Measured Performance of Time Interval Concurrency Control Techniques*

Jerre D. Noe and David B. Wagner
Computer Science Department
University of Washington, Seattle, WA 98195.

*This work was supported in part by National Science Foundation grants DCR-8004111 and DCR-8420945.

Abstract

This paper reports on an implementation of Bayer's Time Interval concurrency control method and compares it to the performance of a conventional timestamp method. The implementation was done on the Eden experimental local area network. Insofar as the authors are aware, this is the first actual implementation of the time interval technique.

The time interval approach clearly is better than time stamping. It provides higher throughput, causes one-third as many distributed transaction aborts, and requires very little additional overhead compared to time stamps.

Within the time interval method we further explored and compared the early and late serialization schemes described by Bayer and his colleagues. Early and late serialialization with time intervals show comparable performance over a range of read/write ratios and multiprogramming levels. In systems that write to disk at the end of all alterations, rather than writing incrementally, late serialization performs better than early serialization because checkpointing to disk can run in parallel with the concurrency control phase.

1 Motivation For This Study

There has been a great deal of interest in the performance of concurrency control algorithms in the literature in recent years [1, 7,9,10,11,13,17,20]. Most of these studies were either simulation-based or analytical in nature, although some used a combination of the two approaches. To the best of our knowledge, there have been no comparative *implementations* of distributed concurrency control methods.

The comparison of actual implementations of concurrency control methods is desirable because modeling and simulation studies generally do not include enough detail to exhibit the effects of finite processing limitations (CPU, disk, and network) on performance. (Three notable exceptions to this are the papers by Agrawal and Carey [1], Carey and Muhanna [7], and Sevcik [20].) Unless the system components are heavily underutilized, this factor should have a noticeable effect on performance whenever a large percentage of the work done by transactions is wasted work, i.e. work which is spent on transaction attempts that ultimately abort. In fact, Agrawal and Carey [1] postulated that contradictory results obtained by different researchers comparing the same algorithms are caused by the inclusion of finite processing resources in the model by some and not by others. Their simulation study included these factors and bore out their hypothesis.

The current study reports measurements on the first known implementation of the Time Interval method proposed by Bayer et. al. [3]. The implementation was carried out on the Eden local-area network, an object-oriented, experimental distributed system [2].

2 Description of the Research

2.1 Choice of Protocols

The Time Interval method was an outgrowth of the RAC protocol [4], which took advantage of the "before" and "after" images used by transaction systems for recovery purposes. RAC was a lock-based protocol that allowed multiple readers, using the old image, even during preparation of the new image by another (single) writer. This meant, of course, that the updating transaction had to be serialized to follow the commitment of the read-only transactions. However, this allowed more concurrency due to the one-writer, multiple-reader compatibility.

It turns out that, in order to guarantee the correctness of the RAC protocol, both the "before" and "after" images of an object may be needed by the system for some period of time, even *after* the new image has been successfully committed[1]. Bayer points out that

[1] The criteria for image deletion are explained briefly in Section 3.2.2; for more details refer to [4].

the protocol can be extended to multiple data versions simply by allowing a subsequent writer to use the previously committed "after" image as its "before" image; similarly, subsequent readers can access the newest committed version of the data that is consistent with serializability requirements. Our protocol implementations contain the necessary extensions to support multi-version data objects.

The RAC protocol was initially formulated using expensive cycle-searching techniques, and was later modified to use much cheaper timestamps or time intervals, which were carried by the lock requests. (The latter two techniques will be summarized briefly in Section 2.2, below.) We chose the Time Interval method as the basis of our work, since it looked like it would provide a significant improvement over ordinary timestamp-based methods and had not heretofore been implemented. Although this method had been simulated by Kiessling and Pfeiffer [13], we felt that their study suffered from the omission of system resource limitations discussed earlier. Also, they compared the use of time intervals combined with *early serialization* to conflict-graph cycle searching combined with *late serialization* and hypothesized that the relative performance difference between them was due to the cycle searching versus time intervals, rather than to early versus late serialization[2]. Primarily we wanted to either validate or refute the usefulness of time intervals, so we chose to compare a time interval based method to one which was completely identical except that it used simple timestamps.

After comparing time intervals with timestamps, the time interval implementation was extended to enable comparison of the early and late serialization alternatives described by Kiessling and Pfeiffer. Since late serialization is in some sense a more "optimistic" method than early serialization, by doing this comparison we were hoping to characterize the tradeoff between increased concurrency and wasted work.

2.2 Fixed Timestamps versus Dynamic Time Intervals

In a fixed timestamp method (hereafter referred to simply as TS) timestamps are chosen for transactions when they begin. Whenever a transaction makes a request that would create a conflict[3] between itself and another transaction, the timestamps of the two transactions are compared. If the order of the timestamps is the same as the serialization order required by the conflict, the request is allowed; otherwise the requesting transaction is aborted and restarted with a new timestamp. Thus, the transaction serialization order is essentially fixed in advance, which has the potential to cause many unnecessary aborts.

Using the Time Intervals method [3] (hereafter referred to as TI) each transaction has two timestamps. These timestamps can be thought of as the upper and lower bounds of an *interval* of timestamp time in which the transaction must appear in the serialization order. Time intervals are partially ordered, with the relations "<" and ">" applying only to intervals that are disjoint (note that non-disjoint intervals can always be truncated in such a way as to impose either ordering on them). Every transaction's initial interval spans the entire allowable timestamp range, representing the fact that there is no restriction on its place in the serialization order until it encounters conflicts with other transactions.

