MULTIPLE DATA VERSIONS IN DATABASE SYSTEMS

Navid Haghighi
Sang H. Son

Computer Science Report No. TR-89-01
June 5, 1989

Multiple Data Versions in Database Systems

Navid Haghighi
Sang H. Son
Department of Computer Science
University of Virginia
Charlottesville, Virginia 22903

ABSTRACT

A prototyping environment is a valuable tool for evaluating new techniques and design alternatives for database systems prior to the actual implementation of the system. Using such a prototyping tool, the characteristics and possible performance benefits of maintaining multiple data versions with time-stamp based scheduling are investigated. In addition, the characteristics of a database system employing a multiversion concurrency control mechanism in a real-time environment are explored. This research is focused on how the availability of multiple data versions may be used in order to better meet the timing constraints of the system. A scheduling algorithm which incorporates the consideration of the temporal requirements in its scheduling decisions is presented.

TABLE OF CONTENTS

1. Introduction

A prototyping environment is a valuable tool for evaluating new techniques and design alternatives for database systems prior to the actual implementation of the system. Through the use of the prototype, the designers are not required to spend considerable time and effort building new systems before the design of their integrated functional components are evaluated and justified. This paper describes the experiments conducted and the results which were observed while using such a database prototyping environment. The prototyping environment was developed based on a flexible message-passing concurrent programming kernel. A full description of its functional components may be found in [1].

The focus of this paper is to investigate, using the prototyping environment, the characteristics and possible performance benefits of a multiversion database system with time stamp based scheduling. A number of researchers have studied the nature and performance of multiversion time-stamp algorithms [2,3,4,5]. Although these studies provide a valuable insight in regards to the particular approach adopted in undertaking the evaluations, it is almost impossible to compare or integrate their results. This is due to the fact that they each have made different assumptions about the environment and the system, and they have often used widely varying performance metrics. In our model, the assumptions which significantly influence the system behavior may be specified as parameters during the evaluation process. Therefore we not only have the capability of investigating the influence of these assumptions on system performance, we can also study the other major performance factors while retaining the assumptions at a constant.

The organization of the paper is as follows. Section 2 outlines a series of four experiments that were conducted in order to compare the relative performances of the single version and multiversion systems under varying system parameters. Each experiment concentrates on the effect of one major parameter. Section 3 describes the experiments that were conducted in order to investigate the nature of two different strategies employed in implementing the multiversion system. In each experiment, a comparison of performance results is presented. Section 4 presents the results of our study on the performance benefits of the multiversion system in a real-time environment. Section 5 is a summary of our conclusions. Section 6 is a list of references.

2. Multiversion Vs. Single Version Concurrency Control

The primary reason for implementing multiversion concurrency control algorithms is to allow the use of previous versions of data items in order to improve the level of achievable concurrency [3]. Each update (or write) transaction creates a new version of the data item. In a time-stamp based scheduling system such as ours, each read transaction selects the appropriate version of the data item to be read from amongst a collection of available versions. This selection is based on how closely the time-stamp of the data version matches that of the transaction without surpassing it. In a single version algorithm, a conflict between a read and a write transaction requesting to access the same data item is problematic and usually results in the abortion (or delay) of one of the transactions in order to solve the conflict. The motivation for utilizing multiversion schemes that intuitively result in a performance improvement is to allow read transactions to access older versions of the data while the update transactions concurrently create new versions of the same data items. This would alleviate the need for aborting or delaying either one of the transactions.

A variety of multiversion algorithms have been studied. These approaches vary on how the creation, maintenance, and accessing of data versions are treated. While some of the algorithms allow for only one older version of any data item to be maintained [6,7], others allow the creation of a certain number or an unlimited number of data versions [8,9,10]. Some algorithms allow only one transaction to create a new version at a time, while others allow for simultaneous creation of new versions of the same data item by different transactions. The performance trade-offs resulting from different algorithms incorporated in multiversion database systems is the subject of Section 3. In this section we will present a comparison study of the relative performance of the single and multiversion concurrency control algorithms which were implemented using our prototyping environment.

The user is given a chance to enter values for the key parameters that influence the performance of the simulator before each run. According to the user specifications, processors (or sites) in the distributed environment are initialized. Then transactions are generated as processes. Each transaction contains a number of read and/or write requests depending on the transaction mix and size specified by the user. Once a transaction has been created, it will successively access the data items in its read/write set until it either aborts or commits. If no conflicts have occurred by the time a transaction has successfully accessed all its data items, the transaction commits. An abortion due to unresolved conflict causes the transaction to be restarted at a later time with the same read/write set. The total time from when a transaction was created until it commits is computed. The key performance metric used is the average response time of all the transactions. The system also reports the total number of abortions during each simulation.

In the multiversion database, each version of a data item contains a read and a write time stamp associated with the transaction that created the version. Each data version also has a tag variable associated with it. A tag value of 0 indicates that the transaction which generated the data version has not yet committed. The data version is considered temporary and thus can not be used by other read or write transactions. Once a transaction commits, all the tag values associated with the data versions it created are changed to 1. These data versions are now accessible to other transactions.

A read request in the multiversion system involves the comparison of the time stamp of the requesting transaction with the write time stamp of the data item and its different versions. The comparison begins with the base version and continues until the data version with the largest write time stamp which is less than the time stamp of the transaction is found. In Fig 1, the read request with time stamp 14 will read the base version with a write time stamp value of 12.

The operation of a write request in the multiversion database involves the comparison of the time stamp of the transaction with the read time stamp of the data item. If the transaction's time stamp is greater than the read time stamp of the data item, a new version is created with a tag value of 0 to indicate that it is not yet valid until the transaction commits. Fig 2 illustrates a write operation. A transaction with time stamp value of 19 has successfully created a new version for data item dset[o].

The multiversion database system currently under study has very strong constraints in regards to the creation of new versions for data items. At any given time, only one temporary version for each data item is allowed. An attempt to create a new version on top of a temporary version results in the abortion of the transaction. Also new versions are only allowed to be added at the end of the queue of different versions of a data item. An attempt to insert a new version between two already existing versions causes an abortion. In Fig 3 the transaction with time stamp 25 is aborted since there already exists a temporary version of the data item with write time stamp 19. Also the write transaction with time stamp value 8 is aborted since it requires the insertion of a new version between two valid versions. In Section 3 we will examine the consequences of removing these constraints.

The four experiments that are presented in this section were conducted using the multiversion database model described above. In each experiment the performance of the multiversion system was compared to that of a database system implementing a basic time stamp ordering scheduling algorithm with a single

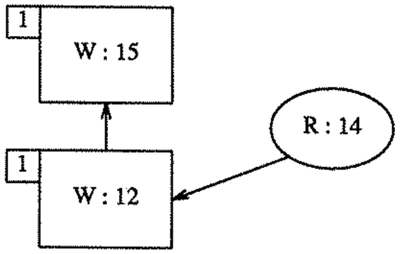


Fig 1

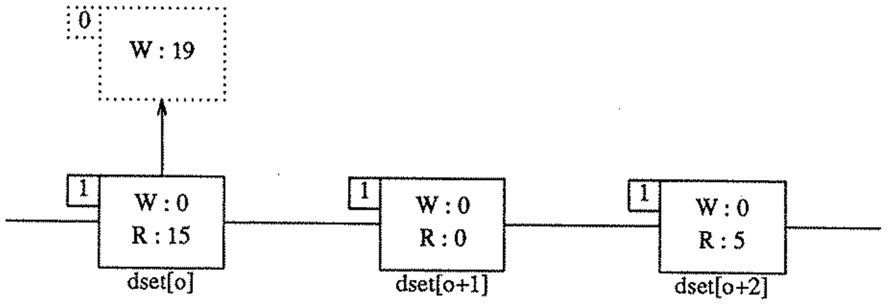


Fig 2

version of every data item. In each case, both systems ran with the same set of input transactions.

2.1 Experiment 1: Parameters in Favor of Multiversion System

The purpose of this experiment is to examine the behavior of the multi-version database system under a mix of transactions for which intuitively it was thought that having multiple versions would be beneficial. The performance of the multi-version system is compared with the corresponding single-version system under the same set of transactions.

Intuitively, the multiple versions of data objects can help the scheduler to avoid the abortion of read-only transactions that arrive too late. Since multiple versions are maintained, tardy read-only transactions can access earlier versions of the data item whose write time-stamp value is less than the current transaction's time-stamp. In the single version system, if the time-stamp of a read transaction is less than the write time-stamp of the data item, the transaction is aborted.

The following 3 cases illustrate the conflicts that may arise when processing read-only transactions:

1. The time-stamp of the transaction is less than the write time-stamp of the base version.

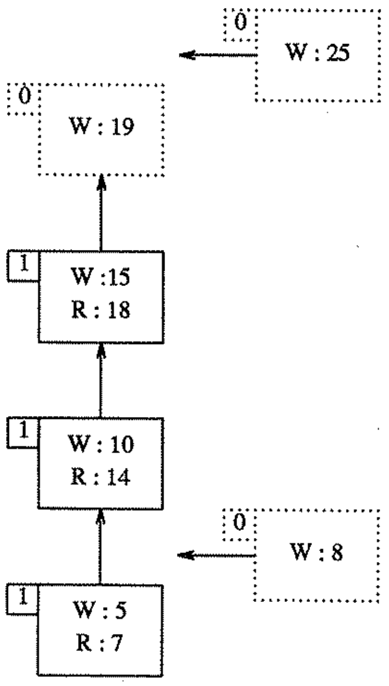


Fig 3

Since the write time-stamp of the base version is less than that of any other versions that may be available, there are no earlier versions that may be used to satisfy the current transaction. The transaction is aborted. The single version system would have similarly aborted the transaction.

2. The base version of the data item is the only one available and its write time-stamp is less than the transaction's time-stamp. However, there is an active (non-committed) write transaction operating on this data item whose time-stamp is greater than the read transaction's time-stamp. In a single version system, the read transaction would be aborted. However, the multi-version system allows the read transaction to proceed. The active write transaction will create a new version of the data item, so there is no problem with reading the base version.

3. There are multiple versions of the data item. The time-stamp of the transaction is less than the write time-stamp of the latest version. In a single version system, a similar situation would cause the transaction to be aborted. However in a multi-version system, the transaction may be satisfied by an earlier version whose write time-stamp is less than the transaction's time-stamp.

The transaction mix emphasizes conflicts involving read-only transactions. In each case, 80% of the transactions are update transactions that read and write the data items, and the remaining 20% are read-only transactions. Update transactions are very small. They always read and write 2 data items. On the other hand, read-only transactions are large, ranging in size from 8 to 40. Conflicts between update transactions are possible but not very likely. Conflicts between read-only and update transactions occur quite frequently. The multi-version system should avoid these conflicts by allowing read-only transactions to read earlier versions of the data items[3]. The experiment is conducted in a centralized environment with only one site (processor). The average response time and the number of abortions are examined as we increase the number of transactions present in the system.

The average response times of both systems under the parameter settings described above are illustrated in the graph of Fig 4. Fig 5 is a graph of the number of abortions resulting from the same simulation. None of the read-only transactions were aborted in the multi-version system. The aborted update transactions were equally distributed over the following two cases:

1. The base version is the only version and its read time stamp is less than the current transaction's time stamp. We will call this transaction T1. We should be able to allow T1 to create a new version. However, there is at least one other active transaction, T2, with a time stamp greater than that of T1. In addition, T2 is currently reading the same data version. Consequently, T1 is

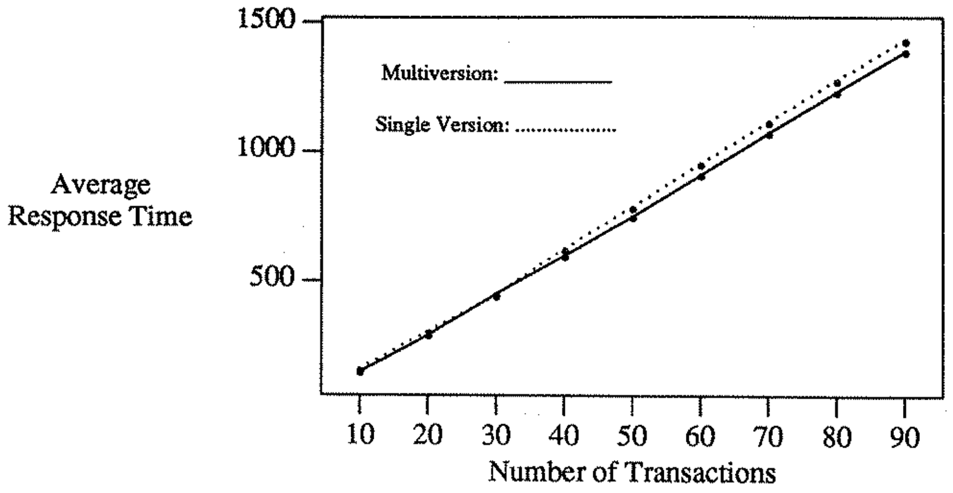


Fig 4

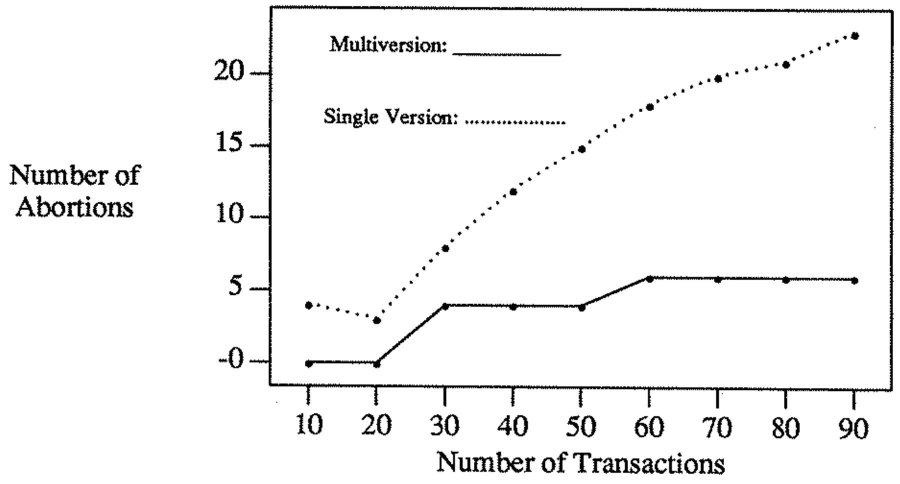


Fig 5

aborted. A similar situation in the single version system would also cause an abortion. The following explains the logic behind why in the multi-version system we abort the update transaction:

A. Consider the case where T2 is an update transaction. If we don't abort T1, we are effectively pursuing the creation of two new versions of the same data item at the same time. In the current implementation, we have made the choice to allow only one temporary version at a time. Therefore, the second update transaction must be aborted.

