based scheme for replication control and attempts to incorporate the urgency of real-time transactions into the conflict resolution policies. The algorithm employs epsilon-serializability, new correctness criterion which is less stringent than conventional one-copy-serializability. The performance of the algorithm is evaluated using a simulation environment.

1. Introduction

In real-time distributed database systems, transactions must be scheduled to meet the *timing constraints* and to ensure *data consistency* [Son90]. Real-time task scheduling can be used to enforce timing constraints, while concurrency control is employed to maintain data consistency. Unfortunately, the integration of the two mechanisms is non trivial because of the trade-offs involved. Serializability may be too strong as a correctness criterion for concurrency control in database systems with timing constraints, for serializability limits concurrency. As a consequence, data consistency might be compromised to satisfy timing constraints.

In real-time scheduling, tasks are assumed to be independent, and the time spent synchronizing their access to shared data is assumed to be negligible compared with execution time. Knowledge of resource and data requirements of tasks is also assumed to be available in advance.

In replication control methods, on the other hand, the objective is to provide a high degree of concurrency and thus faster average response time without violating data consistency [Son87]. Two different policies can be employed in order to synchronize concurrent data access of transactions: *blocking* and *aborting*. However, blocking may cause priority inversion in which a high priority transaction is blocked by lower priority transactions. Aborting lower priority transactions wastes the work done by them. Thus, both policies have negative effects on time-critical scheduling.

Conventional replication control algorithms are synchronous, in the sense that they require the atomic updating of some number of copies. This leads to reduced system availability and decreased throughput as the size of the system increases. On the other hand, asynchronous replication control methods that would allow more transactions to meet their deadlines suffer from a basic problem: the system enters an inconsistent state in which replicated copies may not share the same value. Standard correctness criteria such as the *1-copy serializability* (1SR) [Ber87] are thus hard to attain with asynchronous consistency control.

A less stringent, general-purpose consistency criterion is necessary. The new criterion should allow more real-time transactions to satisfy their timing constraints by temporarily sacrificing database consistency to some small degree. *Epsilon-serializability* (ESR) is such a correctness criterion, offering the possibility of maintaining mutual consistency of replicated data asynchronously [Pu91]. Inconsistent data may be seen by certain query transactions, but data will eventually converge to a consistent (1SR) state. Additionally, the degree of inconsistency can be controlled within a specified threshold.

The goal of our work is to design a replication control algorithm that supports transactions to meet their deadlines and at the same time maintains data consistency in the absence of any *a priori* information. Our algorithm is based on a token-based synchronization scheme for replicated data. Real-time scheduling features are developed on top of this platform and epsilon-serializability is employed as the correctness criterion.

2. Transactions and Tokens

The smallest unit of data accessible to the user is called *data object*. A data object is an abstraction that does not correspond directly to a real database item. In distributed database systems with replicated data objects,

This work was supported in part by ONR, DOE, and IBM.

a logical data object is represented by a set of one or more replicated physical data objects.A *transaction* is a sequence of operations that takes the database from a consistent state to another consistent state. Two types of transactions are allowed in our environment: *query* transactions and *update* transactions. Query transactions consist only of read operations that access data objects and return their values to the user. Update transactions consist of both read and write operations.

Transactions arriving at the system are assumed to be non-periodic. A globally *unique timestamp* is generated for each transaction [Lam78]. Each time a transaction is aborted and resubmitted, a new timestamp value is assigned to it. If a transaction T_1 has a smaller timestamp than another transaction T_2, we say that T_1 is the *older* transaction and T_2 is the *younger* one.

A *token* designates a read-write copy. Each logical data object has a predetermined number of tokens, and each token copy is the latest version of the data object. The site which has a token-copy of a logical data object is called a *token site*, with respect to the logical data object. In order to control the access to data objects, the system uses timestamps. When a write operation is successfully performed and the transaction is committed, a new version is created which replaces the previous version of the token copy.

When a transaction performs a write operation to a physical data object, there are two values that are associated with the data object: the *after-value* (the new version) and the *before-value* (the old version). Because the before-value is available during the transaction processing, it is natural to ask if concurrency can be improved by giving out this value [Bay80].

Let T_1 be the transaction which already issued an access request, and T_2 cause the conflict. For each token copy of X, conflicts are resolved as the following [Son89]:

(1) RW conflict: If T_2 is younger than T_1, then it waits for the termination of T_1. If T_2 is older than T_1, then it reads before-value of X.