When a conflict is encountered, the time intervals of the transactions involved are compared. If the intervals are disjoint, then their relative ordering has already been established; in this situation the algorithm is exactly the same as for the fixed timestamp case. On the other hand, if the intervals overlap, then they can certainly be truncated so as to effect the desired ordering, after which the request can be granted (possibly involving some blocking by the requestor). In the limiting case, an interval may be shrunk down to a single point, which is then no different in its interpretation than a fixed timestamp. The important distinction is that there are no *a priori* dependencies between transactions, thus there should be fewer unnecessary aborts. The drawback to this method is that extra overhead is required to manage the time intervals, although it will be seen that our measurements show this extra overhead to be small.

2.3 Early Serialization versus Late Serialization

The original RAC protocol specified what has come to be known as *late serialization*, because reader-writer dependencies are not checked until the writer attempts to commit (in contrast to writer-writer dependencies, which are checked when a write lock is requested.) At commit time, a writer's timestamp must be later than the timestamps of all conflicting readers (or, using time intervals, it must be possible to truncate some or all of the intervals to comply with this ordering) or else the writer aborts.

The motivation for late serialization is that read requests are always granted, and read-only transactions are always able to commit. It has as a side effect the possibility that a writer may continue to execute even though another transaction with a later timestamp is reading the same version of the data that the writer is basing its modifications on. This wastes work, since one of the transactions must eventually abort, but if it occurs rarely the increased parallelism will predominate. Note that the latter reason is often cited in support of so-called *optimistic* concurrency control methods [8,14], but most performance studies of these methods are simulations that do not take into account the system resources utilized by the backed-up transactions. Since this is a study of an actual implementation of the protocols, if such effects are present they should be measurable.

On the other hand, under *early serialization*, not only writer-writer but also reader-writer dependencies are checked at the time a read or write lock is requested. For example, if a transaction making a read request at a data object must follow an existing writer at that object, then in contrast to late serialization the would-be reader is forced to wait for the writer to finish.

Using early serialization, a transaction has already met all serialization requirements by the time it reaches the end of its computation. Thus, in the absence of failures, a transaction which attempts to commit is guaranteed to be able to do so. (We do not consider failure atomicity in this study, although failures are considered extensively in [15,16,18]). Early serialization lowers the probability that transaction aborts cause a great deal of wasted work, at the expense of occasionally aborting a transaction that would have eventually committed. Thus, early serialization is more "pessimistic" than late serialization.

[2] Refer to Section 2.3 for a description of the early and late serialization techniques.

[3] E.g. when a transaction requests write access to a data object which is currently being read or written by another transaction. Such a situation indicates a dependency in the execution order of the two transactions. Bayer's use of the term *conflict* is unfortunate, because it suggests a situation in which one of the transactions must be aborted. This is sometimes, but not always, the case. The term *serialization dependency* [5] seems more appropriate.

3 Implementation Strategy

All protocols were implemented on the Eden system [2], an object oriented, capability based, distributed operating system. The Eden kernel provides a location-independent remote procedure call mechanism for inter-object communication. The kernel is currently implemented on a network of 13 Sun workstations running UNIX. Four of the workstations are disk servers, and the remaining nine share the disks over a 10Mb/s ethernet.

The protocol implementation consists of several Eden object types: the MultiVersion ByteStore (abbreviated MVB from now on), the Conflict Manager (CM), and the Transaction Manager (TM).

3.1 The Multiversion ByteStore

The Multiversion ByteStore implements an abstraction of a data object with a version history [19]. The MVB consists of a chain of immutable versions of the data it represents, linked together in order of their times of creation. New versions are created in a three-step process: first, a tentative copy of the (formerly) newest version is made; second, the modifications are done on the copy; and third, the new version is "frozen". Once frozen, a version can never be overwritten; however, a tentative version can be destroyed any time before the freeze operation is done. This allows an aborted transaction to restore an object to its previous state, enabling transactions to appear atomic.

3.2 The Conflict Manager

The CM's job is to monitor conflicts between transactions as they occur. It implements the locking policy of Bayer's RAC with time intervals [3]. The implementation for dynamic time intervals also works for fixed timestamps, simply by having each transaction manager set the upper and lower bound of its time interval to the current time[4] whenever it begins a new transaction. Therefore, we will confine our discussion to the more general time interval case.

3.2.1 Conflict Resolution

Conflicts between two transactions with disjoint time intervals can be resolved immediately, since disjoint time intervals are ordered. When conflicting transactions have non-disjoint time intervals, the status *MustPrecede* or *MustFollow* is returned to the requesting TM, along with information necessary for the TM to carry out the serialization. The requesting TM is responsible for shrinking its own interval, and, if necessary, for negotiating the shrinking of the conflicting TM's interval. (Refer to Section 3.3.2 for more details.) Since time intervals can only be contracted, never expanded, once two intervals become ordered, they will remain in that order despite further adjustments.

3.2.2 Transaction Deletion

In accordance with [3], there are no timestamps stored in the data objects. This is a virtue for two reasons. First, it allows the concurrency control to be applied to arbitrary objects. Second, it cuts down on checkpointing[5]. If timestamps were stored in the data objects, the objects would need to checkpoint whenever a read with a later timestamp than any previous read was done.

A consequence of this is that, even after a transaction T has committed, the CM must retain a record of all locks obtained by T until there are no other transactions remaining in the system that T must follow. T's write locks are changed to *commit locks*, and neither these commit locks nor any of T's read locks may be deleted until the following two conditions have been met:

1. For each object version on which T has a read lock, the commit lock resulting from the creation of that version must already have been deleted.
2. For each object version on which T has a commit lock, all read locks on *all previous versions* of the object must already have been deleted.

These conditions ensure that T has become a *sink* in the global dependency graph [4][6].

Whenever the last lock record is removed from the oldest version of an object, that version can be destroyed, provided that it is not the only committed version of the object. Consequently, these criteria provide a discipline for purging old versions from the system.