B. In the case when T2 is a read-only transaction, If we allow T1 to proceed, then T2 may read an older version when it should have really read the new version T1 is currently creating.

2. There are multiple versions. However, the read time-stamp of the very last version is greater than the current transaction's time-stamp. Clearly, we can not create a new version after the last version. Since the creation of a new version between two already existing versions is not allowed in this implementation, we must abort the current transaction.

Clearly the graph of figure 5 shows that the multiversion system provides a noticeable improvement in regards to the number of abortions. Every time a transaction is aborted, we effectively lose the work that transaction has already accomplished, and in addition, that transaction must be restarted later. Since the read-only transactions are large, this could amount to considerable wasted processing time. As mentioned earlier, we expect the multiversion system to perform better in terms of lowering the average response time by avoiding the abortion of read-only transactions. Therefore, it would be of interest to examine the effect of the multiversion system on the average response time of the read-only transactions. In order to accomplish this, we must isolate the read-only transactions from the update transactions. This way, the behavior of the update transactions will not affect the average response times of the read-only transactions. The graphs in Fig 6 and Fig 7 show the average response times and the number of abortions of the read-only transactions in isolation.

Single version time stamp ordering is biased against large read-only transactions[3]. The algorithm aborts a transaction every time it tries to read a data object whose write time-stamp is greater than the transaction's time-stamp. As the read-only transactions get larger, the possibility that the transactions get aborted increases. With this experiment's transaction mix, these sort of conflicts are very likely. In the single-version system the large read-only transactions often get starved out (restarted over and over) as the small update transactions proceed with their computations. These abortions result in high average

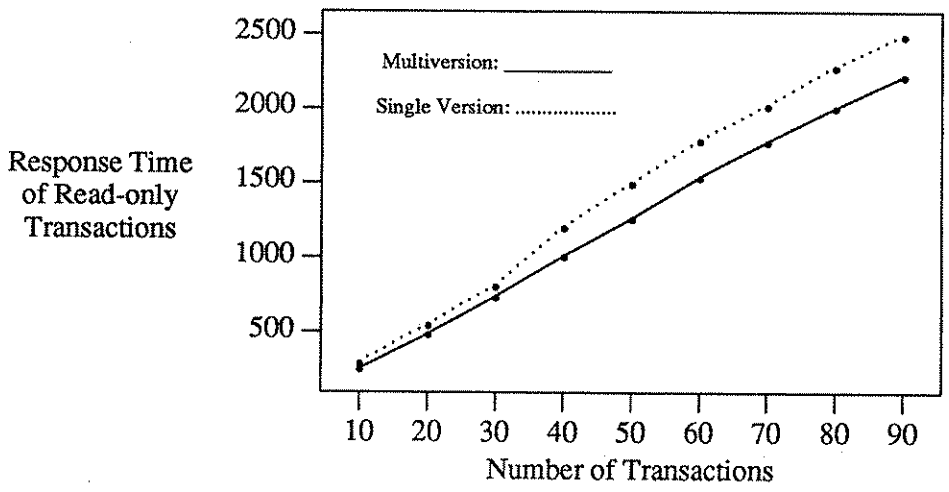


Fig 6

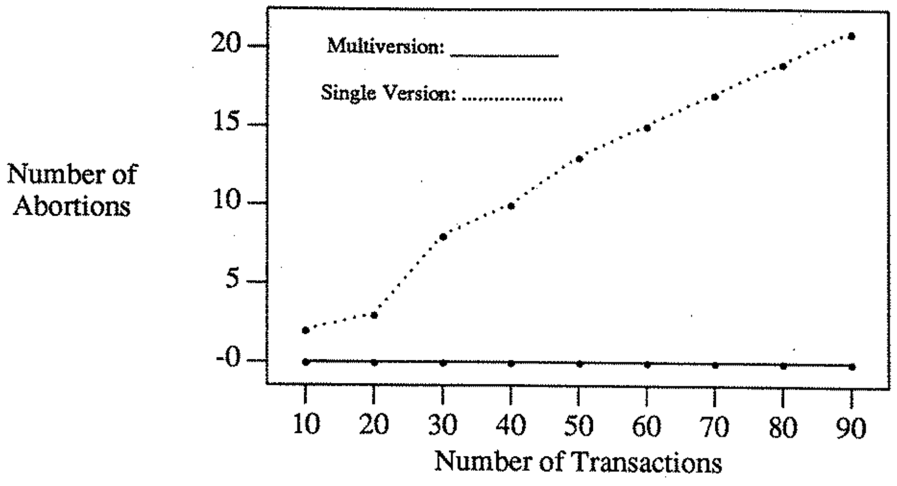


Fig 7

response times. The multi-version system avoids this problem by satisfying read requests with older versions. Therefore the average response times are lower. The multiversion concurrency control algorithm results in considerable performance improvement when the transaction mix consists of large read-only transactions and small update transactions.

2.2 Experiment 2.2: Effect of Transaction Mix

The purpose of this experiment is to test the effect of the transaction mix (update/read-only ratio) on the performance of the single and multiversion systems. In experiment 1, we evaluated the performance of the systems under a transaction load mix of 80% update and 20% read-only transactions. This experiment analyzes the following update/read-only ratios:

- 20% update, 80% read-only

- 100% update

- 100% read-only

- 50% update, 50% read-only

Throughout this experiment, the transaction size is held constant at 5. This means that all transactions read or write 5 data items. The simulations were conducted in a centralized environment with only one processor. What follows is a description of the outcome of these simulations.

20% Update, 80% Read-Only Transactions

The graph which demonstrates the results of the simulation under this system configuration appears in Fig 8. Since there is never a conflict between two read-only transactions and read-only transactions dominate the transaction mix, the potential for read-only/update conflicts is small to begin with. This probability is further reduced due to the fact that each transaction accesses a small number of data items(5). Since the read-only transactions rarely get aborted in the single version system, there is little hope that the existence of multiple versions would help in preventing abortions. The graph indicates that multiple versions do not present a performance benefit under these conditions.

These results are further explained by the fact that both systems favor read-only transactions. An active read-only transaction is aborted only if it conflicts with a completed update transaction that has a later time-stamp [2]. This only occurs rarely because of two reasons. First, update transactions take longer to complete. Secondly, update transactions only make up 20% of the transaction mix. Therefore, there is rare opportunity for a read-only transaction to conflict with a completed update transaction. As a result, multiple versions are of little value. We may conclude that when transactions are uniformly small and read-only transactions dominate in the mix, multiple versions do not provide a performance benefit.

50% Update, 50% Read-Only Transactions

The graph of the average response times observed in this simulation appears in figure 9. Again, due to the small size of transactions, there is not a significant number of read-only transaction abortions that the multiple versions can help to avoid. Therefore, the performances of the single and multiversion systems

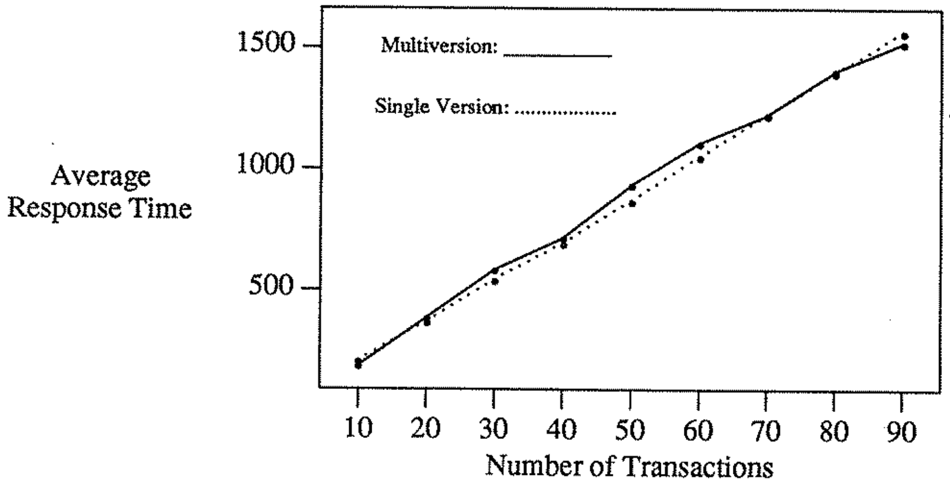


Fig 8

are almost identical.

100% Read-Only Transactions

The graph appears in Fig 10 . As seen, the multiversion and single version systems show identical performances. This observation is easily explained by the fact that with 100% read-only transactions there are no write operations that would allow the multiversion system to create additional versions of data items. Therefore, with a transaction load consisting of 100% read-only transactions the multiversion system effectively operates in the same manner as its single version counterpart.

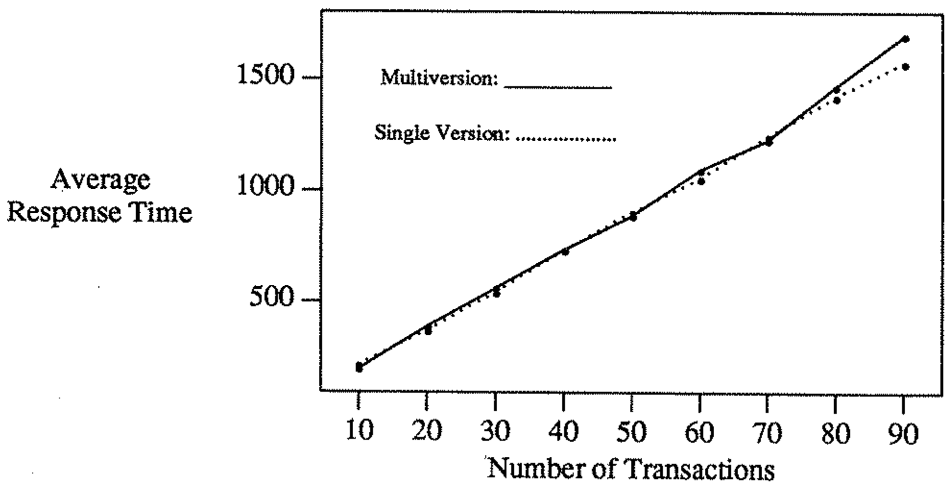


Fig 9

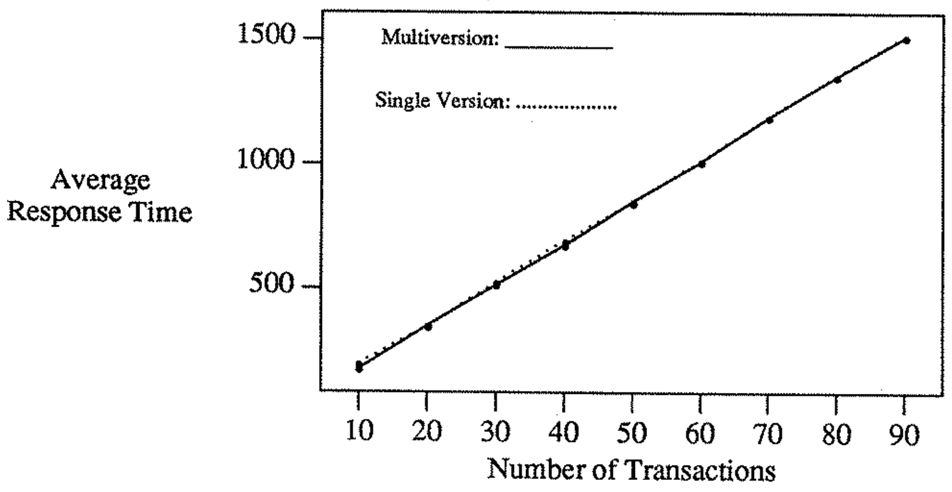


Fig 10

100% Update Transactions

The graph is shown in Fig 11. Since there are no read-only transactions, and therefore no abortions involving them, we would not expect the multiversion system to outperform its single version counterpart. As a matter of fact, as shown in the graph, the single version system performs better. This difference in the performances becomes more noticeable as we increase the number of transactions. The difference is due to the fact that the implementation of the multiversion system is prejudiced against update transactions. This is caused by the restrictions that we have employed regarding the creation of new versions of data items in the multiversion system. As a result, the number of update transactions that are aborted due to write-write conflicts is higher. These implementation restrictions and the consequences of removing them are further discussed in sections 2.5 and 3.1.