(2) WR conflict: If T_2 is younger than T_1, then its write request is granted with the condition that T_2 cannot commit before the termination of T_1. If T_2 is older than T_1, then T_2 is rejected.

(3) WW conflict: If T_2 is younger than T_1, then it waits for the termination of T_1. If T_2 is older than T_1, then T_2 is rejected.

The coordinator of an update transaction maintains the *before-list* (BL), a list of transactions which read the before-value of any data object in its write set, and the *after-list* (AL), a list of transactions which write the after-value of any data object in its read set. The BL and AL are used during the commitment phase of every update transaction.

When a transaction T_2 reads the before-value of a data object locked by T_1, the token-site which gives the before-value, conveys the identifier of T_2 to the coordinator of T_1. Hence, the identifier of T_2 is inserted in the before-list of T_1, which stores all the transactions that read the before-values of any data object in T_1's write-set. The transaction manager at the read-only site of T_2 also conveys the identifier of T_1 to the coordinator of T_2. Actually, the identifier of T_1 is inserted in the after-list of T_2, which stores all the transactions that write the after-value of any data object in T_2's read-set.

When a transaction terminates (either commits or aborts), the coordinator of the terminating transaction must inform the coordinator of each transaction in its AL about the termination by sending Termination Messages (TM). On receiving a TM from the coordinator of a transaction in its BL, the coordinator of the active transaction removes the identifier of the terminating transaction (sender of the TM) from the BL. A transaction can commit only when its BL is empty. By this way, we prevent non-serializable execution sequences to occur.

Update transactions have their own *private workspace* where they initially apply their write operations. Update transactions commit by employing a two-phase protocol. In the first phase (*vote-phase*), an update transaction sends an update message to each token-site of every data object in its write-set. The transaction waits until it gets a response from all the token-sites for each data object. If all token-sites vote YES, then the transaction enters the second phase (*commit phase*). It sends the actual value of each data object to be written to the respective token-sites. Update messages to non token-sites can be scheduled after commitment. Therefore, a temporary and limited difference among object replicas is permitted; these replicas are required to converge to the standard 1SR consistency as soon as all the update messages arrive and are processed. An update transaction that executes its commit phase can never be aborted, even if it potentially conflicts with another transaction.

Query transactions fall into three different categories as far as the correctness of their response is concerned:

- *Required consistent queries.* Queries are specified as such when they are first submitted by the user, and they are always guaranteed to return consistent data;
- *Consistent queries.* Their final output is correct regardless of any requirement by the user;
- *Possibly inconsistent queries.* In case of such a

query, there exists a small possibility that returned values of a replicated data object might reflect an inconsistent state of the database.

3. Epsilon-Serializability

Epsilon-serializability (ESR) is a correctness criterion that enables asynchronous maintenance of mutual consistency of replicated data [Pu91]. A transaction with ESR as its correctness criterion is called an *epsilon-transaction* (ET). An ET is a query ET if it consists of only reads. An ET containing at least one write is an update ET. Query ETs may see an inconsistent data state produced by update ETs. The metric to control the level of inconsistency a query may return is called the *overlap*. It is defined as the set of all update ETs that are active and affecting data objects that the query seeks to access. If a query ET's overlap is empty, then the query is serializable.The overlap of an active query transaction Q can be used as an *upper bound of error* on the degree of inconsistency that Q may accumulate. Given that we are interested in how many update transactions overlap with Q more than which transactions those are, the term overlap, in its further usage, will reflect the cardinality of the set of update transactions that conflict with the query ET Q.

Among several replica control methods based on ESR, we have chosen the ordered updates approach [Pu91]. The ordered updates approach allows more concurrency than 1SR in two ways. First, query ETs can be processed in any order because they are allowed to see intermediate, inconsistent results. Second, update ETs may update different replicas of the same object asynchronously, but in the same order. In this way, update ETs produce results equivalent to a serial schedule; these results are therefore consistent.

There are two categories of transaction conflicts that we examine: conflicts between update transactions and conflicts between update and query transactions.

Conflicts between update transactions can be either *RW* conflicts or *WW* conflicts. Both types must be strictly resolved. No correctness criteria can be relaxed here, since execution of update transactions must remain 1SR in order for replicas of data objects to remain identical.

