Design, Implementation and Performance of a Real-Time Version of a Commercial RDBMS

Rohan.F.M. Aranha, Venkatesh Ganti+, Srinivasa Narayanan+*
*C.R. Muthukrishnan**, S.T.S. Prasad++, Krithi Ramamritham##* [1]

∗ Stanford University, + University of Wisconsin,
∗∗ Indian Institute of Technology, Madras, ++ HCL-HP, Madras, India
University of Massachusetts (krithi@cs.umass.edu)

Abstract

A real-time database system is a database system in which transactions have explicit timing constraints such as deadlines. Apart from satisfying the database consistency constraints, transaction executions must also satisfy these timing constraints. The goal of transaction and query processing in real-time databases is to maximise the number of successful transactions in the system.

This paper reports on the design, development and performance evaluation of RT-Genesis, a real-time database management system resulting from modifying an existing commercial DBMS, Genesis. RT-Genesis is a relational database management system that accommodates SQL queries and transactions having time constraints. It features time-cognizant algorithms for scheduling, concurrency control and buffer management. The system has been tested and the performance of the different algorithms compared in isolation as well as in combination with different classes of SQL workloads. In addition, a two-phased approach to transaction execution has also been implemented with the goal of exploiting access invariance to improve predictability.

This work demonstrates the feasibility of converting non-real-time DBMSs into real-time DBMSs for firm deadline transactions. Besides reporting on how this was achieved and on the performance of the incorporated algorithms, lessons learned from this experience are also discussed.

[1]This work was supported, in part, by NSF under grant IRI-9208920.

1 Introduction

A real-time database system is a database system in which transactions have explicit timing constraints such as deadlines. Apart from satisfying the database consistency constraints, these transactions must also satisfy these timing constraints. The goal of transaction and query processing in real-time databases is to maximise the number of successful transactions in the system.

This paper reports on the development of a real-time database, RT-Genesis, resulting from modifying an existing commercial DBMS, Genesis. This necessitated

- priority-based transaction execution – where the priorities reflect the deadlines of transactions and are under the sole control of the DBMS, and
- deadline-cognizant resolution of conflicts over data and other resources – where the transaction that wins in a conflict is chosen based on deadline considerations.

This paper reports on the design, development and performance evaluation of RT-Genesis, a real-time database management system resulting from modifying an existing commercial DBMS, Genesis. RT-Genesis is a relational database management system that accommodates SQL queries and transactions having time constraints. It features time-cognizant algorithms for scheduling, concurrency control and buffer management. The system has been tested and the performance of the different algorithms compared in isolation as well as in combination with different classes of SQL workloads. In addition, a two-phased approach to transaction execution has also been implemented with the goal of exploiting access invariance to improve predictability.

In our transaction model, the timing constraints take the form of *firm* deadlines, i.e., a transaction imparts no value to the system once its deadline expires. This work demonstrates the feasibility of converting non-real-time DBMSs into real-time DBMSs for firm deadline transactions. Besides reporting on how this was achieved and on the performance of the incorporated algorithms, lessons learned from this experience are also discussed.

The rest of the paper is structured as follows. After reviewing the major ingredients of transaction processing in Genesis, we outline the issues involved in converting it to manage transactions with time constraints. The performance evaluation strategy for RT-Genesis is then presented. The focus of the next three sections is on priority assignment, buffer management, and locking protocols. Their integrated performance is then studied. We also discuss the implementation and performance of a two-phase execution of transactions aimed at improving performance. The paper concludes with a discussion of the lessons learned from this work.

2 An Overview of Genesis

Genesis is a centralized secondary storage database management system. Transactions maintain data consistency. Serialization is the criterion employed for correctness. This is achieved by a strict 2-phase locking mechanism. Transactions are issued by clients who can be on remote machines. At any point of time, each transaction has a server serving it. A client requests a daemon process to create a server for it. Thereafter the client communicates with its server directly through a message passing mechanism. Each client submits transactions serially to the server, i.e., at any point of time each client has atmost one transaction being processed. There can be more than one client giving rise to contention for various resources like the CPU, locks and buffers. The server for each client takes care of all the contention and executes the transaction maintaining data consistency. A buffer, a log and a lock table are maintained in shared memory common to all servers. The buffer contains the data read from the disk by the SQL servers. The log consists of the operations done by committed transactions. Flushing the log writes it onto the log file on the disk. The buffer is partitioned into pages each 4KB in size. A page typically holds a block read from a data file. The number of pages in the buffer is a tunable parameter, which is set depending on the application requirements.