From the results of these simulations, we conclude that regardless of the transaction mix, multiple versions do not buy us much (if anything at all) when the transaction sizes are uniformly small.

2.2 Experiment 2.3: Effect of Transaction Size

This experiment investigates the behavior of the single and multiversion systems under varying transaction sizes. The purpose is to examine whether transaction size has a noticeable affect on the difference of the performance of the single and multiversion systems. We survey the performance of both systems while varying transaction sizes under the following three different transaction mixes:

- 80% update, 20% read-only

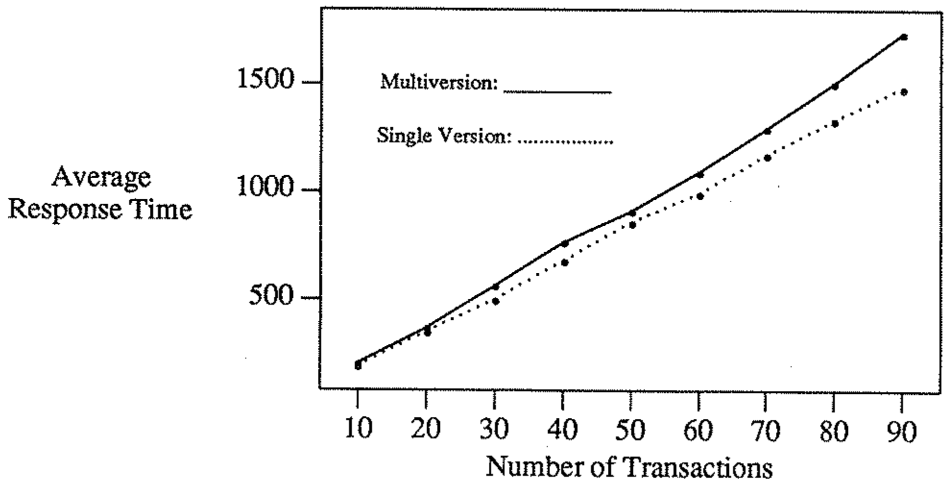


Fig 11

- 20% update, 80% read-only

- 50% update, 50% read-only

In all cases, 40 transactions were randomly generated. The graphs displaying the relative performances for the 3 configurations listed above appear in figures 12, 13, and 14.

As shown in the graphs, the multiple versions did not provide a performance benefit in either one of the three cases. As a matter of fact, in some cases the single version system performed better. The multiple versions do not enhance the performance of the system when the sizes of the read only and update

80% Update, 20% Read-only Transactions

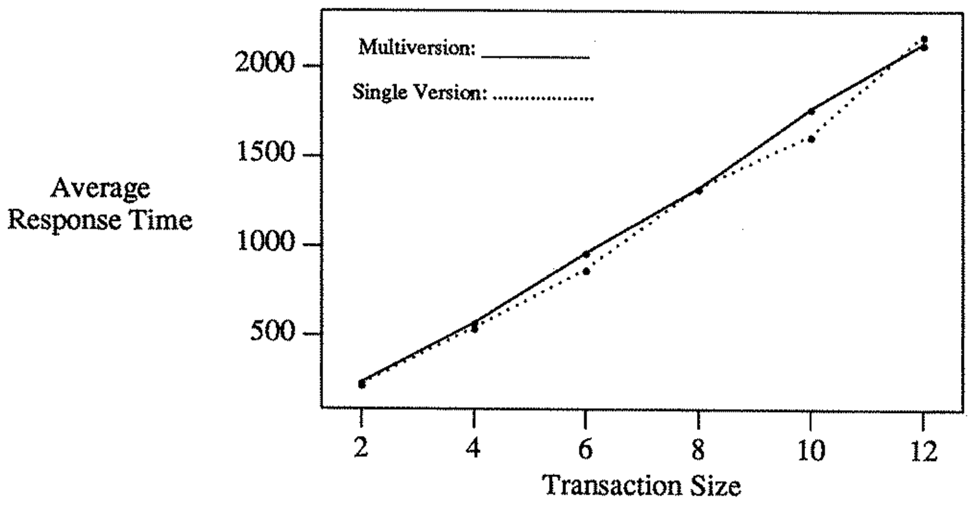


Fig 12

20% Update, 80% Read-Only Transactions

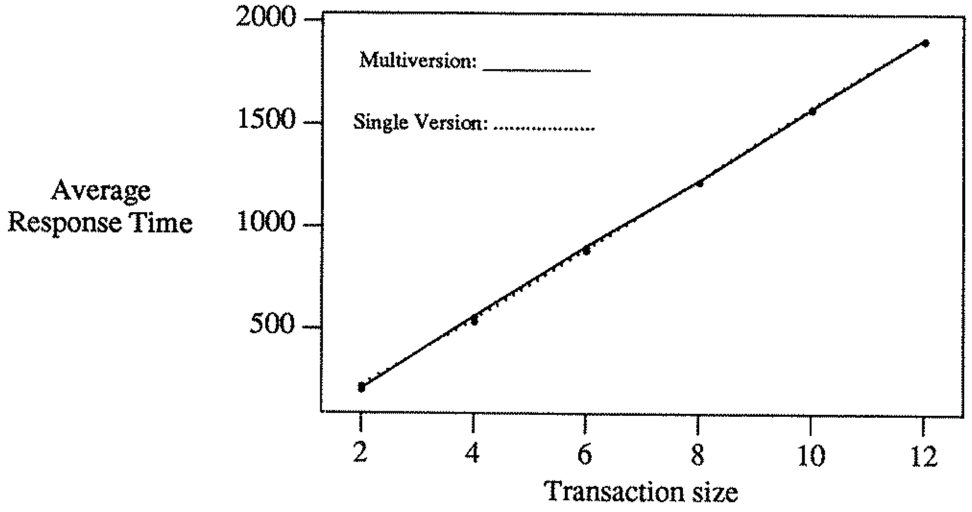


Fig 13

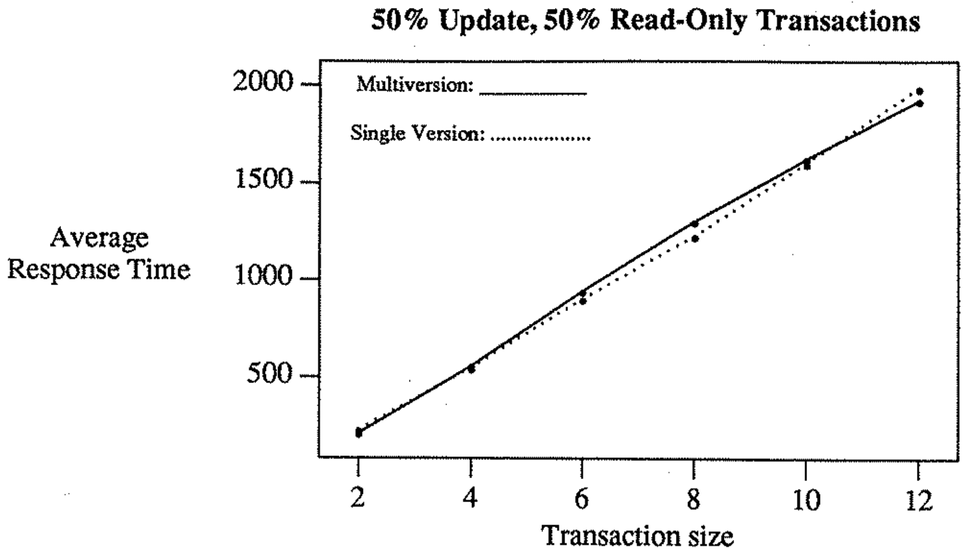


Fig 14

transactions do not differ widely. If the read-only transactions are small or generally the sizes of the read-only and update transactions are close or equal, both systems yield almost the same performances.

In [2] the authors conclude that over a wide range of system conditions, the multiple version method performs only marginally better than the single-version method. They do however caution that the conclusions must be taken in the context of the simulation model assumptions. These assumptions and their influence on the performance evaluation process, as well as the comparison of our results with those of other studies are the topics of section 2.6.

2.4 Experiment 4: Effect of the Degree of Distribution

The purpose of this experiment is to examine the performance of the single and multi-version systems in distributed environments. In our model of a distributed environment, each site (or node) is equipped with its own processor. In addition, each site has its own stable storage which contains a portion of the database. Data items are not replicated; that is each data item (or record) is stored at only one site. The prototyping environment however can be easily modified in order to allow for the duplication of data items at different sites by simply changing the duplication factor. Transactions make data access requests that are either local or remote. Local requests (read or update) are satisfied by accessing data items stored at the site itself. Remote requests call for accessing data items that are stored at other sites. Remote requests are handled through the transaction sending a request message to the data manager of the site that stores the data item. This message passing between sites provides the sharing of the information. In our simulation model, data items accessed by a transaction are randomly and uniformly distributed across the entire database.

The time stamp ordering scheduler used in the implementation to assure serializability and consistency lends itself well to decentralization. Each site has its own scheduler which coordinates the requests for access to data items stored at that site. The decision to schedule, delay, or reject a request depends only on other operations accessing the data items at the site [11]. Since the scheduler at each site is informed of the status of all its local data items, it can make the decision regarding how to respond to access requests without communicating with schedulers at other sites. This independent character of each scheduler makes the distributed synchronization only slightly more complex than the centralized case.

The prototyping environment provides the facility to set up different numbers of sites. Up to 30 different sites in the distributed environment are supported. The system initializes the database by allocating storage for the data items at each site. This allocation follows an even and systematic distribution of data items across sites. In a distributed environment, one of the key factors that affects performance is the communication delay. This delay reflects the price paid in terms of response time for the communication that must take place between two physically separate sites. In our model the inter-site communication delay is 5 times longer than the delay involved in accessing a data item within the same site. This communication factor can also be easily modified in order to examine the influence of varying communication delays on the performance of the system.

Our experiments investigate the behavior of the single and multiversion systems in distributed environments having 2 or 5 sites. In each case the size of the transactions which were generated randomly was held constant at 5 data items. The transaction mix consisted of 80% update and 20% read-only transactions. Figures 15 and 16 show the graphs of the average response times of the systems in a distributed environments with 2 and 5 sites respectively.

Our first observation is that the average response times in the distributed systems were generally much lower than those in the centralized cases observed in experiments of previous sections. This is due to the fact that in the distributed cases we have increased our computational power by providing additional processors. The communication delay introduced as a result of distributing the data items across sites is offset by the increased computational power. With only one site, as seen in the slope of the curves of previous sections, there is a direct relationship between the number of transactions and the average response time. As the number of transactions increase, the average response time also increases proportionally. The slope of Fig 15 is not as sharp, indicating that the effect of increasing the number of transactions on the average response times is more subtle.

In Fig 16 we see that increasing the number of transactions does not seem to affect the average response times at all. This is observed by the horizontal lines of the graph. With 5 processors one at each site, the computational power provided surpasses the delays that having a large number of transactions introduces. Therefore, increasing the number of transactions does not cause a significant increase in the workload of each processor which would consequently result in higher response times. This observation however is not necessarily a true generalization for all distributed environments. The distribution of data access

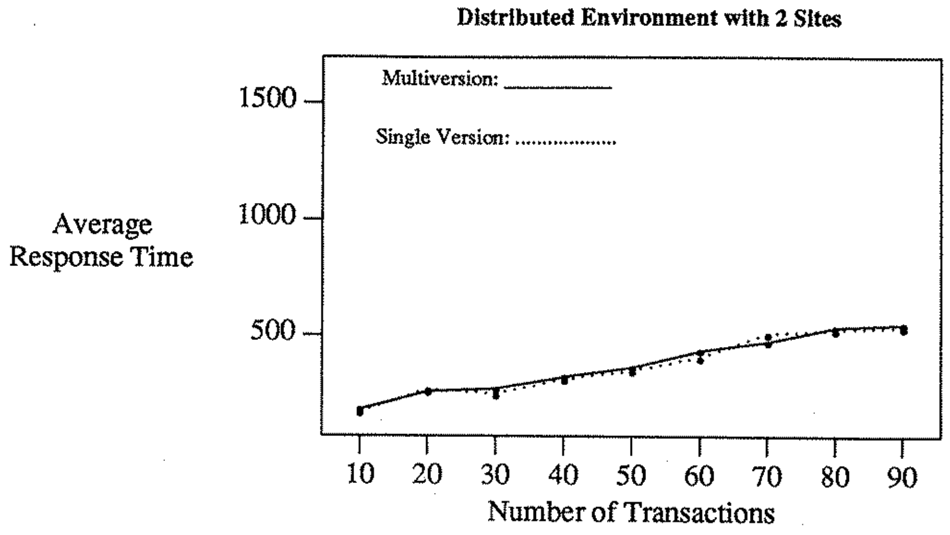


Fig 15

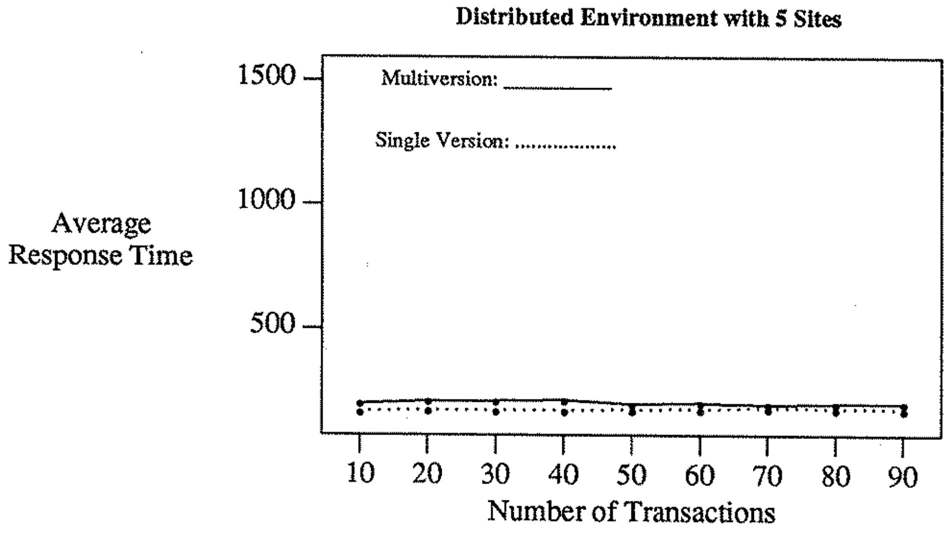


Fig 16

requests from transactions is a very influential parameter affecting the overall performance. Depending on the nature of this distribution, the performance characteristics of a distributed database system may vary widely.