Conflicts between update and query transactions are of *RW* type. Each time a query conflicts with an update, we say that the query overlaps with this update, and the overlap counter is incremented by one. If the counter is still less than a specified upper bound, then both operation requests are processed normally, the conflict is ignored, and no transaction is aborted. Otherwise, RW conflict must be resolved by using the conventional 1SR correctness criteria of the accommodating algorithm.

The performance gains of the above conflict resolution policies are numerous. Update transactions are rarely blocked or aborted in favor of query transactions. They may be delayed on behalf of other update transactions in order to preserve internal database consistency. On the other hand, query transactions are almost never blocked provided that their overlap upper bound is not exceeded. Finally, update transactions attain the flexibility to write replicas in an asynchronous manner.

4. Real-Time Issues

In real-time databases, transactions are characterized by their timing constraints and their data and computation requirements. Timing constraints are expressed through the *release time* and the *deadline*. Computation requirements for transactions are unknown, and no run-time estimate is available for every transaction that enters the system. Our goal is to minimize the number of transactions that miss their deadlines.

The real-time scheduling part of our scheme has three components: a policy to determine which transactions are eligible for service, a policy for assigning priorities to transactions, and a policy for resolving conflicts between two transactions that want to lock the same data object. None of these policies needs any more information about transactions than the deadline and the name of the data object currently being accessed.

All transactions which are currently *not tardy* are eligible for service. Transactions that have already missed their deadlines are immediately aborted. When a transaction is accepted for service at the local site where it was originally submitted, it is assigned a priority according to its deadline. The transaction with the *earliest deadline* has the highest priority. This policy meshes efficiently with the "not tardy" eligibility policy adopted above, so that transactions that have already missed their deadlines are automatically screened out before any priority is assigned to them. *High priority* is the policy that is employed for resolving transaction conflicts. Transactions with the highest priorities are always favored. The favored transaction, i.e. the winner of the conflict, gets the resources that it needs to proceed. The loser relinquishes control of any resources that are needed by the winner. The loser transaction will either be aborted or blocked depending on the relative age of the two conflicting transactions and the special provisions made by the replication control scheme.

5. Replication Control Scheme

In this section, we present the token-based replication control scheme in detail, along with the embedded ESR correctness criteria and real-time constraints.

5.1 Controlling Inconsistency of Queries

Queries are only involved in RW/WR conflicts. When a query transaction is submitted to the system, the user may quantify it with the restriction *"required to be consistent."* Such a characterization means that all possible future RW/WR conflicts between this query and update transactions will have to be resolved in a strict (1SR) way. In other words, *consistent queries* (*CQs*) are treated in the same fashion as update transactions. Values returned by *CQs* are always correct, reflecting the up-to-date state of the respective data objects.

If no consistency constraints are specified explicitly by the user on a submitted query, then the *ESR correctness criterion* is employed to maintain the query's consistency. The overlap upper bound is computed, and an overlap counter is initialized to zero. Each time the query conflicts with an update transaction over the same data object and the counter is less than the overlap upper bound, the conflict is ignored, the counter is incremented, the query reads the value of the data object in question and proceeds to read the next object. When the overlap counter is found to be equal to the upper bound, current and all subsequent conflicts must be resolved in a strict manner, so that no more inconsistency will be accumulated on the query.

When a query transaction eventually commits, the user is able to determine the degree of correctness of the data values returned. If the query was qualified as a *CQ*, then the user can be confident that the values returned are consistent. For regular query transactions, the private overlap counter is checked. If the counter is still zero, this means that no conflict has occurred throughout the entire execution of the query and the results must again be perfectly accurate. Such a query falls into the *CQ* class. An overlap counter greater than zero indicates that a certain number of conflicts with update transactions remained unresolved; the query had seen some possibly inconsistent states, and might yield some inaccurate data. This last type of query falls into the *"possibly inconsistent"* queries class.

Since arbitrary queries may produce results beyond allowed inconsistency even within its overlap limit, it is important to restrict ET queries to have certain properties that permit tight inconsistency bounds. A first attempt in this approach is proposed in [Ram91]. It is beyond the scope of this paper to deal with such strategies. In the remainder of the paper, we assume that inconsistency bounds can be enforced by the system if necessary.