A transaction which is committed but still in the CM's data structures because one or both of the above conditions have not been met is called an *inactive* transaction. In Bayer's model, a TM is doing no useful work while its transaction is in the inactive state. We have avoided this waste of resources by relieving the TM of any further responsibility for the inactive transaction. This is accomplished by truncating the transaction's time interval to a point-timestamp at the interval's lower endpoint, thus "pushing the transaction into the past" as far as possible and eliminating the need for any subsequent negotiation. Thus, the TM can begin working on a new transaction as soon as it has committed the old one.

3.3 The Transaction Manager

The TM is responsible for making transactions appear to be atomic. It requests access to the various data objects from the CM, adjusting its time interval as directed by the CM (or in response to requests from other TMs). It handles the creation of new MVB versions, and, depending on the outcome of the transaction, the eventual freezing or deleting of the new versions. It also coordinates the distributed commit.

When a TM finishes one transaction, it immediately begins working on another, using a different unique identifier to avoid confusion.

3.3.1 Transaction Atomicity

In these tests we gathered data only for the cases in which no crashes occurred, but we did include the performance penalty paid by all transactions to provide atomicity: the TM implements

[4] An elaborate, distributed timekeeping mechanism was not needed for the purpose of these experiments, as all times were provided by the clock in the CM.

[5] Changes to an Eden object are not made permanent until the object *checkpoints* itself to stable storage. An atomic checkpoint primitive is provided by the Eden kernel.

[6] We have extended Bayer's original correctness criteria to handle the possibility of multiple committed versions of a data object being in existence at the same time.

a two-phase commit protocol [12]. When the TM successfully reaches the end of a transaction, it needs to checkpoint all MVBs that it has modified. Because checkpoint is a time-consuming operation, all checkpoints are done in parallel.

3.3.2 Negotiation Strategy

Since TMs cannot know what conflicts they will encounter in the future, they cannot adjust their time intervals in an optimal manner. The strategy used by the TMs is a greedy one: operating under the assumption that reducing communication overhead is best whenever possible, a requesting TM will never ask a conflicting TM to truncate its time interval unless the requesting TM is unable to effect the desired ordering by unilaterally truncating its own interval. This is true even when unilateral truncation would result in the requestor's time interval shrinking all the way down to a single point; no attempt is made to obtain a "fair compromise" with the conflicting TM if the communication can be avoided. On the other hand, if negotiation is necessary, the algorithm used attempts to make the serialization "cut" between the two transactions as close to the current real time as possible.

4 System Parameterization and Measurement

4.1 Transaction Workload

The transaction workload was similar to that used by Keissling and Pfeiffer [13]:

- The database contains a "hot-spot" of 100 data objects. Accesses to non-hot-spot data do not encounter serialization conflicts[7].
- Accesses are normally distributed, with 80% of the accesses going to hot-spot objects.
- Transaction size is uniformly distributed between 5 and 15 data accesses.
- Resource access order is random, and locks are obtained incrementally rather than being preclaimed[8].

For each of the three protocols, we varied the percentage of data accesses that required writing (the "writeshare") and the TM multiprogramming level.

4.2 Measures of Interest

Each experiment consisted of running the system until 100 transactions were completed successfully. Therefore, in the discussion that follows the number of *successful transaction attempts* is always 100, but the number of *aborted transaction attempts* could be any number greater than or equal to zero (it could even be greater than 100).

To make our results comparable to [13] we broke down the total amount of time spent processing each transaction into the following components:

- Useful work (data access and manipulation (non-checkpoint) and data checkpointing[9]). This is the amount of time that the transaction would take to execute in the absence of any concurrency control.
- Useful concurrency control overhead. This is the amount of time that a successful transaction attempt spends invoking the CM and negotiating with other TMs.
- Useful blocking. This is the period that a successful transaction attempt must wait for earlier transactions to finish. The conditions for blocking depend upon the protocol being investigated.
- Wasted work, concurrency control overhead, and blocking are analogous to the useful components, except that they comprise the elements of failed transaction attempts.

From these measured quantities we calculate system throughput. Throughput is defined as

$$T = \frac{P_{\max}}{\text{average transaction response time}}$$

where $P_{\max}$ is the (fixed) multiprogramming level, i.e. the maximum attainable parallelism. Transaction response time is the elapsed time used by a TM to successfully execute and commit a transaction, including any time consumed by aborted attempts on the same transaction. The units of T are transactions per second.

5 Discussion of Results

The principal results of this study are presented in Figures 1 through 8. They compare the time interval technique (TI) with a conventional time stamp technique (TS) and, within time intervals, they compare late versus early serialization. In each of the graphs, smooth curves were obtained by quadratic interpolation of the data points.

5.1 Comparison of Timestamps and Time Intervals (Late Serialization)

Using late serialization, we compared performance of the time interval and timestamp concurrency control protocols over a range of multiprogramming levels and proportions of read/write accesses. Throughput, blocking time, number of aborted transaction attempts, and the concurrency control overhead cost were examined.

[7] Keissling and Pfeiffer presumably used this approach to make their simulations run faster. In our case, it obviates the need for having a huge number of separate MVB objects. Accesses by a TM to non-hot-spot data are mapped transparently to a special object called a NULL MVB, the invocation of which results in the same utilization of system resources as the invocation of a "real" MVB. There is one NULL MVB dedicated to each TM, to avoid bottlenecks. Thus, in terms of the load placed on the system, it appears as though the database contains an arbitrarily large number of data items.

[8] Deadlocks are not a problem because the protocols only allow blocking of younger transactions by older ones.