The data access patterns for transactions have been classified in [4] as follows:

1. Accesses that are randomly distributed across the entire database;

2. Accesses that are biased in such a way that, say, 90 percent of the access is for 25 percent of the

database;

3. Accesses that are sequential within one or more areas of the database.

In our simulation, as noted earlier, data items accessed by a transaction are randomly and uniformly distributed across the entire database. This strategy causes the workload to be divided and shared equally by processors at different sites. Therefore, a significant increase in the workload was not experienced by each processor as we increased the number of transactions. Also since requests follow a random pattern, each processor spends a good portion of its time processing remote requests from other sites.

If however, the access pattern is biased toward a certain subset of the database, increasing the number of transactions does not uniformly increase the workload of all processors. In such cases, a small number of processors which are located at the sites where the "hot" areas of the database reside perform the majority of the work. Therefore, a large subset of the processors wait idlely, while a selected few process the remote requests of idle transactions. Consequently, increasing the number of transactions would increase the workload of only a small set of the processors, and therefore we can expect to see a more direct relationship between the number of transactions and the average response times. Also since the majority of data requests are concentrated on a small subset of the database, the number of conflicts resulting from transactions accessing the same data items rise directly as a result of increasing the number of transactions. Under such circumstances we can expect the average response times to increase proportionally as we increase the number of transactions.

From the results of this experiment one might be inclined to conclude that increasing the number of sites will generally lead to lower response times. This however is only true if the communication delay introduced as a result of the distribution does not offset the additional computing power of the processors at different sites. In our simulation where the ratio of local to remote access delay is 1 to 5, the added computational power overrides the communication delay.

As we increase the number of sites within the system, we are potentially increasing the amount of communication that is to take place between sites. As we have more and more sites, we are also partitioning the database into a larger number of units. As the database becomes more "spread out" the possibility that a request will not be satisfied locally increases. This means that at one point we may even get poorer response times when we pass a certain threshold in increasing the number of sites.

2.5 Analysis of Results

The results of the experiments in this section indicated that multiple versions only provide a significant performance enhancement under a mix of transactions for which they were intuitively thought of as being beneficial. In general, a database system adapting a multiversion concurrency control algorithm performs better while processing read requests. Read requests that would be aborted in a database system with a single version of each data item due to time stamp conflicts will be successfully processed in a multiversion system using older versions of the data items. Therefore, when the read requests dominate the transaction load, and there is ample opportunity for read-only transactions to be aborted due to conflicts with update transactions, a multiversion system performs better than its corresponding single version system. The relative sizes of the read and write sets of transactions is an important factor affecting the performance.

Originally we had thought that maintaining multiple versions could only serve to improve the system performance. Intuitively, if the added versions are not used in order to prevent abortions of read-only transactions, they should not degrade the performance of the system either. However in practice, as observed in some of the graphs of the experiments presented in this section, the multiversion system performed even more poorly than its single version counterpart. While examining the operation of the multiversion system we found out that it is possible to implement a multiversion system that is prejudiced against update transactions.

The multiversion system favors read requests. Therefore, if the transaction mix predominantly consists of read requests, the multiversion system outperforms the single version system by avoiding abortions which involve read-only transactions. If, however, write requests dominate the transaction load, the multiversion system performs more poorly by aborting update transactions which were involved in write-write conflicts. The higher number of write-write conflicts causing the abortion of update transactions is due to the way the multiversion system implements the creation of new versions of data items as shown by the example below. Let us assume that our transaction load consists of two update transactions each writing two data items. As shown in the table of Fig 17, transaction 1 updates items 1 and 2, while transaction 2 updates items 2 and 1 in the specified order. The time stamps of transactions 1 and 2 are 5 and 10 respectively. To enhance the illustration of the operations of the transactions in our example, we have also included in the 'Write Set' column of our table the new values to be written. That is, transaction 1 will update data item 1 with the value 'A' and data item 2 with the value 'B'. Transaction 2 will similarly update data items 2 and 1 with the values 'C' and 'D' respectively.

transaction	write set	time stamp
1	1(A), 2(B)	5
2	2(C), 1(D)	10

Fig 17

First we consider the operation of the single version system. The system begins by processing the first requests of both transactions. Since the time stamps of both transactions are greater than the original time stamps of the data items (0), both requests are granted. Note that the actual updating of the data items and the changing of their time stamps do not take place until a transaction's requests are all successfully processed and the transaction commits. Therefore, the time stamps of data items remain 0. This state can be seen in step 0 of Fig 18. Within each data item the values of the read and write time stamps as well as the actual data values are shown.

Next the system processes the second requests of the transactions. Again, since the time stamps of data items have not yet been updated, the requests are granted. Since the write requests of both transactions have been processed successfully, they proceed to commit. First transaction 1 commits changing the time stamps and data values as shown in step 1 of Fig 18. Next transaction 2 commits changing the state of the database as shown in step 2 of Fig 18. Both transactions have updated the database and committed

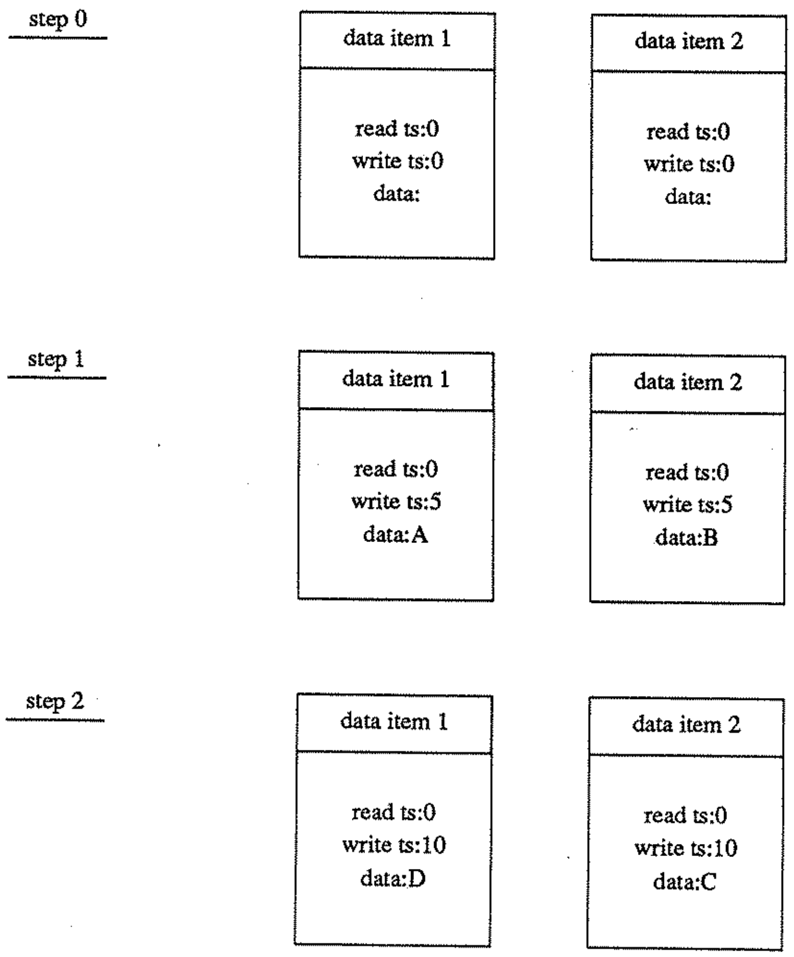


Fig 18

successfully.

We now consider the operation of the multiversion system. As in the single version case the system begins by processing the first requests of both transactions. As seen in Fig 19, temporary versions are built on top of data items 1 and 2. However, as the system proceeds to process the second request of each transaction, it is faced with an attempt to create a new version on top of an already existing temporary version. The implementation of the system allows only one temporary version of each data item at a time, therefore one of the transactions must be aborted.

After one of the transactions has committed and has changed the status of its temporary version to permanent by changing its tag value, then the restarted transaction may proceed with its requests. As seen here, the operation of the multiversion system involved an abortion while in the single version system both transactions committed without conflicts.

It turns out that the restriction of allowing only one temporary version for each data item which was adopted in the implementation of the multiversion system is too costly. It causes abortions due to write-write conflicts which do not occur in the single version system. That is why when the transaction sizes are uniform and update transactions dominate the load, as in the graphs of figures 11 and 12 where 100% and 80% of transactions were update transactions, the multiversion system performed more poorly than the single version system.

The additional constraint of only allowing new versions to be added at the end of the list of versions associated with a data item was also the cause of some abortions. In order to alleviate these limitations which were found to be too costly in terms of additional abortions of transactions, we implemented a new version of the multiversion system which did not incorporate the above mentioned constraints. In the new system, new versions of data items are allowed to be inserted in between two already existing versions and an unlimited number of temporary versions are allowed. When checking to see whether a transaction may commit, the system takes the existence of other temporary versions of data items into consideration.

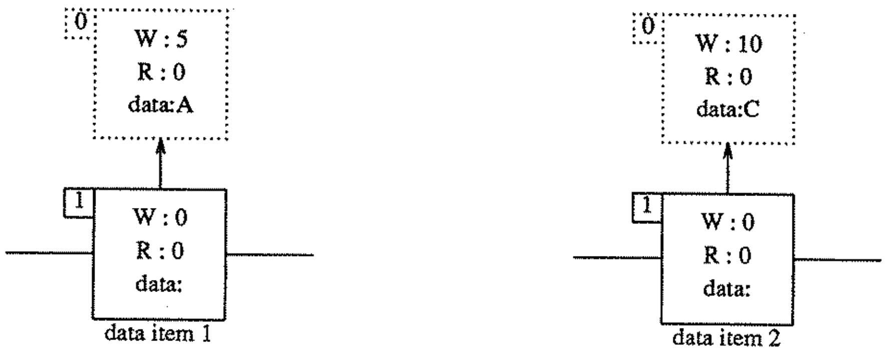


Fig 19

In section 3 we present the results of the experiments which were conducted in order to examine the effect of the removal of the constraints mentioned earlier on the performance of the multiversion system.

2.6 Comparison with other studies

The purpose of this section is to compare our approach to simulation and performance analysis with other similar works in the field and to examine how our results relate to those of other researchers. Multiversion algorithms have been studied from a theoretical point of view [12,13]. These studies have shown that logically, the multiversion approach is capable of enhancing the possible degree of concurrency in a transaction processing system. However, as noted in [3], the actual performance of multiversion systems has received little attention.

The nature of most previous studies and their results were highly affected by the implementation assumptions. For example, in [14] the results are highly influenced by the fact that only two versions of each data item were allowed. In addition, very few of the studies have actually examined the response time of the multiversion concurrency control algorithms. In order to examine the approaches to performance evaluation studies, we need to explore the model assumptions which were involved.

Model assumptions characterize the environment in which the performance evaluation process takes place. The main problem associated with the majority of previous studies is that in most cases the performance evaluation process is completely dependent on the underlying hardware assumptions. In these cases, it is extremely difficult to examine the behavior of different concurrency control algorithms independently of the influence of the underlying architecture. For instance, in [4] the authors present a performance analysis for two different concurrency control algorithms in a distributed environment with two sites. Each site has its own processor (VAX 11/780) and main memory. The database itself resides on two different file servers with different performance characteristics. It is extremely difficult, if not impossible, to consider the performance of the concurrency control algorithms in isolation from the effects of the hardware factors.

Among the many hardware factors which have a significant impact on the transaction processing performance are:

- Virtual memory paging caused by transaction requests

- disk I/O time for accessing the database

- track-to-track seek time

- number of blocks per track

- peak transfer performance of the disks at different sites.

The implementation and test environment dependencies make it extremely difficult for the results given in one study to be applied to other environments.

Among the major limitations of most of the previous modeling studies is that they only consider the performance of different algorithms in centralized environments. In addition, although message-based simulations appear to be more natural for simulating distributed systems, prior to our work, a message based approach to discrete-event simulation of distributed systems had not been fully developed [5]. Our database prototyping environment is a powerful and effective tool for investigating the properties of database control techniques. A major problem of other performance evaluation efforts is that many interrelated factors which affect the performance have been studied as a whole, making it difficult to understand the influence of each on the overall performance. Our system's flexibility allowed us to alter the underlying configuration to a state most suitable for a particular evaluation. Therefore, different design alternatives and algorithms were evaluated in a uniform environment yielding a fair comparison.