5.2 Conflict Resolution

Mechanisms for conflict resolution between update transactions comprise the core of our scheme. Query transactions need not be considered separately because queries that are forced to resolve their RW conflicts with update transactions can be treated as update transactions.

We examine three separate categories of conflicts, and for each category we present a table of all possible conflicts between younger and older transactions, and between higher priority and lower priority transactions.

Before we get into the detailed discussion of conflict resolution in each case, we must clarify what we mean by "conditional abort." Let T_1 be an accepted and successfully processed transaction issuing a write request on data object X. Let T_2 be another transaction attempting to read or write X. In the case that T_1 has to be aborted (possibly for real-time priority reasons or to preserve database consistency), T_1's phase of commitment must first be checked. If T_1 is in the vote-phase, it can be aborted normally. All writes on T_1's private workspace will be discarded, and T_1 will be resubmitted, provided that it has not already missed its deadline. However, if T_1 is in the commit-phase and update messages have already been sent to token-sites, then T_1 cannot be aborted. Consequently, "conditional abort" refers to the action of aborting an update transaction T only in the case where T is still in the vote-phase. If T_1 and T_2 are in conflict and T_1 has entered the second phase of commitment, then T_2 must be aborted in order for the database to remain consistent.

(1) R - W Conflict.

Transaction T_2 requests to read a data object X for which transaction T_1 has already issued a write request.

If T_2 is younger than T_1, then the original token-based scheme (Section 2) requires that T_2 must wait for the termination of T_1 before it reads the value of X. In case T_2 has lower priority than the priority of T_1, this requirement does not contradict with the priority status and T_2 can wait for T_1 to terminate. On the other hand, if T_2 has higher priority than T_1, we cannot apply this rule and force T_2 to wait. Instead, T_2 must read the data object and proceed whereas T_1 has to be conditionally aborted in order for the consistency of data read by T_2 to be preserved.

If T_2 is older than T_1, then T_2 should be allowed to read the before-value of X and proceed. T_2 is then inserted in the before-list (BL) of T_1, and T_1 is inserted in the after list of T_2. According to the commitment criteria of the token-based scheme, T_1 has to wait for T_2 to terminate before it can commit. Such resolution is acceptable when T_2 has higher priority than T_1. However, in the case that T_2 has lower priority than T_1, we conditionally abort T_2 when T_1 requests to commit in order to maintain the consistency of data written by T_1.

(2) W - R Conflict.

Transaction T_2 requests to write data object X for which transaction T_1 has already issued a read request.

If T_2 is younger than T_1, then T_1 should be allowed to read the before-value of X and T_2 write an after-value of X. Moreover, T_1 is inserted in the BL of T_2, and T_2 is inserted in the AL of T_1. In this way, T_2 has to wait for the termination of T_1 before it can commit. In the case that T_2 has lower priority than T_1, there is no contradiction between its priority status and its obligation to be blocked. In the case that T_2 has a higher priority than T_1, T_1 is conditionally aborted when T_2 requests to commit.

If T_2 is older than T_1, then T_2 should be conditionally aborted and T_1 reads the correct, up-to-date value of X. Aborting T_2 is perfectly justified in the case T_2 has a lower priority. In the case that T_2 has a higher priority, aborting T_2 would violate the real-time constraints. Therefore, we let T_2 proceed and write a new value for X while T_1 is aborted, since it has seen a value of X that has already become obsolete.

(3) W - W Conflict.

Transaction T_2 requests to write data object X for which transaction T_1 has already issued a write request.

If T_2 is younger than T_1, then T_2 should wait for the termination of T_1 before it writes a new value for data object X. Such conflict resolution favors T_1 and is compatible with the situation where T_2 has lower priority than T_1. However, when T_2 has a higher priority, it is not required to wait for the lower priority transaction T_1. Hence, T_2 will proceed, and T_1 will be conditionally aborted in order for the database to remain internally consistent.

If T_2 is older than T_1, then T_2 should be aborted. Note that we are interested only in the most recent value of X, i.e. the value written by the younger T_1 transaction. In the case that T_2 has a lower priority, the above resolution is acceptable, since the higher priority T_1 is favored to proceed. On the contrary, when T_2 has the higher priority, T_2 must be allowed to write its own new value of X, and T_1 must be conditionally aborted for the database to remain consistent with respect to the data object X.