[9] The rationale for measuring Eden-specific quantities such as checkpoint time is to enable us to "factor them out" of the results: we would like to be able to hypothesize about the performance of the protocols on systems with architectures that are significantly different than Eden's.

5.1.1 Throughput

Figure 1 shows the throughput in transactions completed per second as a function of the number of concurrently operating TMs. Throughput curves are plotted for both the TI and TS methods. The effects of increasing the frequency of conflicts are also shown in the figure by plotting pairs of curves for write shares of 10%, 40% and 70%.

The Time Intervals technique is a clear winner; for each pair of curves TI provides higher throughput. Note that the advantage of TI is greatest at low write shares. TI is a more optimistic protocol, in the sense that a given transaction attempt can be expected to survive apparent conflicts that TS would have aborted immediately. For greater write shares, more of these transaction attempts abort anyway, but at a later point than TS would have allowed them to reach. This wastes work. But note that, over the range explored, the increase in throughput due to the greater level of concurrency achieved with TI more than offsets the loss due to wasted work.

At the highest conflict rate, write share = 70%, throughput reaches a maximum in the neighborhood of 5 to 6 concurrent transactions. This maximum could probably be shifted to a higher multiprogramming level with faster processors, even though blocking time, as will be shown in the next section, is the main contributing factor to decreasing throughput and blocking per se does not use processor time. However, faster processors would more quickly finish the transactions during their non-blocked periods, allowing transactions waiting for them to finish to unblock sooner as well.

5.1.2 Blocking Time

Figures 2 and 3 show blocking time for the TI and TS cases, for both the successful and aborted transaction attempts. For the high conflict 70% write cases, the more optimistic TI blocks for 20 to 25% longer than does TS for both the aborted and successful transaction attempts. The TI system is more often waiting, because transactions don't abort so soon. The fact that the throughput for TI is greater, as we saw in Figure 1, means that the gamble is paying off. Another interesting feature shows in figure 3 for the 10% write case: in spite of the pessimism of TS, which causes it to abort transactions earlier, blocking time for those transaction attempts that eventually abort is much longer than for those that succeed. This indicates that with TS, in a low conflict environment, a transaction that blocks more that some time limit – a tuning factor for a given system – should be aborted and restarted, since it will most likely abort anyway if allowed to continue executing.

The greater significance of the blocking times in Figures 2 and 3 appears when comparing them with the average total transaction response times, which do not appear in these two figures. ("Average total transaction response time" means the time for 100 successful transaction attempts, plus the time for the associated aborted attempts, all divided by 100). At the 10% write level, blocking time was only 2% to 5% of response time. At the 70% write level, blocking for successes took 24 to 31% of response time (for multiprogramming levels of 6 to 10) and for the aborts it took 11% to 34% of response time. Blocking time for successes and aborts both affect throughput. Success-blocking contributes directly to delay, whereas abort-blocking increases the time a transaction manager spends on "doomed" transactions and thus decreases effective parallelism. This effect is in addition to the delay of successful transactions due to the wasted system utilization on behalf of the transaction instances that ultimately abort.

5.1.3 Number of Aborts

The number of aborts was measured by running the system until 100 successful transactions were completed and counting the number of transaction *attempts* that resulted in aborts; thus, in extreme cases the number of aborts could be greater than 100. Figure 4 compares the number of aborts for TI and TS for write shares of 10%, 40%, and 70%. TS caused between two and three times the number of aborts compared to TI. Separate measurements, not indicated in this figure, show that the amount of work wasted per aborted transaction attempt under TS was equal to or greater than the work wasted using TI. The resulting increase in wasted work contributed to lower throughput both directly and indirectly, the latter by increasing the blocking time of other transactions which had to wait for the aborting ones.

5.1.4 Overhead Cost of Time Intervals

There is very little additional concurrency control overhead in using time intervals compared to using timestamps as is shown in Figure 5. The overhead is the time spent in the conflict manager and, in the TI case, the additional time required for negotiation among transaction managers to alter the time intervals. This overhead time is plotted as a percentage of the total average transaction response time.

The results strongly suggest a discontinuity between multi-programming levels 1 and 2 (which seems quite reasonable), and for this reason these values were not included in the curve interpolations. Note that the actual overhead time in seconds increases as the multi-programming level increases beyond 2, but not as quickly as blocking time, hence the overhead is a smaller percentage of the total for higher levels of concurrency.

5.2 Comparison of Early and Late Serialization Using Time Intervals

Within the Time Intervals method we further implemented and compared the early and late serialization schemes described by Kiessling and Pfeiffer [13]. Early serialization (ES) allows a single writer and multiple readers with read/write and write/write synchronization being done at request time. Late serialization (LS) does write/write synchronization at request time and read/write synchronization at commit time. It lies closer to optimistic concurrency control [14] except that it allows only one writer at a time.

5.2.1 Throughput

Figure 6 compares the throughput for ES and LS for two different values of write share. Particularly in the high conflict case (70% writes) LS provides greater throughput. However, in both cases LS throughput decreases much faster than ES throughput as the multiprogramming level increases. This is because, on the average, LS allows "doomed" transactions to run longer than ES does. The performance of the ES protocol is therefore affected much less by increasing concurrency.

When the ES, 70% writeshare curve was first plotted, throughput showed a very large dip at multi-programming levels 7 and 8. Subsequent analysis of the raw data revealed three highly suspicious samples that also happened to deviate significantly from their respective means. In our judgement, these samples are the result of errors in the experimental testbed system. Since their inclusion in the sample means obscures any trend in the interpolated curve, we have excluded them from our calculations. However, we have marked their locations on the graph with question marks, so that the reader may draw his or her own conclusions.