Our approach minimizes the limitations imposed upon the performance evaluation process by rigid hardware configurations. For example, one can easily examine the effect of decentralizing the database by increasing the number of sites while keeping the other parameters constant. Different concurrency control algorithms can be selected to process the same set of transactions in identical environments. The CPU and I/O rates are not dependent on the underlying architecture, but are rather simulated by the model and may be altered by the researcher. This flexibility allows the performance evaluation process to be liberated from the limitations imposed by rigid hardware configurations.

Another critical parameter is the delay involved in communicating between two different sites. The nature of the effect of the communication delay on the performance results has not been fully considered. For example in [4], the communication delay is completely dependent on the underlying machine architecture since the VAX/VMS mailbox mechanism has been used. This inter-node communication delay is always fixed at about 7.2 milliseconds. In such a system, it is impossible to avoid the effect of the communication overhead on the CPU utilization as each inter-node communication also consumes CPU time. Our model on the other hand simulates and keeps the CPU and communication costs separately. Also since in our model the communication delay is modeled by causing a process to wait (or be blocked) for a finite time interval, this communication delay may be altered easily by changing a parameter to examine its influence.

Another set of factors that are influential in the results of a performance evaluation process are the characteristics of the underlying database. Most of the studies done either assume for the database to be partitioned (not-replicated) or replicated at different sites. Our model allows for both of these assumptions as well as different degrees of replication by modifying the duplication factor.

The nature of creation and maintenance of new versions of data in a multiversion system has not been considered carefully. The model in [3] allows only one temporary version while creating a new version. This is an implementation assumption which, as we have seen earlier, has a significant affect on results. Our database model is flexible and can easily adopt different characteristics.

Our task, as in [2], has been to compare concurrency control algorithms using the same model assumptions, performance measures, and application environment parameters. Our results confirm those observed in [2]. The multiversion system performs only marginally better in some of the cases. As observed in [3], our multiversion system performed better under a mix of transactions for which the multiple versions were thought to be beneficial. Under a workload of large read transactions and a large number of update transactions, the probability that the read transactions will be aborted due to conflicts with update transactions is very high. The multiversion algorithm avoided the abortion of read transactions in all cases. However, the performance benefits observed in our study were not as significant as those reported in [3]. Our results also confirmed those observed in [2]: the multiple versions do not provide a significant performance benefit when the read-only transactions are small.

The performance evaluation results in [3] indicate that the multiversion algorithm offers a significant performance improvement in all cases. This is in conflict with our observations. When the transactions are small, the probability of aborting the read-only transactions is small anyway, so multiple versions are not of much help in preventing abortions. Unless the transaction mix is such that there is a high probability for the abortion of read-only transactions due to conflicts with update transactions, the multiple versions will not provide a significant performance benefit. When the transaction mix consists of a large number of small update transactions and fewer large read-only transactions, the multiple data versions will help in preventing the abortion of read-only transactions by allowing them to access previous versions of the data items. If the abortion of read-only transactions is not highly probable to begin with, there is little chance that the multiversion system could perform better by avoiding such abortions.

In summary, the following three characteristics of our approach provide us with a more powerful tool for conducting performance analysis of different algorithms:

- The capability to study the single and multiversion concurrency control algorithms in both a centralized and distributed environment so that the benefits of multiple versions (if any) may be isolated from the effects of data distribution;

- The flexibility allowed in deriving different system and application environment configurations;

- The freedom from the effects of the underlying architecture; particularly the way the I/O and CPU ratios are specified as parameters to the simulation.

3. Comparison of Two Multiversion Algorithms

In this section we will compare the performance of a new implementation of the multiversion system with that of the system studied in the previous sections. The reason for implementing the new multiversion system was to remove the constraints in the creation and maintenance of new versions of data items which were found to be too costly in terms of abortions of update transactions. The restriction of allowing only one temporary version for each data item which caused abortions due to write-write conflicts has been removed in the new implementation. The additional constraint of only allowing new versions to be added at the end of the list of versions associated with a data item has also been removed. In the new implementation, new versions of data items are allowed to be inserted in between two already existing versions and an unlimited number of temporary versions are allowed.

Since intuitively the new version should prevent the abortions of update transactions in the multiversion system which are caused by write-write conflicts, we will start our comparison with a transaction load which emphasizes such conflicts: 100% update transactions. The graph is shown in figure 20. This graph should be compared with that of figure 11. As seen in the graph of figure 11, the multiversion system performed more poorly than its single version counterpart. Since all the transactions were update transactions, this illustrates the occurrence of abortions which were caused by the implementation constraints of the multiversion system. In the new implementation, as seen in figure 20, both systems show identical performances. The removal of the constraints in the creation and maintenance of multiple versions was beneficial. Although with 100% update transactions in the workload the multiple versions can not provide a performance benefit, they will not degrade the performance of the system either.

We next examine the behavior of the new implementation under a transaction mix which intuitively

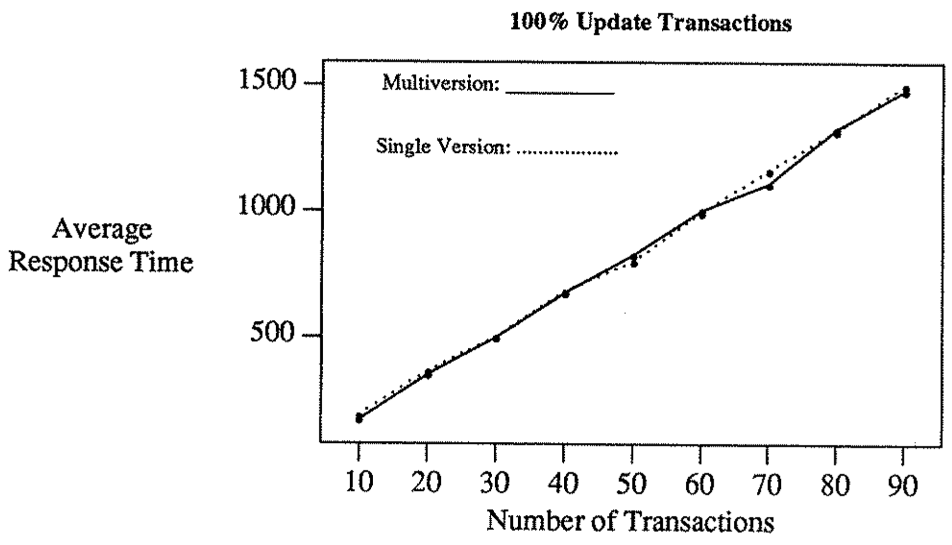


Fig 20

should allow the multiple versions to be of beneficial use: 80% update and 20% read-only transactions. This workload allows ample opportunities for the abortions of read-only transactions as a result of read-write conflicts. The graph of figure 21 displays the relative performances of the new multiversion system and its single version counterpart. This graph should be compared with figure 12. As shown in figure 12, the original implementation of the multiversion system even performed marginally worst than the single version system. This is due to the fact that even though the original implementation avoided the abortion of read-only transactions, there were plenty of opportunities for the abortion of update transactions with 80% update transactions in the multiversion system. So any performance benefits provided as a result of avoiding the abortions of read-only transactions was offset by the more plentiful abortions of update transactions. In the new implementation, since the abortion of update transactions is not a problem, as the transactions get larger, the multiversion system performs better and better than the single version system by preventing the abortion of read-only transactions.

Similar results are observed in the graph of figure 23 where the transaction load consists of 50% update and 50% read-only transactions. This graph should be compared with figure 14.

The results presented in this section indicate that in implementing multiversion concurrency control algorithms, careful consideration must be given to the constraints imposed by the manner data versions are created and maintained. Particularly important are the number of temporary versions that are allowed for a single data item at any given time and the restrictions that may be imposed on the insertion of new data versions within a list of already existing versions. We have found that best performance results were observed when such restrictions were minimized.

4. Multiple Data Versions in Real-Time Environments

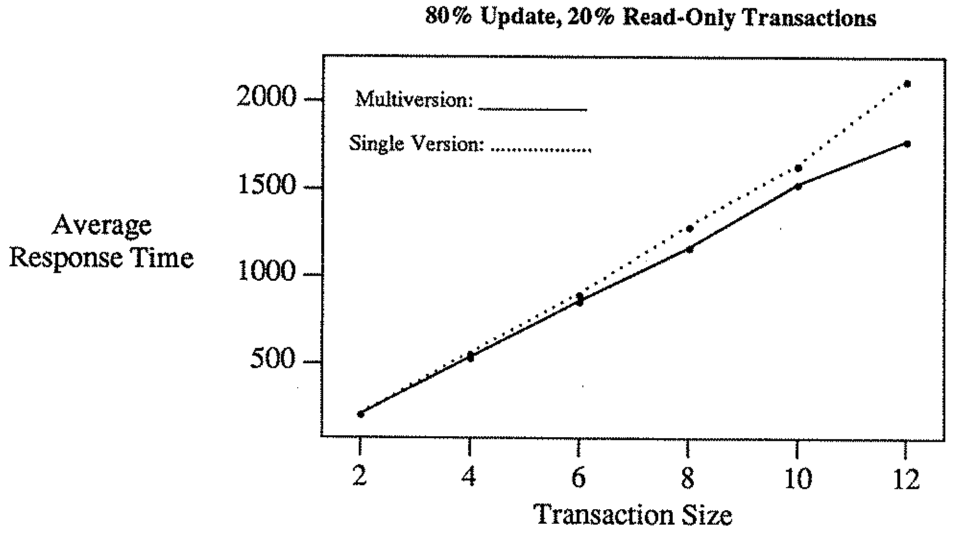


Fig 21

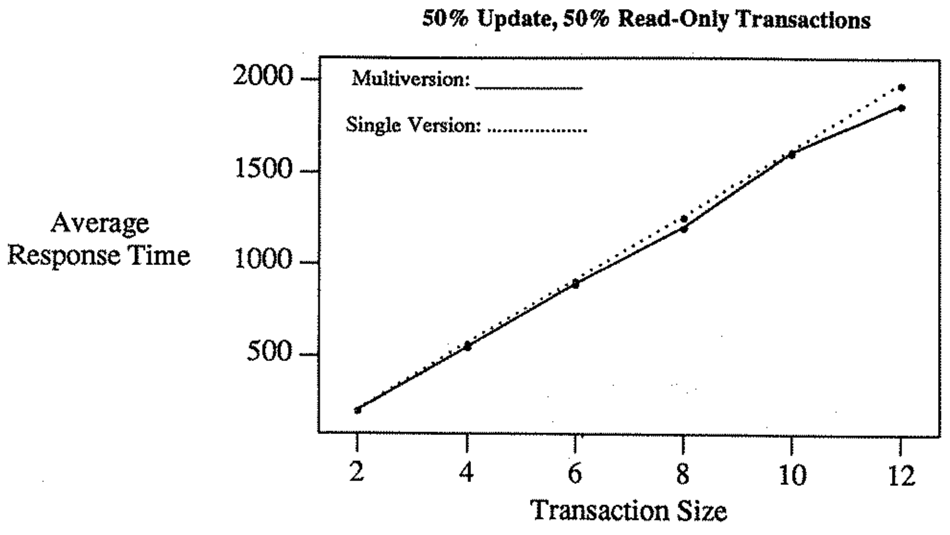


Fig 22

Although the maintenance of the multiple data versions did not degrade the system performance, as discussed in the previous section, their availability only proved to be of significant benefit in some specific cases with transaction loads of special characteristics. The emphasis in this section is to examine whether the multiple data versions can be valuable in database systems operating in real-time environments.

Real-time systems have gained significant importance and provide a wide open research area of intellectually challenging computer science problems[16]. We define a database system in the context of a real-time environment such that the transactions and/or data items having real-time constraints. An example of such a system is program trading as described in [17]. In such a system for instance, a monitor/update process monitors the current state of a physical system (e.g. the stock market) and updates a database accordingly. If the database is to contain an accurate representation of the current physical system, these monitoring and updating activities must be performed with some real-time constraints associated with them. Other examples of the application of database systems in real-time environments may be found in radar tracking systems, nuclear power plants, and flight control systems.

In real-time systems, correctness may be defined as providing the correct results in a timely manner[18]. The focus of our research outlined in this section is how multiple data versions can be used in order to better meet the timing constraints characteristic of such systems. In addition, we propose an algorithm which considers temporal requirements when scheduling real-time transactions.

4.1 Timing Constraints Associated with Transactions

We first consider the case where the timing requirements are attached to the transactions. Our prototyping tool was modified such that each transaction incorporated these temporal constraints. Both single and multiversion systems were implemented. Within these systems, a transaction is characterized by its:

- data requirements,
- timing constraints, and
- accuracy requirements.

As usual, the data requirements provide the list of data items that a transaction accesses for reading and/or writing purposes. The timing constraints are provided in terms of deadlines. The deadline specifies the time by which the transaction must be completed. Figure 23 shows an example of a transaction in such a system. The entries indicate that the transaction with start time of 20 must be completed by time unit 55. In the current implementation, we treat the timing constraints as *hard deadlines* meaning that upon the transaction missing its deadline, the transaction gets aborted. We assume that once the transaction's deadline has been missed, there is no value in completing its computation. We will discuss this issue more fully when considering the scheduling of real-time transactions.