5.3. Transaction Commit

The coordinator of a transaction decides to commit when the following conditions are satisfied:

- The transaction must not have missed its deadline;
- All the token-sites of each data object in the write-set of the transaction have precommitted (this only applies to update transactions);
- There is no active transaction that has seen before-value of any data object in the transaction's write-set. In other words, the *before-list* of the transaction must be empty (this only applies to update transactions).

6. Performance Results

In this section we compare the above real-time replication control scheme (RTS) with the respective conventional non real-time scheme (NRTS) on which our algorithm is based. We present a number of performance experiments for the two approaches under various assumptions about the transaction load, the percentage of update transactions, and the database size.

A real-time distributed database prototyping environment was used to build the simulation program [Son92b]. The environment provides the user with multiple threads of execution and guarantees the consistency of concurrently executing processes. Several requests can be submitted at the same time, and many read/write operations take place simultaneously at different sites.

The performance metric employed is the percentage of transactions that missed their deadlines (% missed) [Abb92] in the total number of transactions that were submitted to the system during the simulation period.

Certain parameters that determine system configuration and transaction characteristics remain fixed throughout the experiments: the database size (1000 data objects), the transaction size (12 data objects), the computation cost per update (8 msec), the I/O cost (20 msec), the percentage of operations that are updates in each transaction (40%), the abort cost for each transaction (19 msec), and the overlap factor (0.03, i.e. 3% of the queries return possibly incorrect data). These values are not meant to model a specific distributed database application, but were chosen as reasonable values within a wide range of possible values. In particular, we want transactions to access a relatively large fraction of the database (1.2%) so that conflicts occur more frequently.

Parameters used as independent variables in one-variable functions describing the % missed deadlines performance metric are the mean inter-arrival time of transactions (varying between 10msec and 80msec), the database size (varying from 200 to 1,000 data objects), and the percentage of read-only transactions submitted to the system (varying from 10% up to 90%). We assume that the database is fully replicated at all sites. A standard, modulo-based formula is used to assign token-sites to a data object.

6.1. Transaction Inter-arrival Time

In this experiment we vary the inter-arrival time of transactions from 10msec (for a heavily loaded system),

to 80msec (for a non saturated system state). The database has a size of 1,000 data objects and 40% of the transactions submitted to the system are read only.

Figures 1 and 2 show that curves start from a high percentage of missed deadlines and descend until a minimum percentage is reached at the vicinity of 80msec. When the inter-arrival time is 10msec, the system is overloaded by a huge number of transactions striving to access a limited number of data objects (hot spots). Consequently, the number of conflicts becomes so large that transactions will be blocked and eventually miss their deadlines. As the inter-arrival time increases, the number of active transactions per second in the system decreases. Fewer transactions conflict with each other and need to be blocked and, as the graphs show, the vast majority of the transactions meet their deadlines.

RTS performs better than NRTS in all cases. When we have a small number of token-sites (i.e. 1 token-site out of 5 sites, or 3 token-sites out of 10 sites), the RTS and the NRTS curves are relatively close to each other. The greater the number of token-sites becomes, the further the respective RTS and NRTS curves veer away from each other. Finally, when all sites are token-sites for each data object, we observe the maximum distance between the curves for the "% missed deadlines" of the two schemes.

The results shown in Figures 1 and 2 clearly favor RTS. This was expected, since in the majority of the cases RTS schedules the most urgent transactions first, while NRTS is not at all sensitive to how close a transaction is to missing its deadline. The difference in performance of the two algorithms becomes even greater when the number of token-sites increases. A large number of token-sites means that each update transaction T must request consent for commitment from a large number of sites. Therefore, T must remain pending for a longer period of time, and consequently T will be exposed to more conflicts with other transactions. The increased number of conflicts that must be resolved, combined with the fact that conventional NRTS does not incorporate any intelligent real-time conflict resolution policy, makes the inferiority of NRTS even more apparent.

6.2. Read Only Transactions

We vary the percentage of read-only transactions submitted to the system from 10%, where the majority of the transactions are updates, to 90%, in which case the system behaves almost like a static database, where very few values of data objects change. We assume an average inter-arrival time of 30msec, and a database size of 1,000 data objects.

Figures 3 and 4 show that both RTS and NRTS behave similarly. The respective "% missed deadlines" curves start from a very high missed-deadline percentage, in the neighborhood of 10% (which is reasonable given that 90% of the transactions are update ones) and descend very steeply as the percentage of read only transactions increases. In the vicinity of 90%, almost all transactions are queries, very few conflicts occur, and the two approaches perform identically with each other.