5.2.2 Throughput Without Overlapped Write to Disk

The throughput figures of the previous section are somewhat biased in favor of LS, because our implementation was on a system that does not allow incremental writing to disk[10]. Checkpoints are done on a complete re-write basis after all changes to the data have been made. This corresponds to a system creating shadow copies for crash resistance during commit rather than creating them incrementally during the transaction. Since LS does much of its concurrency control work at the end of the transaction, checkpointing can be overlapped with the serialization checking. Of course, this concurrency will waste system resources (in the form of kernel activity and disk accesses used for checkpointing) if the outcome of the concurrency control phase results in an abort. However, it saves elapsed time in the case of a successful transaction. A comparable savings cannot be made in the ES case because all of the concurrency control work is done before the end of the transaction.

Although it was not possible to conduct experiments in which the disk writing took place incrementally during the transaction, we did change the implementation of LS so that checkpointing did not begin until the completion of all serialization checking. This, at least, gave a comparison of ES and LS on the same basis although it did not give throughput figures as high as would have been obtained if the incremental writing had really been possible. Figure 7 adds to Figure 6 the throughput of LS with all checkpointing following the concurrency control phase. It appears that, within the limits of experimental error, the two protocols give similar performance at a writeshare of 70%, and that ES is now superior in the 40% writeshare case. One can see by comparing Figure 7 with Figure 6 that overlapped checkpoint causes a substantial increase in throughput compared to non-overlapped checkpoint.

5.2.3 Wasted Work Due to Aborts

ES, through its pessimistic approach to conflicts, causes as many as three times the number of aborts as LS. However the fact that these aborts are made relatively early in the transaction means that less work is wasted before the abort decision is made. The product of the number of aborts and the wasted work per abort is therefore of interest and this is shown in Figure 8. The product looks roughly comparable for the two methods at the 70% writeshare, although LS appears to waste more work at the 40% percent write share. However, the reader should again note that several samples from the the ES, 70% writeshare curve at higher multi-programming levels were dropped from the sample mean calculations. Nonetheless the main point is clear: the two systems do not differ greatly in the amount of work wasted at high write shares.

[10] A feature that has frequently been recognized as a bad design decision; see, for example, [6].

6 Summary

This first experimental implementation of Bayer's Dynamic Time Interval method for concurrency control provided the opportunity to compare time interval serialization options with timestamps.

Time intervals clearly perform better than timestamps, giving greater throughput, a smaller number of aborts and less wasted work due to aborts. This was accomplished with very little increase in concurrency control overhead compared to timestamps. This advantage decreases when the workload contains a higher proportion of updates, since the more optimistic TI method loses more of its "gambles" and must finally abort transactions that were executing concurrently for a period longer than TS would have allowed. However, over the range explored in these experiments, the increase in concurrency with TI more than offset the wasted work, and throughput was increased.

Blocking time is the principal cause of decreasing throughput as the multi-programming level increases and, of course, becomes worse for a high proportion of write requests. In particular, using TS in a low-conflict environment, a transaction that blocks longer than a certain threshold is most likely to abort, so it may pay to re-start the transaction after some time-out interval that is a tunable parameter of the system.

Within the Time Interval method, early and late serialization appear comparable in throughput and wasted work. LS is better if the entire updated object must be written at the end of the transaction because this write to disk can overlap a substantial portion of the concurrency control time. This cannot be done with ES since its concurrency control must take place before the alterations. If this overlap advantage is removed from LS the two appear comparable. This work, in general, confirms the speculation in the paper by Kiessling and Pfeiffer [13] that ES and LS would give similar performance.

The implementation was done on the Eden experimental local area network which proved to be a useful testbed to examine and compare these concurrency control methods.

Acknowledgments

The authors would like to express their appreciation for the efforts of Gita Ghopal (Bell Communications Research) and Edgar Nett (GMD Birlinghoven), whose careful reading and constructive criticisms of earlier drafts helped to improve the clarity and technical content of this paper. We would also like to acknowledge the many graduate students and faculty at the University of Washington who contributed to the Eden project; they are far too numerous to name individually.

References

[1] R. Agrawal and M.J. Carey. The performance of concurrency control and recovery algorithms for transaction-oriented database systems. *IEEE Database Engineering*, 58–67, June 1985.

[2] G.T. Almes, A.P. Black, E.D. Lazowska, and J.D. Noe. The Eden system: a technical review. *IEEE Transactions on Software Engineering*, SE-11(1):43–58, January 1985.

[3] R. Bayer, K. Elhardt, J. Heigert, and A. Reiser. Dynamic timestamp allocation for transactions in database systems. In H. J. Schneider, editor, *Distributed Data Bases*, North-Holland, 1982.

[4] R. Bayer, H. Heller, and A. Reiser. Parallelism and recovery in database systems. *ACM Transactions on Database Systems*, 5(2):139–156, June 1980. See also *Distributed Concurrency Control in Database Systems* by Bayer, Elhardt, Heller and Reiser, in the Proceedings of 6th Int. Conf. on Very Large Data Bases.

[5] P.A. Bernstein and N. Goodman. Multiversion concurrency control — theory and algorithms. *ACM Transactions on Database Systems*, 8(4):465–483, December 1983.

[6] A.P. Black. Supporting distributed applications: experience with Eden. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 181–193, ACM/SIGOPS, December 1985.

[7] M.J. Carey and W.A. Muhanna. The performance of multiversion concurrency control algorithms. *ACM Transactions on Computer Systems*, 4(4):338–378, November 1986.

[8] S. Ceri and S. Owicki. On the use of optimistic methods for concurrency control in distributed databases. In *Proceedings of the 6th Berkeley Workshop on Distributed Data Management and Computer Networks*, pages 117–129, Lawrence Berkeley Laboratory, University of California, Berkeley, February 1982.