When a transaction is submitted to an SQL server, it is parsed and optimised to generate an execution plan. The transaction would typically access data items belonging to different object types in the database, where the term object type refers to a table, i.e., a relation, or an index. For each accessed object type, a page set is allocated. When the plan is executed, data blocks for the objects are accessed and the page sets for the transaction get filled. A maximum size and page replacement policy is identified for each of the page sets, depending on the type of object and predicted reference pattern. This is done using the DBMIN algorithm [6]. Whenever a transaction writes a data item it is copied from the buffer onto its local address space if it is not already present. The local memory where the data modified is kept are called *intentions.* All writes are done only to the intentions and intentions are moved to the buffers at transaction commit time. So the buffer contains only committed data. At commit point the following events occur:

- the server writes the operations to the log,
- the server informs the log flush process,
- the log flush process writes the log buffer to the log file on disk,
- the log flush process informs the server, and
- the server writes its intentions to buffer.

A checkpoint process periodically flushes the dirty buffer pages (which contain committed data) onto disk.

The above implies a *no steal, no-force* buffer management policy.

During execution, when the query objects need to be accessed, a transaction tries to get a free buffer page for the objects. This attempt will fail if all the buffer pages are used by the transactions executing currently (i.e., are "pinned" – to be released when the transactions terminate). In such cases, we have contention over the buffer resource. Contention can also arise during commit time when a transaction tries to acquire free buffer pages to write the intentions into. In both cases, the transaction times out, i.e., aborts, after a duration which is of the order of the execution time of a transaction.

Since we did not want to completely overhaul the underlying DBMS, the design of RT-Genesis was constrained by the above aspects of Genesis.

3 Issues in Building a RTDBMS

Deadlines associated with transactions differentiate a real-time database system from a normal database system. The SQL interface must be extended to specify the deadlines. The unit of the real-time database system which recognises the transactions assigns priorities to transactions based on their deadlines. The buffer manager and lock manager use these priorities to resolve contentions in a time cognizant manner so that resources are judiciously allocated especially under overloads and indefinite waits are avoided. This section elaborates on these issues.

3.1 Specifying Transaction Deadlines

We have modified the SQL interface to incorporate real-time transactions. In case of real-time transactions, the usual SQL statements for executing a transaction are preceded by the following statement which sets the deadline for the transaction:

set transaction deadline d;

3.2 Scheduling Transactions with Deadlines

Scheduling of transactions in RTDBs should take into account the deadlines of transactions. We must be able to execute a particular transaction before another because, say, it has an earlier deadline. We need to control the order in which all runnable transactions execute. So we need to assign deadline-based priorities to transactions. Our database platform, namely Genesis, runs on a System V Unix Operating System. We implemented real-time priorities using the handles provided by the *Real-time class* of SVR-4.2. It provides a real-time class; the operating system does not change the priority of a process in a real-time class unless requested by the user. We implemented different *earliest deadline first*-based scheduling policies and evaluated their performance with different types of transaction workloads.

3.3 Time Cognizant Buffer management

Data items which are accessed by transactions are first brought into buffer pages before they are read or written. Contention for buffer pages arises when the available buffer space is insufficient to accommodate all the pages needed by all the transactions. One of the important issues in transaction processing in real-time databases is the method of managing the database (volatile) buffer resource among the various transactions. We have developed and implemented different policies for managing the buffer resource and evaluated the performance of these policies under different real-time transaction workloads. These policies are constrained by the no-force, no-steal buffer management policy adopted by Genesis.

3.4 Time Cognizant Lock Management

Concurrency control is the activity of coordinating the actions of processes that operate in parallel, access shared data and therefore potentially interfere with each other. When two or more transactions execute concurrently, their database operations execute in an interleaved manner. That is, operations from one transaction may execute in between two operations from another transaction. This interleaving may cause transactions to behave incorrectly or interfere, thereby leading to an inconsistent database. The goal of concurrency control is to avoid this interference. If the result of the concurrent execution is the same as that of a serial execution, that is, if the execution is a *serializable* one then we have achieved correctness in spite of concurrency. The concurrency control mechanism in Genesis is based on strict 2-phase locking and therefore all our work in concurrency control is restricted to tailoring the 2-phase locking protocol for real-time behavior.