When the percentage of read only transactions is near 10%, then the conflict rate between update transactions is very high, and RTS clearly exhibits lower miss rates than NRTS. When the percentage of read only transactions is 10%, the whole burden of scheduling is shifted onto the conflicting-updates resolution policy. NRTS allows time-critical updates to be aborted and restarted in order for less critical transactions to proceed, provided that the database remains consistent. On the contrary, RTS adopts the time criticality of a transaction as the first criterion for resolving conflicts with other transactions: less critical transactions are blocked or even aborted whenever necessary in order for transactions closer to their deadlines to proceed freely. An increased number of token-sites makes the difference in performance even greater for the same reasons as mentioned in the previous section.

The smaller the percentage of read-only transactions becomes, the fewer conflicts occur and the less important role the conflict resolution mechanism plays. Therefore, the two approaches behave almost identically near the 90% point.

6.3. Database Size

We assume an average inter-arrival time of 30msec, and 40% read only transactions. We vary the database size from 200 data objects to 1,000 data objects. Figures 5 and 6 show that RTS performs much better than NRTS. The distance between the curves is great throughout the spectrum of possible database sizes. When the database size is very small, in the vicinity of 200 data objects, the possibility that two transactions might conflict is very high given that the transaction size remains constant at 12 data objects. As a result, more conflicts between update transactions will have to be resolved. More conflicts means that more sophisticated real-time scheduling is necessary to meet the maximum possible number of deadlines. Moreover, a small-size database causes more queries to conflict with pending updates. In such case, ESR comes into play; queries in RTS are allowed to overlap freely with a limited number of updates, and neither of them need to be blocked or aborted as the strict 1SR criteria would require in NRTS. Controlled query inconsistency is a feature that works for the benefit of RTS and leads to further decrease of the missed deadlines

percentage.

7. Concluding Remarks

In this paper we have presented a synchronization scheme for real-time distributed database systems. The algorithm is based on a *token-based approach*, in which two additional components are built. The first is a set of *real-time constraints* that each transaction has to meet. A separate priority scheme is employed to reflect the demand of a transaction to finish before its deadline. The second component is the ESR correctness criterion with which query transactions have to comply. Instead of applying 1SR to all transactions, 1SR is applied only to updates, and queries are left free to be interleaved with updates in a more flexible way.

By relaxing the consistency criteria for query transactions, queries and updates hardly ever have to abort or block each other due to conflicts between them. As an immediate consequence of this, more transactions may terminate successfully before their deadlines expire. Additionally, the ESR further improves performance; updating the different replicas of the same data object is done asynchronously, but in the same order. Thus, logical write operations become disjoint from the corresponding physical write operations, and update transactions are free to proceed to the next step of their execution or even to commit. Internal database consistency is preserved strictly. Data returned by certain queries are allowed to exhibit limited inconsistency, under user control.

Another advantage of our scheme lies in the fact that there is very little information the user has to provide to achieve efficient system operation. No *a priori* knowledge of the kind or the number of the data objects that are included in the read-set or the write-set of a transaction is needed. The only information required is the kind of each submitted transaction (query or update). Moreover, no execution time estimate is required for each submitted transaction. It would be extremely difficult to compute a run-time estimate, especially in the distributed environments for which our scheme is designed.

There is a price to pay for relaxing correctness criteria and meeting more deadlines. Although the user can control the maximum permissible inconsistency of queries, one cannot know exactly which one transaction out of the set of all possibly inconsistent queries will return incorrect data, unless a tight inconsistency bound is provided. Note that an overlap counter greater than zero does not necessarily mean that the respective query transaction is inconsistent. It simply indicates that certain RW/WR conflicts were passed unresolved, and inconsistency might be present among the data values returned.

REFERENCES

[Abb92] R.Abbott, and H.Garcia-Molina, "Scheduling Real-time Transactions: a Performance Evaluation," ACM Trans. on Database Systems, vol. 17, no. 3, pp. 513-560, Sept. 1992

[Bay80] R. Bayer, H. Heller, and A. Reiser, "Parallelism and Recovery in Database Systems," ACM Trans. Database Syst. 5, 2, June 1980.