[9] R. Cordon and H. Garcia-Molina. The performance of a concurrency control mechanism that exploits semantic knowledge. In *Proceedings of the 5th Conference on Distributed Computer Systems*, pages 350–358, IEEE Computer Society, March 1985.

[10] P. Franaszek and J.T. Robinson. Limitations of concurrency in transaction processing. *ACM Transactions on Database Systems*, 10(1):1–28, March 1985.

[11] H. Garcia-Molina. *Performance of Update Algorithms for Replicated Data in a Distributed Database*. PhD thesis, Department of Computer Science, Stanford University, June 1979. Technical report STAN-CS-79-744.

[12] J.N. Gray. Notes on data base operating systems. In *Operating Systems - An Advanced Course*, Springer-Verlag, 1978. Also IBM Research Report RJ 2188, Feb. 1978.

[13] W. Kiessling and H. Pfeiffer. A comprehensive analysis of concurrency control performance for centralized databases. In *Fourth International Workshop on Database Machines*, Springer-Verlag, 1985.

[14] H. T. Kung and J.T. Robinson. On optimistic methods for concurrency control. *ACM Transactions on Database Systems*, 6(2):213–226, June 1981.

[15] J.D. Noe and Agnes Andreassian. *Effectiveness of Replication in Distributed Computer Networks*. Technical Report 86-06-05, Department of Computer Science, University of Washington, June 1986.

[16] J.D. Noe, A. Proudfoot, and C. Pu. Replication in distributed systems: the Eden experience. In *Proceedings of the FJCC, Dallas, Texas*, November 1986.

[17] M.T. Oszu. Performance comparison of distributed vs. centralized locking algorithms in distributed database systems. In *Proceedings of 5th Distributed Computing Systems Conference*, pages 254–261, IEEE, 1985.

[18] C. Pu, J.D. Noe, and A. Proudfoot. Regeneration of replicated objects: a technique and its Eden implementation. In *Proceedings of the Second International Conference on Data Engineering*, February 1986.

[19] D.P. Reed. *Naming and Synchronization in a Decentralized Computer System*. PhD thesis, Massachusetts Institute of Technology, September 1978.

[20] K.C. Sevcik. Comparison of concurrency control methods using analytic models. In C. Mohan, editor, *Tutorial: Recent Advances in Distributed Database Management*, IEEE Computer Society Press, December 1984.

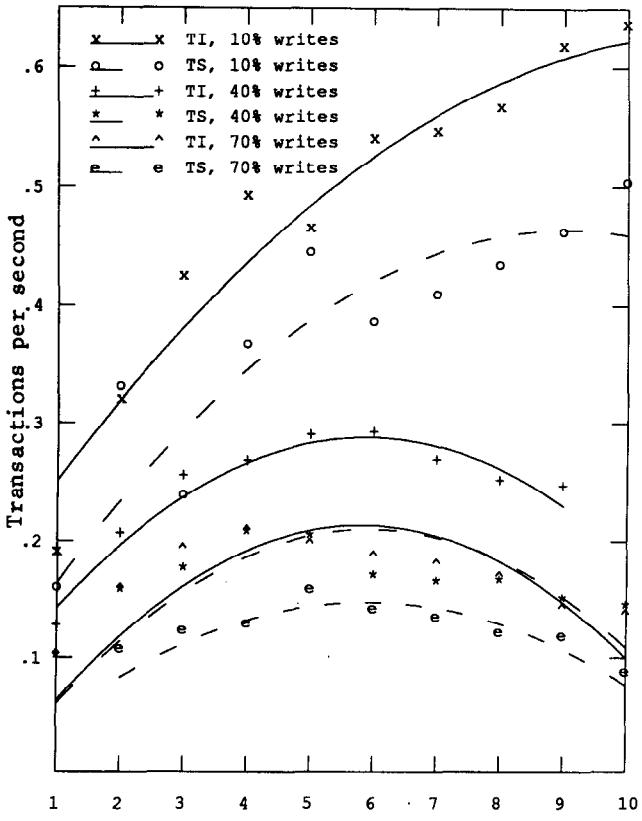

Figure 1. Throughput vs MPL (TI, TS)

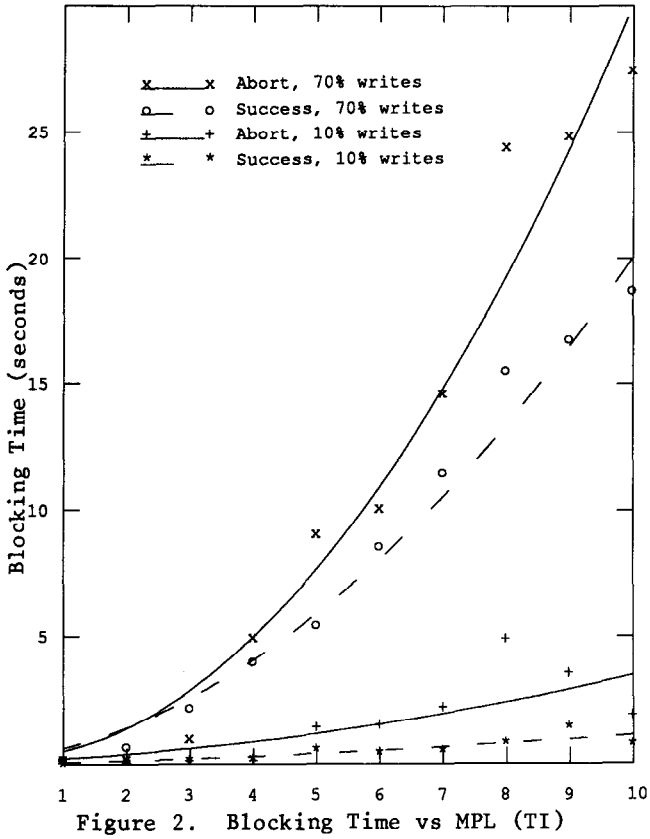

Figure 2. Blocking Time vs MPL (TI)