The accuracy requirements of transactions are provided in terms of temporal consistencies. The temporal consistency provides a time interval, relative to the start time of a transaction, during which accurate states of data items may be accessed. In our example, the temporal consistency requirement is 15, indicating that the data items accessed by the transaction can not be *older* than 15 time units relative to the start time of the transaction. That is, all the data items read by this transaction must have write time stamps in the range of 5 to 20. An attempt to read an inaccurate data item(i.e. one whose write time stamp is outside of this interval) will cause the transaction to abort. While a deadline can be thought of as providing a time interval as a constraint in the future, the temporal consistency specifies a temporal window as a constraint in the past.

In order to illustrate the usefulness of multiple data versions in such a setting, we next outline the operations of both the single and multiversion systems under the same set of input transactions. The scenario consists of two transactions as depicted in figure 24. Transactions 1 and 2 each have start times of 5 and 15 while their respective deadlines are 60 and 70. The temporal consistency requirement of transaction 1 is 10, while that of transaction 2 is 13. Transaction 1 writes data item 1 and reads data item 2, while transaction 2 writes 2 and 3 and reads 1. The initial state of the database is shown in figure 25. Each

Trans. #	Start Time	Deadline	Temporal Cons.	Data Items
1	20	55	15	1,2,3

Fig 23

rectangle represents a data item. Note that the write time stamp value shown within each data item is set to 0 in order to indicate the initial state of the data items. The write time stamp of each data item represents the time stamp of the transaction which updated that data item last. In the multiversion system's case, the write time stamp represents the time stamp of the transaction which created the particular version of the data item.

The scenario representing how the single version system handled the workload is shown in figure 27. Note that we are not presenting a step by step representation of what takes place at every single time unit as the transactions are being processed. Such an illustration would be too lengthy and detailed for our purposes. We show the significant events which are of interest to us in order to examine the relative behaviors of both systems. The time units reported are the actual ones observed in our simulations. At each time unit shown, we present the state of the database as well as the transactions present in the system. The read and write requests of transactions are shown above their respective data items. Requests which were successfully granted are shown with bold-face lettering, while the failed requests are printed

Trans. #	Start Time	Deadline	Temporal Cons.	Data Items	
				Write	Read
1	5	60	10	1	2
2	15	70	13	2 3	1

Fig 24

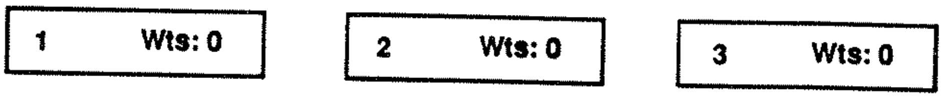


Fig 25

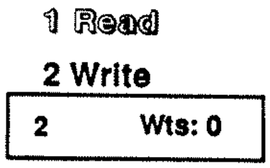


Fig 26

in outline lettering. The example in figure 26 shows that transaction 2 has successfully processed a write request for data item 2 while transaction 1's read request for that data item has failed.

As shown in figure 27, at time unit 16 both transactions are present in the system. Transaction 1 has successfully submitted a write request for data item 1 while transaction 2's write request for data item 2 has also been granted. The transactions proceed and at time unit 19, transaction 1 submits a read request for

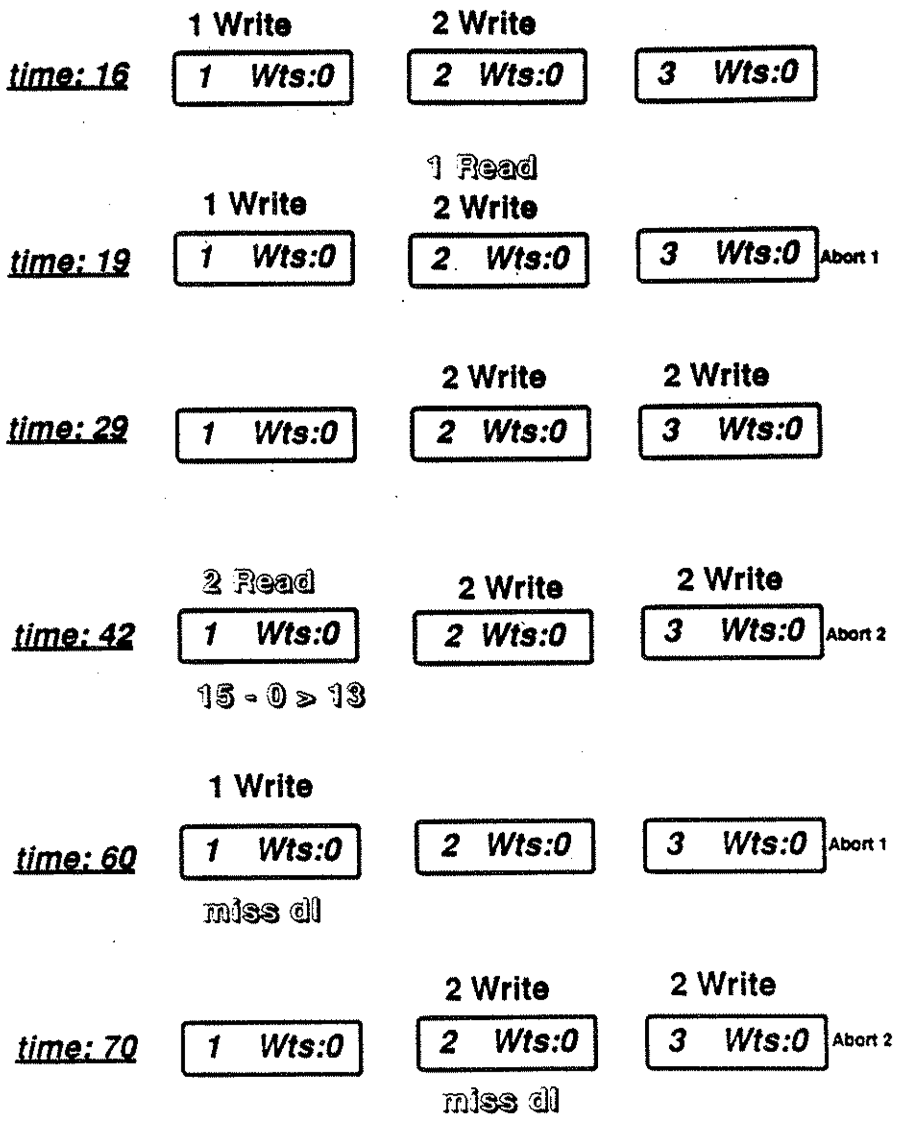


Fig 27

data item 2. Since in the single version system a transaction is not allowed to read a data item while another transaction is simultaneously updating it, this situation causes a read/write conflict and the read request of transaction 1 can not be granted. Therefore, the system aborts transaction 1 for a later restart.

Transaction 2 proceeds and at time unit 29 its write request for data item 3 is granted. Note that at this time, transaction 1 since aborted earlier, is not present in the system. Transaction 2 continues by submitting a read request for data item 1 at time unit 42. Before a read request for a data item can be granted, the temporal consistency requirement of the transaction with respect to that data item's write time stamp must be satisfied. The write time stamp value of the data item is subtracted from the start time of the transaction. This gives the age of the data item with respect to the transaction's start time. If this value is less than the temporal consistency requirement, the read request may be granted. Otherwise, reading the data item would violate this temporal requirement and therefore the transaction is aborted.

Returning to our example, we see that at time unit 42, the age of the data item (15 - 0) is greater than the required consistency interval of the transaction(13). Therefore, transaction 2 is aborted. Later on, at time unit 60, we see that transaction 1 has been restarted. But by the time this transaction acquires its write request for data item 1, it misses its deadline of time unit 60 and is therefore aborted. Transaction 2 is similarly aborted at time unit 70 as a result of missing its deadline. So as seen through this scenario, both transactions miss their deadlines and exit the system without successfully completing. We next see how the multiversion system handled the same set of transactions.

As shown in figure 28, at time unit 19 both transactions are present in the system. Transaction 1's write request for data item 1 and transaction 2's write request for data item 2 have been successfully granted. In addition, transaction 1's read request for data item 2 has also been granted. This is because in the multiversion system, while transaction 2 is actively pursuing the creation of a new version of data item 2, we can simply allow transaction 1 to read the original version of this data item. As seen here a read and a write on the same data item do not create a read-write conflict in the multiversion system and can take place simultaneously.

Since transaction 1 has processed both its requests successfully, it proceeds to commit. As seen at time unit 41, transaction 1 has created a new version of data item 1 with a write time stamp value of 5. This is the start time of transaction 1. In the current implementation, we have adopted the practice of assigning the start time of a transaction as the write time stamp of the data versions it creates. Returning to our example, we see that transaction 2 proceeds by submitting its write request for data item 3 at time unit 41 and its read request for data item 1 at time unit 43. The write request is obviously granted. However, before the read request can be granted, the temporal consistency requirement of the transaction must be checked.

Since transaction 1 has already committed, the more recent version of data item 1 created by this transaction is available for transaction 2 to read. The age of this version of the data item with respect to the start

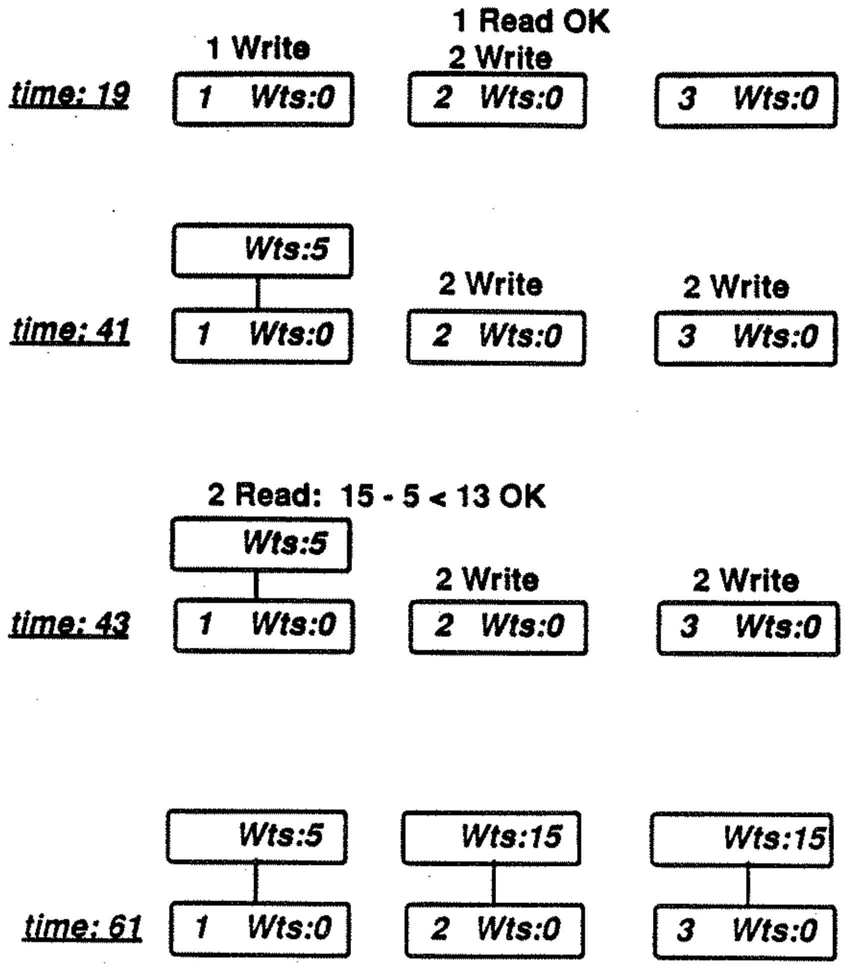


Fig 28

time of transaction 2 (i.e. 15 - 5) is less than the temporal consistency requirement of the transaction (i.e. 13) and therefore the read request is successfully granted. Note that if this later version of the data item were not available, reading the original version would have caused the abortion of transaction 2 as in the single version system. Transaction 2 proceeds to commit since all its requests have been granted and the final state of the database with the new data versions are shown at time unit 61. Note that both transactions have successfully committed before their deadlines.

Our scenario has shown that avoiding the abortion of a transaction because of a read-write conflict by the multiversion concurrency control algorithm may lead to the availability of more recent data versions which can be used to satisfy the temporal consistency requirements of transactions and therefore successful completion of transactions before their deadlines. We next examine how the multiple data versions may be of benefit when the timing constraints are associated with the data items.

4.2 Timing Constraints Associated with the Database

In a real-time database system the temporal requirements may be associated with the data items. Examples of such requirements may be found in a radar tracking system where because of the database being characterized by frequent and rapid changes, the accuracy of the data items may gradually expire. Certain degrees of accuracy may be associated with accessing data values during various time intervals after their most recent updating. In this section we examine the usefulness of the multiple data versions in a system incorporating such temporal requirements. Both the single and multiversion systems in our prototyping environment were modified to reflect these requirements.

In our implementation, the real-time requirements associated with the database are specified in terms of valid time intervals. Each data item in the database has a valid interval specification attached to it. The valid interval indicates the time interval after the most recent updating of a data item during which a transaction may access a data item with 100% degree of accuracy. What occurs when a transaction attempts to access a data item outside of its valid interval is dependent upon the semantics of data items and the particular implementation. In our system, we assume that reading a data item after its valid time interval has expired will result in 0% accurate data values. Therefore, in the current implementation, an attempt for such data access will force the transaction to abort.