[Ber87] P.A.Bernstein, V.Hadzilacos, and N.Goodman, "Concurrency Control and Recovery in Database Systems," Addison-Wesley Publishing,,1987.

[Lam78] L.Lamport, "Time, Clocks, and the Ordering of Events in a Distributed System," Comm. ACM, vol. 21, no. 7, pp. 558 - 565, July 1978.

[Pu91] C.Pu, and A.Leff, "Replica Control in Distributed Systems: An Asynchronous Approach," ACM SIGMOD Conference, May 1991.

[Ram91] K. Ramamritham and C. Pu, "A Formal Characterization of Epsilon Serializability," Tech. Rep. 91-91, Dept. of Computer Science, Univ. of Massachusetts, Dec. 1991.

[Son87] S.H.Son, "Synchronization of Replicated Data in Distributed Systems," Information Systems, vol. 12, no. 2, pp. 191 - 202, 1987.

[Son89] S.H.Son, "A Resilient Replication Method in Distributed Database Systems," Proceedings of IEEE INFOCOM '89, Ottawa, Canada, April 1989.

[Son90] S.H.Son, "Real-Time Database Systems: A New Challenge," Data Engineering, vol. 13, no. 4, Special Issue on Future Directions on Database Research, December 1990.

[Son92] S.H.Son, J. Lee, and Y. LIn, "Hybrid Protocols using Dynamic Adjustment of Serialization Order for Real-Time Concurrency Control," Journal of Real-Time Systems, vol. 4, pp. 269-276, Sept. 1992.

[Son92b] S.H.Son, "An Environment for Integrated Development and Evaluation of Real-Time Distributed Database Systems," Journal of Systems Integration, vol. 2, no. 1, pp. 67-90, February 1992.

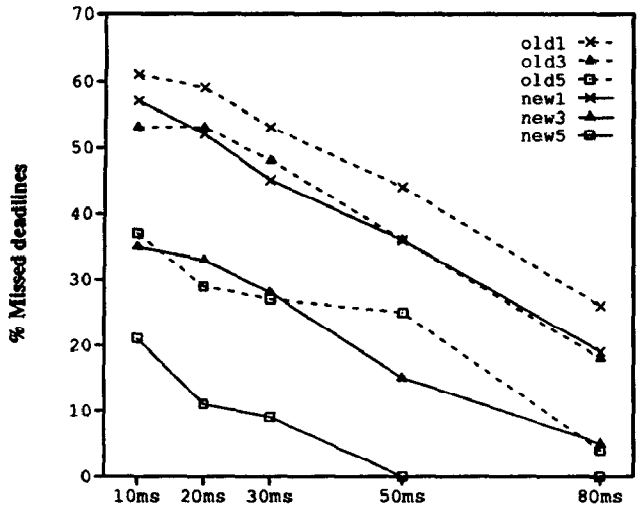

Figure 1: SITES=5, Interarrival time.

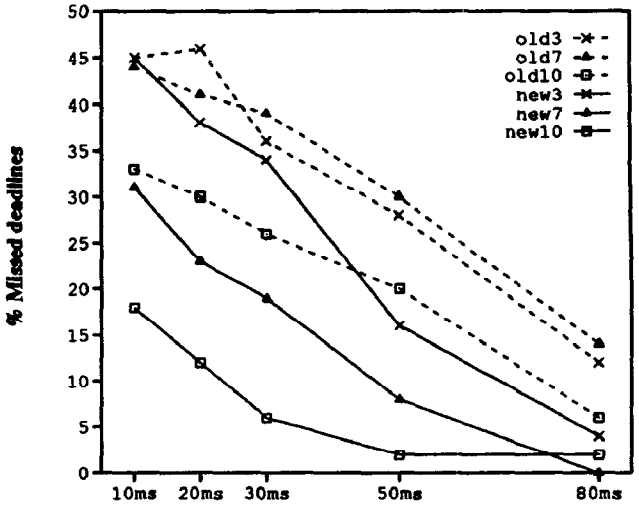

Figure 2: SITES=10, Interarrival time.