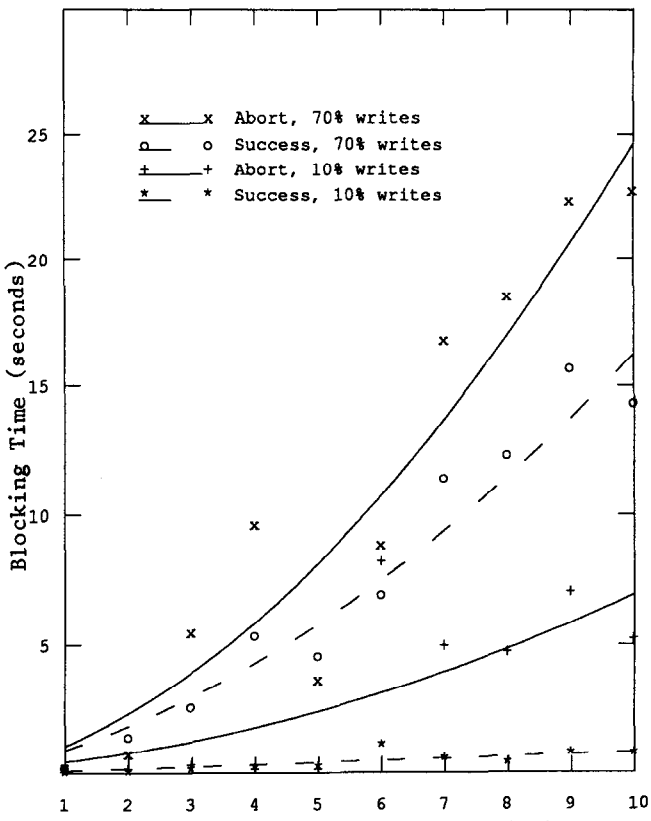

Figure 3. Blocking Time vs MPL (TS)

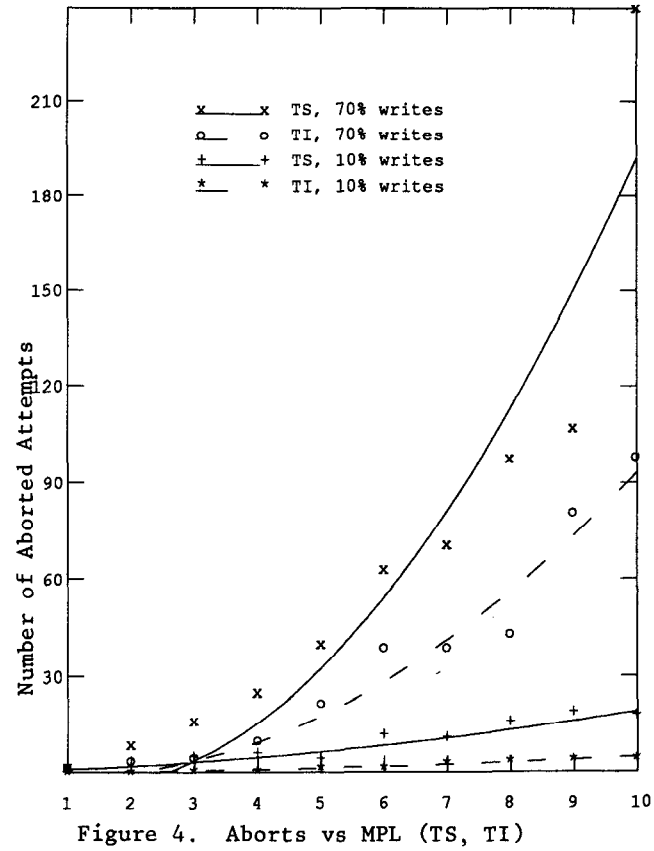

Figure 4. Aborts vs MPL (TS, TI)

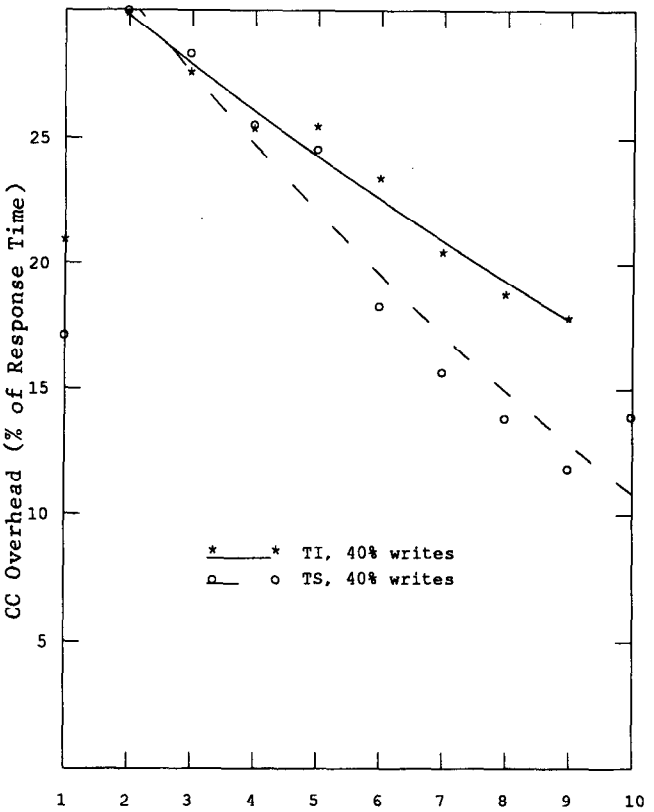

Figure 5. CC Overhead vs MPL (TI, TS)