4 RT-Genesis and its Performance

As mentioned earlier, the objective of our real-time database system is to maximize the number of transactions that complete within their deadlines. Transactions that miss their deadlines are of zero value to the system. Our transactions have firm-deadlines. But, as we explain later, they may still stay in the system until they complete execution or are aborted by some other transaction during conflict resolution.

It is important to test the algorithms under different real-time workloads and contention situations. How this was achieved is explained in this section.

In the absence of a real application to test with and given our intention to evaluate under different loads, we developed a synthetic database and SQL transactions. A major consideration was to keep the database objects as well as the transactions simple enough that we could control and explain the behavior of the policies and of the database system.

The tests were performed on a test database which was created and populated to have 20 tables. Ten of them (call them *type 1*) have 100 records each and another ten (*type 2*) have 10 records each.

We classify transactions into 2 classes based on the number of records they access. Each transaction in Class 1 accesses 100 records and has an execution time of 100 ms. This transaction accesses all the records of a type 1 table. Each transaction in Class 2 accesses 1000 records and has an execution time of 1000 ms. These transactions *join*ed type 1 and type 2 tables.

As the number of data items accessed by concurrent transactions increases the lock conflict probability also increases. Because of this, the higher class transactions are likely to have more conflicts while accessing the required data items.

The deadline, d, for a transaction T is set as $d = (1+\alpha) \times x$ where α is a random number between 0 and 5 and x is the warm start execution time of transaction T when there are no contentions.

A transaction which completes its execution before the deadline is deemed to be successful. Thus transactions which are aborted as well as transactions which have executed past their deadline are not considered successful.

The success ratio is defined as follows.

$$success\ ratio = \frac{no.\ of\ successful\ transactions}{total\ no.\ of\ transactions\ spawned\ by\ all\ clients}$$

We measured the percentage of successful transactions against the *Multiprogramming level*(MPL). MPL is maintained constant by having a fixed number of clients issuing a continuous stream of transactions. (To achieve reasonable performance, we had to curtail the MPL to be less than or equal to 10.)

A successfully completed transaction will lead to a new transaction being issued immediately by the client to the same server.

Whenever a transaction aborts, the corresponding SQL-server and the client are killed by Genesis. Restarting this client and a server is time consuming in Unix and hence for that duration, the required MPL will not be satisfied. Hence we took a different approach. We had many clients and SQL servers created and remaining dormant. When a transaction is aborted, the SQL-server corresponding to the aborted transaction awakens another SQL server by sending a message to that SQL server which in turn starts a transaction. If the aborted transaction's deadline has not yet expired, that transaction is restarted. Otherwise, a new transaction is started. The time for activating a dormant server is less than a millisecond which is negligible in our scenario. Nevertheless, it may make our results appear worse compared to an an automatic restart policy which does not involve this overhead.

All the performance evaluation studies were done by maintaining a continuous stream of transactions for a sufficiently long duration to obtain results whose 90% confidence intervals have half-widths of 5%.

In the next three sections we examine the policies implemented for priority assignment, lock conflict resolution, and buffer conflict resolution, respectively. Subsequently, we study their combined performance.

5 Scheduling

In this section, we first list the scheduling policies implemented and then discuss their performance on RT-Genesis.

5.1 Scheduling Policies

The earliest-deadline-first policy, according to which the transaction with the earliest deadline is assigned the highest priority, is favored in real-time systems. This policy has been the cornerstone of many real-time systems and in the non-overloaded case, it has been proved to be optimal. We have implemented a dynamic earliest deadline first policy as well as two static policies, the latter inspired by those proposed in [2].

The scheduling policies involve mapping deadlines linearly onto the priorities. They involve a time interval $[0, \infty)$ divided into $(max - 1)$ equal intervals of length t_s and a final interval $[(max - 1)t_s, \infty)$ of a different length.