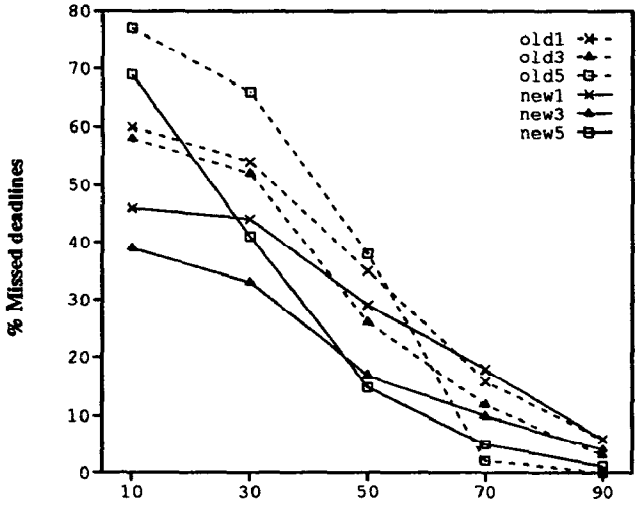

Figure 3: SITES=5, Read only trx.

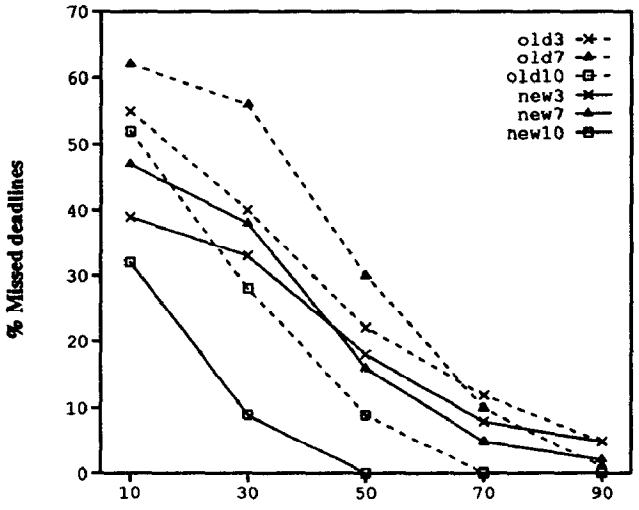

Figure 4: SITES=10, Read only trx.

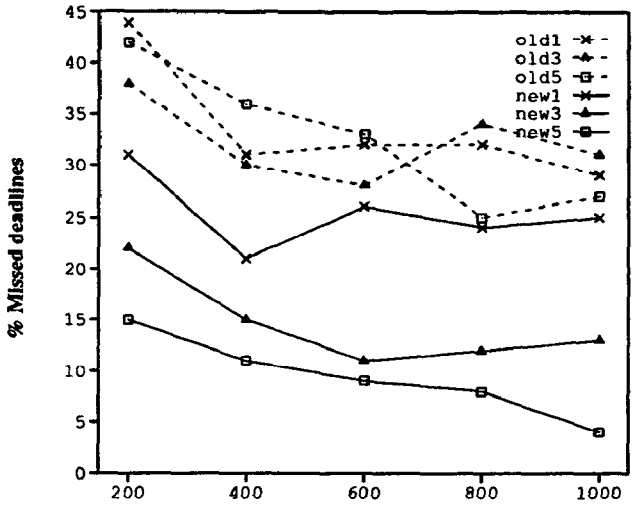

Figure 5: SITES=5, Database size.

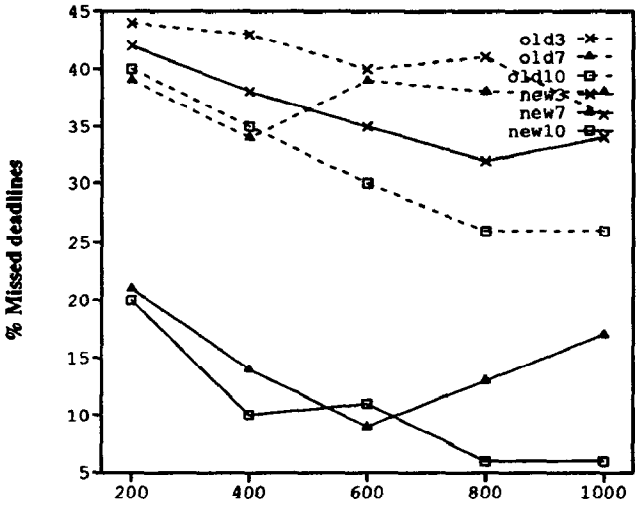

Figure 6: SITES=10, Database size.