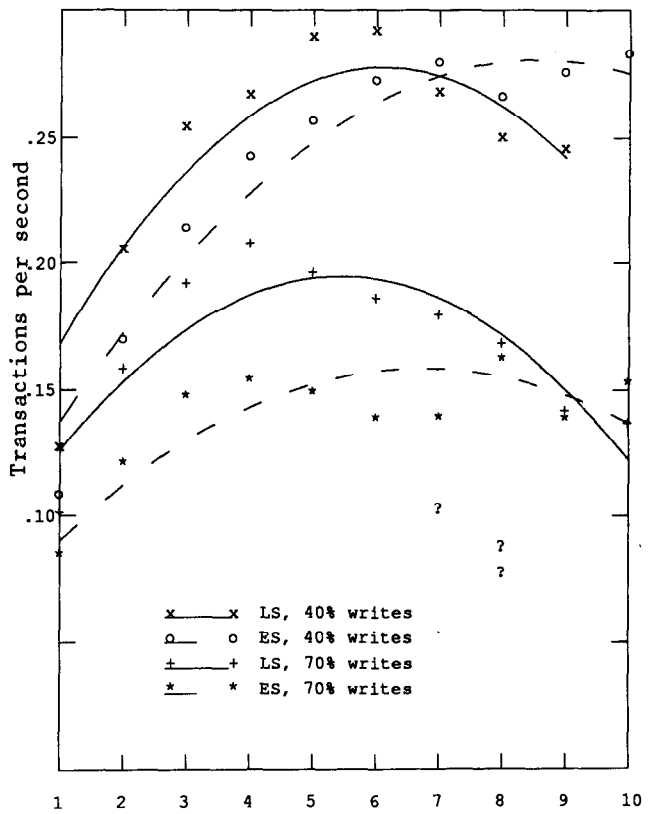

Figure 6. Throughput vs MPL (ES, LS)

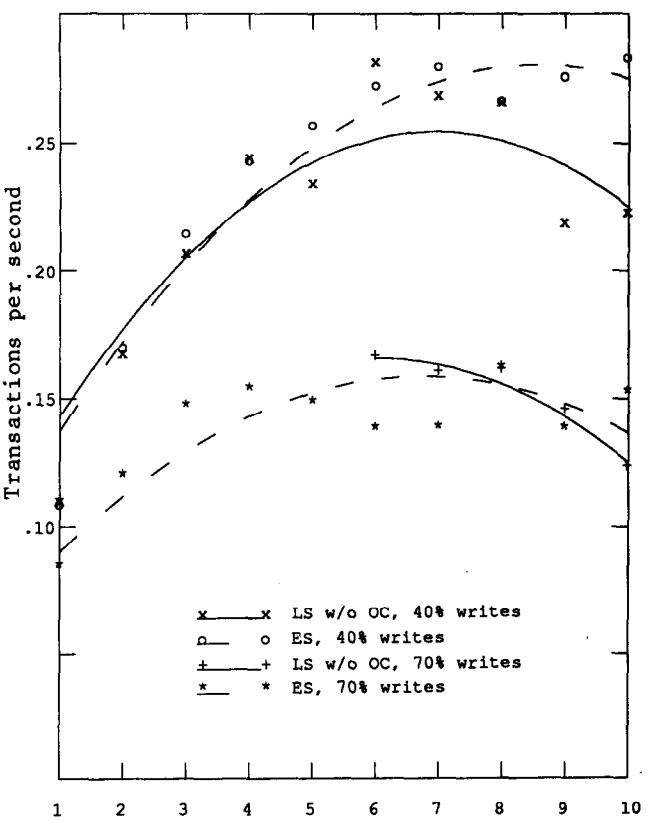

Figure 7. Throughput w/o Overlapped Chkpt.

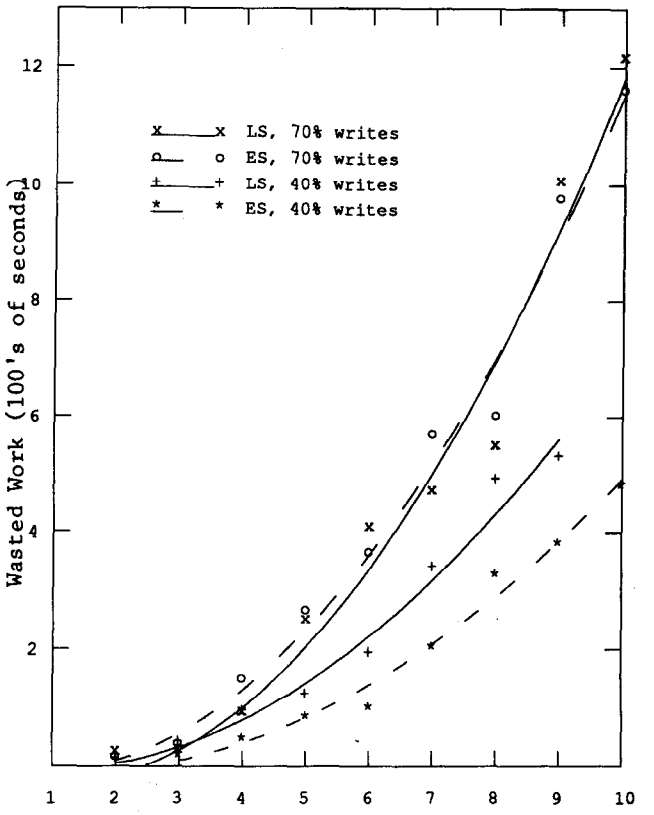

Figure 8. Wasted Work vs MPL (ES, LS)