Given ct, the current time, $a(T)$, the arrival time of a transaction, and $d(T)$, the absolute deadline of T, the function map_{lin} makes this priority assignment. The idea is to take a parameter (some function of the absolute deadline like $d(T) - ct$) and see which interval it falls into. Each interval is associated with a priority. The priority value associated with the interval $[kt_s, (k+1)t_s)$ is $(max - k)$, $(0 \leq k \leq (max - 1))$. The last interval has the value 1. So if the absolute deadlines of two transactions differ by more than t_s they are assigned different priorities.

The value of t_s is fixed and chosen before-hand depending on how close we want the deadlines of two transactions to be for them to have the same priority. In our system t_s is chosen to produce a total of 60(=max) intervals, 60 being the number of real-time priority levels available in the operating system. For example, for tests with Class 1 transactions, we fixed t_s to be 10 ms. and for Class 2 tests, we fixed it to be 100 ms. so that we could spread out the priorities assigned to transactions. If we had fixed t_s at the same value for both class 1 and class 2 (say 100), then while running tests for class 1, all these transaction priorities will be clustered at the higher end.

5.1.1 Dynamic algorithm (DA)

Whenever a new transaction arrives, priorities are assigned to the transactions such that the earlier the deadline, the higher the priority. Priorities are computed relative to the current time and their deadlines, i.e.,

$$priority(T) = map_{lin}(d(T) - ct).$$

We need to (re)compute transaction priorities whenever a new transaction arrives. In practice, priorities have to be recomputed only for those transactions which have higher priorities than the newly arriving transaction.

This algorithm, being dynamic, can involve substantial overheads for the priority assignment. So our next two algorithms are static, that is, once assigned, the priority of a transaction is not changed subsequently when other transactions arrive.

5.1.2 Earliest deadline relative (EDREL)

Here, the priority just depends on the relative deadline of T:

$$priority(T) = map_{lin}(d(T) - a(T)).$$

The problem with this algorithm is that transactions which are close to completion may miss their deadlines because of the arrival of transactions with tighter relative but later deadlines.

5.1.3 Earliest deadline absolute (EDABS)

This is also a static scheduling algorithm. Instead of using relative deadlines, it uses absolute deadlines and uses the notion of a busy interval – a period of time throughout which there is at least one transaction in the system. Let t_{begin} denote the beginning of the latest busy

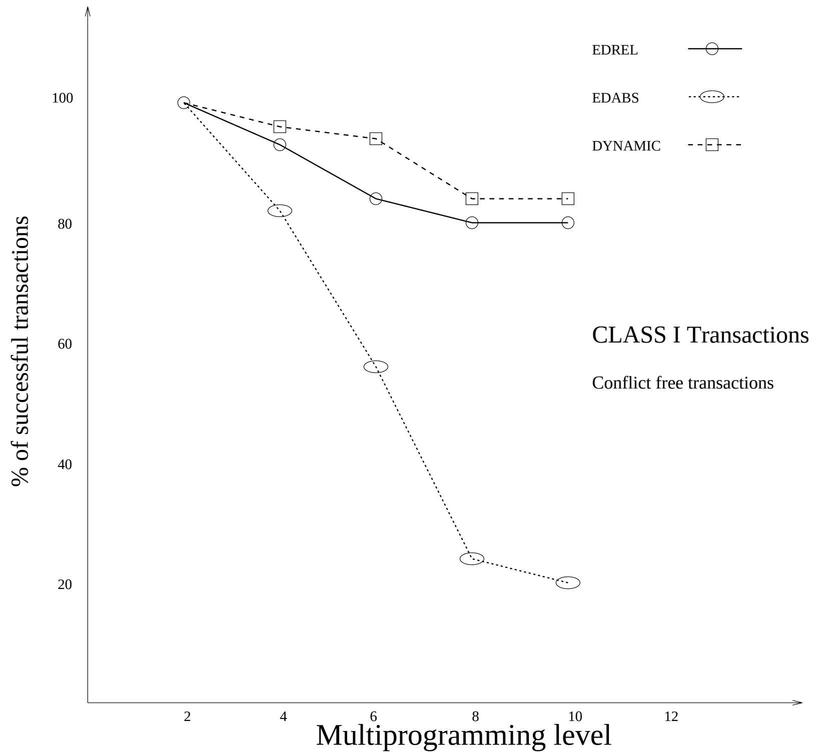

Figure 1: Performance of scheduling policies for class 1 transactions

interval in the system.

$$priority(T) = map_{lin}(d(T) - t_{begin}).$$

A possible shortcoming of this algorithm is that if the busy interval is too long then newly arriving transactions will be clustered at the lowest priority levels.