The example in figure 29 shows a data item with a write time stamp value of 5 and a valid interval of 30. In 29.a we see the current implementation's strategy in treating the valid intervals. 29.b and 29.c show two other possibilities of how the valid intervals may be implemented within the system. In 29.b, reading a data item during its second interval will result in data values which are only 50% accurate. In 29.c, the accuracy of the data values degrade gradually until a 0% accuracy is reached.

As we did in the previous section, we will now examine the operation of the single and multiversion systems under the same set of transactions. The scenario this time as shown in figure 30 consists of 3 transactions. The start time of the transactions are 5, 10, and 45 with each having a deadline of 60, 70, and 100, respectively. Transaction 1 writes data item 1 and reads data item 3. Transaction 2 writes 3 and reads 4, while transaction 3 reads 1. The initial state of the database consisting of data items 1, 3, and 4 is shown in figure 31. All initial write time stamps are set to 0, while the valid intervals associated with the data items are 47, 40, and 60, respectively.

We begin by examining how the 3 transactions were processed in the single version system. Figure 32 shows the operation of the single version system under this transaction load. Since a read-write conflict in the single version system results in the abortion of a transaction, at time unit 19 transaction 1 is aborted because of such a conflict involving data item 3. Transaction 2 proceeds by submitting its read request for data item 4. Before a read request may be granted, the valid interval requirement must be verified. As shown at time unit 28, since the age of the transaction relative to the current time (28 - 0) is less than data item 4's valid interval specification (60), the read request is successfully granted. Transaction 2 proceeds to commit and it therefore changes the write time stamp of the data items it has updated. As seen at time unit 51, the write time stamp of data item 3 has been changed to 10 in order to reflect the start time of transaction 2. At time unit 51 we observe that transaction 1 has been restarted and has already submitted a successful write request for data item 1. At this time transaction 3 submits a read request for the same data item which causes it to abort. Transaction 1 continues by processing a read request for data item 3 at time unit 55. However accessing this data item at this time would involve a violation of its valid interval requirement (since 55 - 10 > 40). Transaction 1 is aborted and by the time it is restarted it misses its deadline at time unit 60. Once restarted, transaction 3 similarly misses its deadline.

As seen in this scenario, as a result of the initial abortion of transaction 1, by the time this transaction is restarted the accuracy of the data item it attempts to read has been expired. Transaction 3 aborts since it does not have access to a more recent update of data item 1 generated by transaction 1. We now see how the transactions were processed in the multiversion system. The scenario is shown in figure 33.

At time unit 19 transaction 1 submits its read request for data item 3. Again, since this transaction can read the original version of this data item while transaction 2 creates a new version, and since the valid interval requirement is satisfied (19 - 0 < 40), the read request is successfully granted. Transaction 1 proceeds to commit and as seen at time unit 29 a new version of data item 1 is created with a write time stamp value of 5 (start time of transaction 1). At this time, transaction 2 submits its read request for data item 4. The valid interval requirement is satisfied (29 - 0 < 60) and transaction 2 commits. The new state of the database with the newly created data versions is shown at time unit 38. At time unit 50 transaction 3 submits its read request for data item 1. Since the latest version of this data item is available for transaction 3 to read, the valid interval requirement is satisfied. That is the age of the latest version of the data item relative to the current time (55 - 5) is less than the specified valid interval value (47). Transaction 3 therefore also commits. Note that all three transactions have committed before their deadlines. In this scenario we observed how avoiding the abortion of a transaction in the multiversion system lead to the availability of more recent data versions which were used to meet the temporal constraints of data items.

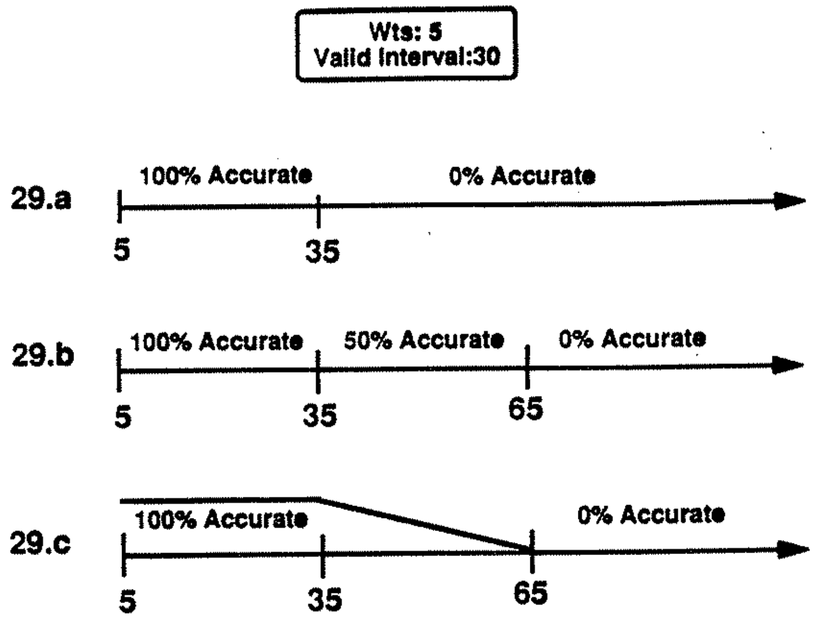


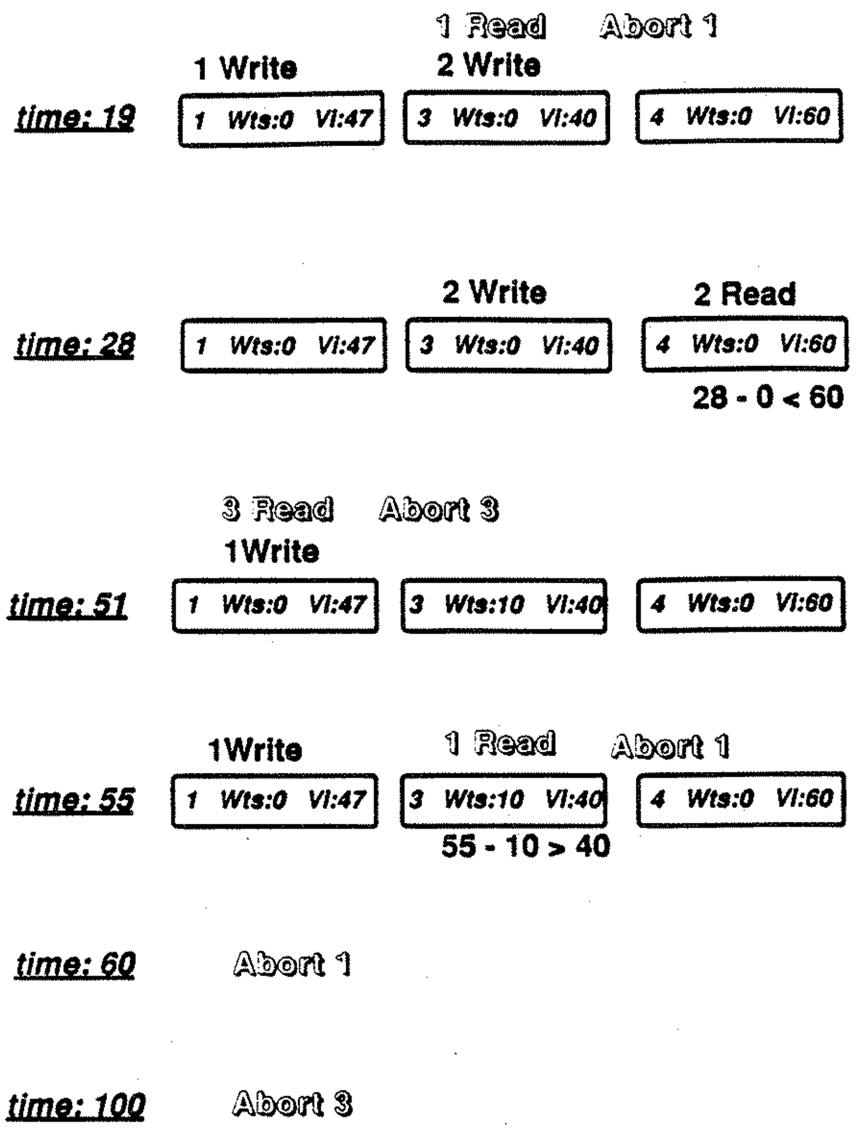
Fig 29

Trans. #	Start Time	Deadline	Data Items Write	Data Items Read
1	5	60	1	3
2	10	70	3	4
3	45	100		1

Fig 30

1 Wts: 0 Vl:47	3 Wts: 0 Vl:40	4 Wts: 0 Vl:60

Fig 31



1 Read
Abort 1
1 Write
2 Write
time: 19
1 Wts:0 Vl:47
3 Wts:0 Vl:40
4 Wts:0 Vl:60
2 Write
2 Read
time: 28
1 Wts:0 Vl:47
3 Wts:0 Vl:40
4 Wts:0 Vl:60
28 - 0 < 60
3 Read
Abort 3
1Write
time: 51
1 Wts:0 Vl:47
3 Wts:10 Vl:40
4 Wts:0 Vl:60
1Write
1 Read
Abort 1
time: 55
1 Wts:0 Vl:47
3 Wts:10 Vl:40
4 Wts:0 Vl:60
55 - 10 > 40
time: 60
Abort 1
time: 100
Abort 3

Fig 32

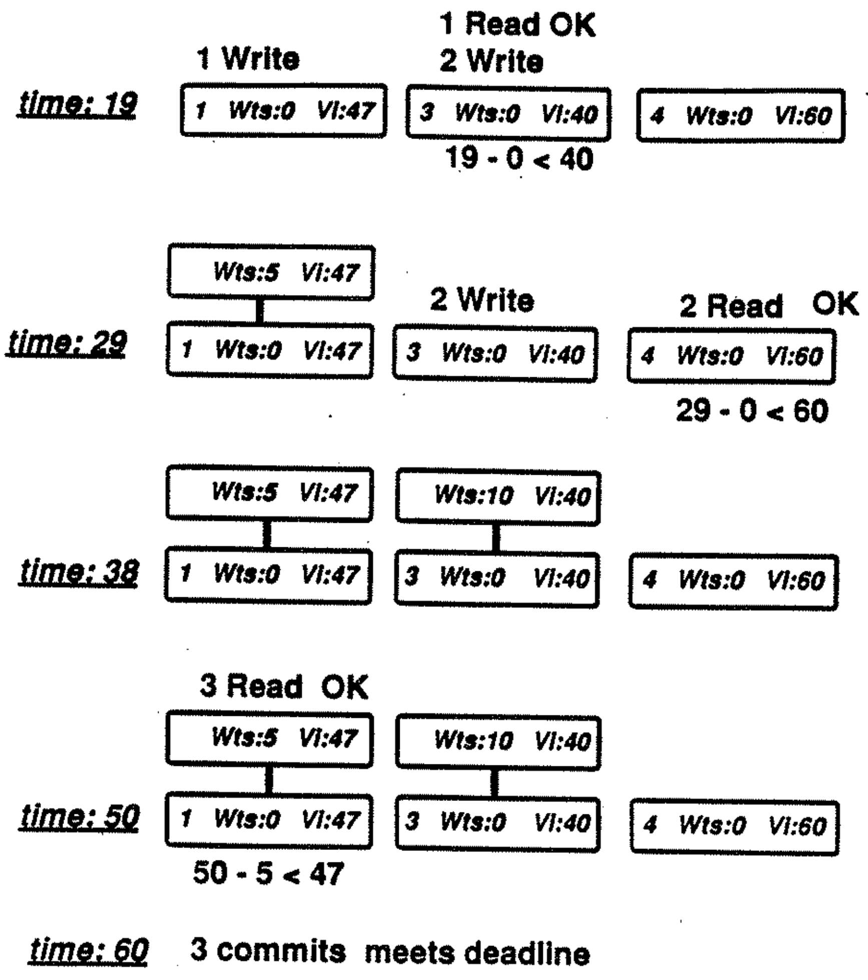

1 Read OK
1 Write
2 Write
time: 19
1 Wts:0 Vl:47
3 Wts:0 Vl:40
4 Wts:0 Vl:60
19 - 0 < 40
Wts:5 Vl:47
2 Write
2 Read OK
time: 29
1 Wts:0 Vl:47
3 Wts:0 Vl:40
4 Wts:0 Vl:60
29 - 0 < 60
Wts:5 Vl:47
Wts:10 Vl:40
time: 38
1 Wts:0 Vl:47
3 Wts:0 Vl:40
4 Wts:0 Vl:60
3 Read OK
Wts:5 Vl:47
Wts:10 Vl:40
time: 50
1 Wts:0 Vl:47
3 Wts:0 Vl:40
4 Wts:0 Vl:60
50 - 5 < 47
time: 60
3 commits meets deadline

Fig 33

4.3 Scheduling Real-Time Transactions

In real-time database systems, scheduling decisions are often directly related to whether the transactions meet or miss their deadlines. Scheduling decisions must be made when the scheduler has to select from amongst a collection of transactions that are ready to be started or when a choice has to be made between two or more transactions which are competing for the same resources (i.e. data items). A decision to abort a transaction for a later restart may result in the transaction missing its deadline. Required is a scheduler which incorporates a consideration of the timing constraints of the transactions and the database in its scheduling decisions.

The goals of such a scheduler in a real-time environment are:

- to minimize the number of transactions that miss their deadlines,
- to ensure meeting the timing requirements of highly critical transactions, and
- to maximize the overall transaction accuracy within the system.

A simplistic approach would be to solely consider the minimization of transaction loss due to missing deadlines. This goal by itself however is not sufficient since the transactions due to their temporal requirements may have different degrees of criticalness associated with them. A scheduler may be able to maximize the overall number of transactions which meet their deadlines by successfully scheduling the less critical transactions and causing the highly critical transactions to miss their deadlines. The failure of these highly critical transactions may be too costly in terms of endangering the consistency of the database. Therefore it would be desirable for the scheduler to make an effort to ensure that the deadlines of the highly critical transactions are met. Finally, since the transactions may have different degrees of accuracy associated with them based on their timing requirements and the temporal constraints of the data items, we would like for the scheduler to consider the overall transaction accuracy within the system before making a decision for or against a transaction.

We propose that the above mentioned goals may be achieved by having the scheduler consider the deadlines, the criticalness, and the accuracy levels which are associated with transactions. We now consider each one of these elements separately before we incorporate them in the scheduling decision.

4.3.1 Deadline

An intuitive approach would be to assign the highest priority in the system to the transaction with the smallest deadline. Since this transaction is at the highest risk of missing its deadline, it would be favored in the scheduling decision. This approach is not acceptable because it fails to consider another important factor. What if the transaction is so close to its deadline that we can be reasonably certain that it will miss its deadline. Making a decision in favor of this transaction may lead to the competing transaction missing its deadline and the transaction itself also aborting in time. Therefore, it would be desirable for the scheduler to consider the *feasibility*[17] of the deadlines of the transactions prior to making a decision.

The feasibility of the deadline of a transaction may be determined as proposed in [17]:

$$t + (E - (t - Ts)) < d.$$

Where t is the current time, E represents a run time estimate of the transaction, Ts is the start time of the transaction, and d represents the deadline of the transaction. The equation above subtracts the age of the transaction (t - Ts) from the run time estimate and adds the result to the current time. If this value is less than the deadline, then the transaction has a feasible deadline. Otherwise, the deadline is infeasible.

We would also like for the scheduler to consider the *slack time* of transactions. The slack time of a transaction is the amount of time a transaction may be delayed and still commit by its deadline. [17] also defines the slack time by:

$$\text{Slack} = d - (t + E - (t - Ts)).$$

The remaining time for a transaction based on its run time estimate is added to the current time and the result is subtracted from its deadline. Other factors having been considered, the smaller the slack time of a transaction, the higher its priority would be within the system.

4.3.2 Criticalness

We define the criticalness of a transaction as a function of the current time and the valid time intervals and write time stamps of the data items that the transaction makes access to. Let us assume that transaction Ti accesses data items A1, A2, through An. Then Ti's criticalness can be determined by:

$$C(Ti) = \text{MIN}(Aj.\text{Valid\_Interval} - (t - Aj.\text{Write\_Stamp})) \quad \text{For } j = 1,2,\ldots,n.$$

For each data item, the write time stamp value is subtracted from the current time. This results in the current age of the data item. This value is then subtracted from the valid time interval of the data item. The resulting number is the time interval that the transaction can be delayed and still access this data item within its valid time interval. We compute this number for all the data items the transaction makes access to and select the minimum value as the criticalness of the transaction. This value represents the maximum amount of time a transaction may be delayed and still satisfy the temporal constraints of all the data items it makes access to. Other factors being considered as well, the smaller this criticalness value associated with a transaction, the higher its priority should be within the system.

4.3.3 Accuracy

Since accessing data items at different times will have different degrees of accuracy associated with it, it would be desirable for the scheduler to consider the resulting accuracy of the transactions prior to making a decision. We define the overall accuracy of a transaction at a particular instant as a function of the current time and the timing requirements of the data items that it makes access to:

$$\text{Accuracy}(Ti) = \frac{F(t, Aj.\text{Valid\_Interval}, Aj.\text{Write\_Stamp})}{n}$$

Function F is defined in figure 34. This function implements the approach in assigning proper accuracy values which was presented in figure 29.b. After determining what interval in relation to the write time stamp and the valid time interval of a data item the current time falls in, the degree of accuracy attained by accessing the data item at that particular instant is assigned. The degrees of accuracies achieved by accessing each one of the data items is computed and the average of these is assigned as the accuracy of the transaction were it to be processed at this instant. While considering other factors as well, the higher the accuracy of a transaction, the higher its execution order should be within the system.

4.3.4 The scheduling decision

We assume that the the transactions in the system can be characterized as belonging to one of the two following groups:

- most critical transactions, or

```
F ( t, Aj.Valid_interval, Aj.Write_stamp)
    IF Aj.Write_stamp < t <= (Aj.Write_stamp + Aj.Valid_interval) THEN
        F = 100
    ELSEIF Aj.Write_stamp+Aj.Valid_interval < t <= Aj.Write_stamp +
                                                2(Aj.Valid_interval) THEN
        F = 50
    ELSE
        F = 0
    END
```

Fig 34

- less critical transactions.

The transactions in the most critical group can be thought of as having hard deadlines. This implies that if these transactions do not meet their deadlines, the consistency of the database may be compromised. Within the most critical group the transactions may have different priorities, but the lowest priority of this group is still higher than the highest priority found in the less critical group. The scheduling decisions of the transactions in the most critical group will be based solely on their priorities and deadlines.

The deadlines of the transactions in the less critical group however can be thought of as being soft. The system will do whatever in its power to satisfy the timing requirements of these transactions but will not guarantee that these transactions will meet their deadlines. Since in a real-time environment the accuracy, criticalness, and the feasibility of the deadline of a transaction changes as a function of the time, we allow the scheduler to determine the priorities of transactions within this group dynamically. So a given transaction in the less critical group may have different priorities at different times. But, at any given time, the dynamic priority of a transaction in this group is less than the priority of all the transactions in the most critical group.

The main scheduling decision of the scheduler is presented in figure 35. In the figure, Th represents a transaction holding some resource, and Tr is a transaction requesting the same resource. Therefore the scheduler must make a decision on whether to keep Th running or to preempt Th and run Tr.

If both transactions belong to the most critical group, then a decision can be made between them based on their priorities. We may use the conditional restart algorithm presented in [17]. If the competing transactions belong to two different groups, we always make a decision in favor of the transaction in the most

```
IF MostCritical(Th) AND MostCritical(Tr) THEN
    ConditionalRestart(Th, Tr)
ELSEIF LessCritical(Th) AND LessCritical(Tr) THEN
    ExecutionOrder(Th,Tr)
ELSEIF MostCritical(Th) AND LessCritical(Tr) THEN
    keep Th running
    delay Tr
ELSEIF LessCritical(Th) AND MostCritical(Tr) THEN
    Abort(Th)
    run(Tr)
END
```

Fig 35

critical group. If however both transactions belong to the less critical group, then we decide between the two transactions based on the value of their dynamic execution orders. Figure 36 shows what this scheduling decision is based on. The structure of this algorithm is based on ideas presented in [17].

The algorithm first considers the feasibility of the deadline of Tr. If this deadline is found to be infeasible, we keep Th running. If however the deadline of Tr is feasible, the scheduler proceeds by considering the dynamic order of both transactions. The dynamic order of a transaction is a function of its accuracy, slack time, and criticalness as shown in figure 37. The weight factors a, b, and c may be adjusted to emphasize or de-emphasize the different factors involved.

We first check to see whether the dynamic order of Th is less than that of Tr. This condition by itself however is not sufficient reason for the scheduler to preempt Th and give access to Tr. Since the execution orders are determined dynamically, it is possible that the execution order of a transaction rises immediately after it has been preempted. We want to avoid the situation when the preempted transaction will immediately have a higher order than the transaction that just gained access to the data item. This may lead to a preemption-restart cycle. We therefore also check on whether the execution order of Th if it were to abort would be lower than that of Tr. If the dynamic orders of Th both at the current time and were Th to abort are less than the dynamic order of Tr, then we check to see whether Th can be completed within the slack time of Tr. If so, then we keep Th running since we can be reasonably sure that we can

```
IF t + Estimate(Tr) - (t - Start(Tr)) > dr THEN
    keep Th running
ELSEIF (Order(Th) < Order(Tr)) AND
        (Order_Abort(Th) < Order(Tr)) THEN
    IF Estimate(Th) - (t - Start(Th)) <= Slack(Tr) THEN
        keep Th running
        block Tr
    ELSE
        delay Th
        run Tr
    END
ELSE
    block Tr
    keep Th running
END
```

Fig 36

finish this transaction and still have time in order to meet the deadline of Tr. Otherwise, Th is preempted for later restart and Tr gains control.

$$\textbf{Order(Ti) = a * Accuracy(Ti) - b * Slack(Ti) - c * Criticalness(Ti).}$$

Fig 37

As we have seen here, the scheduling decision for the transactions in the less critical group has been made while considering their time constraints. The temporal requirements and their implications considered were the deadlines, criticalness, and accuracy of transactions.

5. Conclusions

We have seen that in general, a database system adapting a multiversion concurrency control algorithm performs better while processing read requests. Read requests that would be aborted in a database system with a single version of each data item due to time stamp conflicts may be successfully processed in a multiversion system using older versions of data items. Therefore, when the read requests dominate the transaction load, and there is ample opportunity for the abortion of read-only transactions due to conflicts with update transactions, a multiversion system performs better than its corresponding single version system. The relative sizes of the read and write sets of transactions is an important factor affecting the performance.

We observed that careful consideration must be given to the implementation constraints regarding the creation and maintenance of the multiple data versions. These constraints can degrade the system performance to the point that they offset any performance improvements that the multiple data versions may provide. While minimizing the implementation constraints, it is possible to implement a multiversion database system which never performs more poorly than its single version counterpart and provides a performance improvement in some cases.

Multiple data versions are of significant value for database systems in real-time environments for meeting the temporal constraints. Real-time requirements associated with both the transactions and the database have been studied. In order to meet the real-time goals of such a system, the scheduler must consider the real-time constraints prior to making a scheduling decision. We have proposed a scheduling scheme which incorporates the real-time requirements in its decisions in terms of transaction deadlines, accuracy, and criticalness. Our future plans include implementing this scheduler and incorporating it within our existing prototyping environment.

6. References

[1] Sang H. Son, Jeremiah Ratner, Chun-Hyon Chang; "Prototyping Environment for Distributed Database Systems: Functional Description"; *University of Virginia Computer Science Department Technical Report IPC-TR-88-009*; October 5, 1988

[2] Wen-Te K. Lin, Jerry Nolte; "Basic Time-Stamp, Multiple Version Time-Stamp, and Two-Phase Locking"; *Ninth International Conference on VLDS*, 1983

[3] Michael J. Carey, Waleed A. Muhanna; "The Performance of Multiversion Concurrency Control Algorithms"; *ACM Transactions on Computer Systems*; November 1986, 338-378

[4] Walter H. Kohler, Bao-Chyvan Jeng; "Performance Evaluation of Integrated Concurrency Control and Recovery Algorithms Using a Distributed Transaction Processing Testbed"; *IEEE Transactions on Computers*, September 1986

[5] Sang H. Son, Yumi Kim; "A Prototyping Environment for Distributed Database Systems"; *University of Virginia Computer Science Department Technical Report TR-88-20; August 11, 1988*

[6] Bayer R., Heller H., Reiser A.; "Parallelism and Recovery in Database Systems"; ACM Transactions on Database Systems; June 1980, 139-156

[7] Stearns R., Rosenkrantz D.; "Distributed Database Concurrency Control Using Before-Values"; *Proceedings of the ACM SIGMOD International Conference on Management of Data*; April 29, 1981

[8] Chan A., Gray R.; "Implementing Distributed Read-Only Transactions"; *IEEE Transactions on Software Engineering*; February 1985

[9] Robinson J.; "Design of Concurrency Controls for Transaction Processing Systems"; *Ph.D. Dissertation, Carnegie-Mellon University Department of Computer Science*; 1982

[10] Carey M.; "Multiple Versions and the Performance of Optimistic Concurrency Control"; *Tech. Report 517, University of Wisconsin-Madison Computer Science Department*; October 1983

[11] Bernstein, Hadzilacos, Goodman; "Concurrency Control and Recovery in Database Systems"; *Addison-Wesley*; 1987

[12] Bernstein, Goodman; "Multiversion Concurrency Control - Theory and Algorithms"; *ACM Transactions on Database Systems*; December 1983; 465-483

[13] Papadimitrion C., Kanelakis P.; "On Concurrency Control by Multiple Versions"; *ACM Transactions on Database Systems*; March 1984; 89-99

[14] Peinl P., Renter A.; "Empirical Comparison of Database Concurrency Control Schemes"; *Proceedings of the 9th International Conference on VLDB*; 1983

[15] Gray J., Homan P., Obermarck R., Korth H.; "A Straw Man Analysis of Probability of Waiting and Deadlock"; *IBM Research Report RJ3066*; San Jose; 1981.

[16] Stankovic J.; "Real-Time Computing Systems: The Next Generation"; *COINS Technical Report 88-06*; Department of Computer and Information Science; University of Massachusetts; Amherst.

[17] Robert Abbott, Hector Garcia-Molina; "Scheduling Real-Time Transactions: a Performance Evaluation"; *Proceedings of the 14th VLDB Conference*; Los Angeles; 1988.

[18] Susan B. Davidson, Aaron Watters; "Partial Computation in Real-Time Database Systems"; *Department of Computer and Information Science, University of Pennsylvania*; March 1988.