5.2 Performance Evaluation

The test load consists of read-only transactions thus avoiding lock conflicts. The number of buffer pages is kept sufficiently high to avoid buffer contention. Thus the influence of lock and buffer management algorithms is avoided in the test load.

The results are shown in Figures 1 and 2 for the two classes of transactions.

5.2.1 Analysis

The dynamic algorithm is observed to perform the best for all three classes. It assigns priorities strictly in the earliest deadline first fashion. The transaction with an earlier deadline has a higher priority. This has been proved to be optimal [4]. So the performance of the dynamic scheduling algorithm is better than the static scheduling algorithms. The overhead of recomputation of priorities for every new arrival is not very large in our case since at most 10 transactions were concurrently in the system at any given time in our system. Also, DA

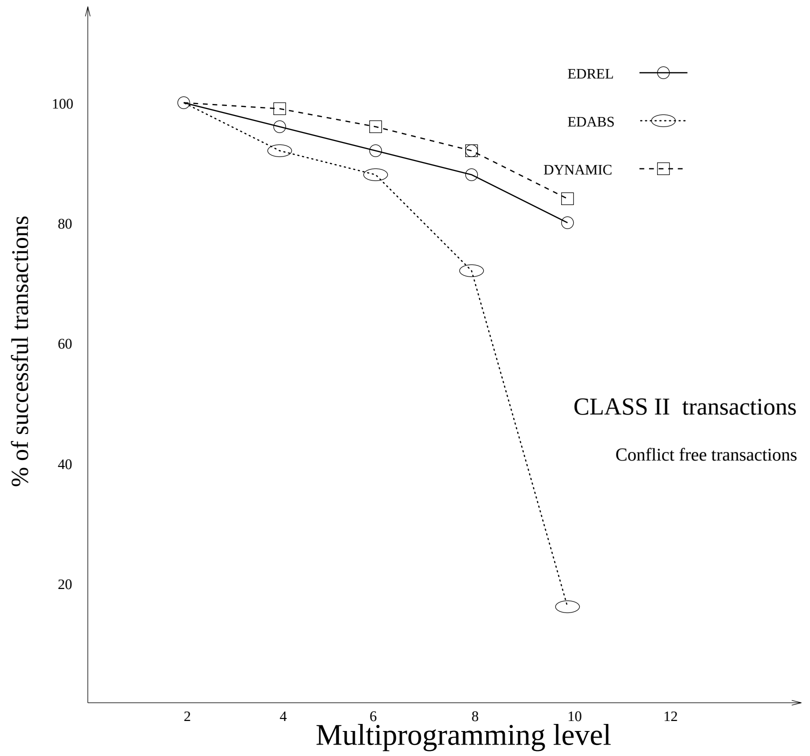

Figure 2: Performance of scheduling policies for class 2 transactions

does not reassign priorities to all transactions. EDREL is better than EDABS. The performance for EDABS falls more steeply with MPL because the length of the busy interval increases very fast at higher MPL and the priorities of later transactions are clustered at the lowest priority. At this point scheduling is independent of the deadline and so the bad performance. The steepness of the fall for EDABS increases with class because the rate of increase in the length of the busy interval increases with class.

6 Buffer Management

When one transaction needs a free buffer page but each page has been "pinned" by some running transaction we need some policies for resolving this contention over the buffer space.

Due to the priority based approach, where transactions are assigned priorities based on their deadlines, a number of issues arise when a transaction requests a resource, such as a buffer page, which another transaction holds.

If a transaction with a higher priority is forced to wait for a lower priority transaction to release a resource, a situation known as *priority inversion* arises. This is because a lower-priority transaction makes a higher priority transaction wait. Since the deadline has a direct bearing on the priority this will make it even more difficult for a transaction with an urgent deadline to acquire a resource which a transaction with a much later deadline holds. This may lead a situation where the transaction with the urgent deadline fails to complete in time while the transaction which held the resource finishes much ahead of its deadline.

In cases where a transaction with an urgent deadline has to wait for a transaction with a much farther deadline (and therefore much less priority) it would be a good idea to abort the lower priority transaction, thereby allowing it to release its resources which may then be used immediately by the waiting high priority transaction. Usually, the abortion of a transaction introduces another important issue which is the *recovery* of the aborted transaction. A considerable overhead may go into aborting a transaction. The resources it utilizes to follow its normal course may be fewer than if it is aborted. Our database uses intentions and so all the updates are done in the local address space of the transaction and are reflected in the global buffer only at commit time. Therefore the abortion procedure need not undo any updates and hence the cost of abortion is small.

Instead of aborting a transaction to avoid priority inversion, we can resort to *priority inheritance*[8, 13]. In this case, we increase the priority of the resource holding transaction to that of the transaction which is waiting for the resource. This inheritance is done for the duration that the transaction with higher priority waits for the lower priority transaction to release the resource. This enables it to finish early and in turn enables the higher priority transaction also to finish early since its waiting time will be reduced. However, even with this policy, the higher priority transaction is blocked, in the worst case, for the duration of the transaction's execution time.

We first outline the implemented policies and discuss their performance.

6.1 Buffer Management policies

When there is buffer contention, we can either make the requesting transaction wait or abort one of the page holding transactions. The latter forces a page to be freed. As we shall see, a third option is also available. The three policies considered in our study are described below.

6.1.1 Non real-time buffer manager (NRT)

This does not take into consideration the priorities of the transactions. When a transaction does not find a free buffer page it just waits until it gets a free page (which is released by some other transaction) and times out after a duration which is of the order of the execution time of the transaction.

6.1.2 Abort policy (BA)

When a high priority transaction, T, finds that it has no free buffer page it aborts a lower priority transaction. Upon abortion a transaction releases all the pages held by it which will now be available for use by the waiting transaction. The lower priority transaction chosen is the one which holds the maximum number of buffer pages among all the lower priority transactions.

It could happen that a high priority transaction, T, after aborting a lower priority transaction, still does not find a free buffer page. This is because a new transaction with priority higher than T could have entered the system and taken the pages released by the lower priority transaction. In this case T may have to abort another lower priority transaction.

So a transaction could abort multiple transactions to get a free buffer page. However, we set a limit on the number of transactions that could be aborted by a transaction needing a free buffer page. In our experiments, this limit was set to 4.

6.1.3 Priority inheritance (BI)

When a high priority transaction, T, finds that it has no free buffer page, it chooses a lower priority transaction and increases the priority of that transaction to its own priority. With the increased priority, the transaction holding buffer pages will complete faster and release the pages sooner. The lower priority transaction chosen is the one which is closest to its deadline among all lower priority transactions. This is because the transaction closest to its deadline is more likely to complete earlier than others. Here, multiple transactions can inherit the higher priority. To see how multiple transactions can inherit higher priority consider the the following scenario: Transaction A raises the priority of transaction B and enters the wait queue. Now B is not able to find a free page and this in turn could raise the priority of another transaction C and so on.

6.2 Performance evaluation

Because of its superior performance, the dynamic scheduling policy was used in all the experiments. Transactions were designed such that no lock conflicts existed. This was done to isolate the effect of the buffer management policies.

The amount of contention over the buffer space is measured as follows: Let N = Total number of buffer pages. Let $T_1, T_2, ...T_k$ be the k transactions running concurrently in the system. k is the multiprogramming level. Let the buffer page requirement of transaction T_i be b_i, i.e., T_i requires b_i pages for execution. This is determined by executing the transaction in isolation and measuring the maximum number of buffer pages used by it. The contention factor f is defined as

$$f = \frac{\sum_{i=1}^{k} b_i}{N}.$$

A value of f greater than 1 implies a potential for contention over the buffers.

To keep the focus on buffer contention, the test transactions are read only class 2 transactions, performing a join between a *type 1* table and a *type 2* table. Each of the transactions accesses a different set of tables and hence the data items accessed by the transactions are disjoint. Each of the test transactions requires eight buffer pages for execution. The contention factor f was varied by varying the total number of buffer pages in the system. The multi-programming level was set at 5. The results are shown in Figure 3.

6.2.1 Analysis

For values of f from 0 to 1, there is no contention over the buffer space. Since we used the dynamic scheduling policy and all our transactions were of class 2 executed at a MPL of 5, the percentage of successful transactions is close to 100. This can be confirmed from the

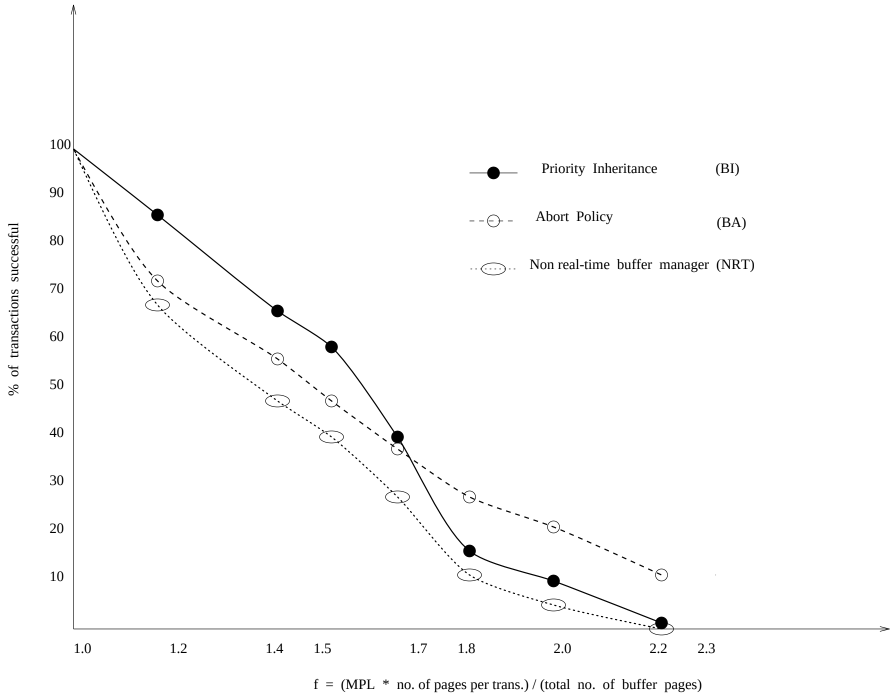

Figure 3: Performance of buffer management policies

value for MPL 5 shown in Figure 2 where the value is nearly 100%. As the contention factor f increases, the percentage of successful transactions falls. This is because some transactions are forced to wait and this causes them to miss their deadlines. Also some transactions get aborted when there is contention, if the policy uses aborts.

Both the real-time buffer manager policies BA and BI perform better than NRT, the non real-time buffer manager. At low values of f (1.0 - 1.6) there is a large difference between the performances of BI and NRT. At higher values of f there is a large difference between the performance of BA and NRT. BA and BI have a crossover point in their relative performance.

BI is found to perform better than BA when the contention is low. This is because, under low contention the number of waiting transactions is small. BI will help in making the resource holders (which are more in number) release the resources faster. The wait time for the transactions is hence small and the number of successful transactions is high. When the load is such that the resource requirements are only slightly more than the existing resources it pays to be patient and wait until someone releases the resources. Since BA blindly aborts transactions without waiting, its performance is poorer than BI. On the other hand under high contention the wait time will be larger and hence more transactions will miss their deadlines with BI. BA is found to be better under such conditions since it aborts, early, transactions that will miss their deadlines anyway due to severe buffer contention.

With the above observations in hand we can say that a buffer management policy that makes its decisions based on the level of resource contention is desirable, choosing an inheritance based policy when it is low and an abort based policy when it is high.

Work done in the area of real-time buffer management includes [3] and [9]. Whereas [9] reports of no significant performance improvements when time-cognizant buffer management policies are used, studies discussed in [3] show that transaction priorities must be considered in buffer management. In this study we see that policies taking the priorities of transactions into account do perform better than those which do not. This corroborates the results from [3].

7 Lock Management

Similar to the buffer case, we can have inheritance and abort based protocols. Since it has already been established that time cognizant protocols are essential, we do not compare with a non real-time protocol. But we do compare the two with another that chooses between abort and inheritance adaptively.

In all the protocols that we consider, at lock release time, the waiting transactions are assigned locks in order of priority. That is, the transaction with the highest priority is assigned locks first. Since the deadlines have a direct bearing on the priorities, this is the right approach. A transaction with higher priority is preferred and hence can try to finish faster.

7.1 Locking Policies

Three data conflict resolution protocols have been implemented and compared. All three protocols have the same structure, given by the following pseudo-code.

T_R *requests a lock on the data item* D
if *no conflict with* T_H
    then T_R *accesses* D
    else
        if $priority(T_R) \leq priority(T_H)$
            then T_R *waits*
            else
                abort-or-inherit
        end if
end if

Each protocol is derived by refining the statement
abort-or-inherit.

7.1.1 Priority Inheritance Protocol(LI)

In this protocol [13], whenever a transaction conflicts with another transaction over a data item, that is, when a transaction requests a lock which another transaction is holding, the lock is not immediately granted to the requesting transaction. If the priority of the lock requesting transaction is less than or equal to the priority of the lock holding transaction, we do nothing and the requesting transaction simply waits. If however the priority of the waiting transaction is larger than that of the holding transaction, the priority of the holding transaction is raised to that of the waiting transaction. That is, the lock holding transaction inherits the priority of the lock requesting transaction.

This inheritance is in effect only for the duration that the lock holding transaction holds onto the lock. In the case of our database, as we have mentioned earlier strict two-phase locking is used. Therefore the priority inheritance exists till the end of the transaction's execution time.

Abort-or-inherit is refined as:

begin
$priority(T_H) := priority(T_R)$
T_R *waits*
end

7.1.2 Abort-based Protocol(LA)

In this protocol whenever a transaction conflicts with another transaction over a data item, that is, when a transaction requests a lock which another transaction is holding, if the priority of the requesting transaction is less than or equal to the priority of the holding transaction, we do nothing and the requesting transaction simply waits. If however the priority of the waiting transaction is larger than that of the holding transaction, we abort the lock holding transaction. That is, the lock holding transaction is aborted because of its lower priority with respect to the lock requesting transaction.

Abort-or-inherit is refined as:

T_R *aborts* T_H.

7.1.3 Lock-Count-based Protocol(LC)

This protocol involves a combination of transaction aborts and priority inheritance. We will see later that at low conflict levels, the priority inheritance algorithm(LI) performs well and under high conflict level, the abort-based(LA) algorithm performs better. In order to improve performance even more, we could dynamically rather than a priori decide whether to abort a transaction or to inherit priorities. This dynamic decision could be based on the parameters of the conflicting transactions such as the work left, slack time, work done etc. An algorithm which is based on the amount of work left has been proposed in [8] and shown to work better than the others.

Our algorithm is based on the amount of work the conflicting transactions have done. Since, for our transactions, it was not possible to infer the percentage of its execution a transaction has completed, we base our decision on the amount of useful work a transaction has already done. The latter is *inferred* from the the number of locks it holds. Since, we base our algorithm on the number of locks the conflicting transactions hold rather than on the execution time left, this protocol turns out to be slightly different from the one in [8]. If we assume that the longer a transaction has been executing the less work it has left, this protocol would be a reasonable approximation to the one in [8].

To be more precise, in the event of conflict over a lock, if the priority of the requesting transaction is less than or equal to the priority of the lock holding transaction, the requesting transaction waits. If the priority of the requesting transaction is greater than that of the holding transaction, then the action to be taken depends on the number of locks each of them possesses. If the lock holder has a larger number of locks than the lock requesting transaction, inheritance is used, that is, the lock holder inherits the priority of the requester, else the lower priority lock holding transaction is aborted.

Abort-or-inherit is refined in this case as:

begin
  if $\#locks(T_H) > \#locks(T_R)$
    then $priority(T_H) := priority(T_R)$
    else T_R *aborts* T_H
end

7.2 Performance Evaluation

The graphs in this section were based on tests carried out using the dynamic scheduling algorithm (the best of the three scheduling algorithms), and adequate pages were allocated in order to avoid conflicts for buffer pages. Thus, all performance differences can be attributed to differences in the lock management policies.

The read/write ratio for transactions in class 1 is fixed at 3:1 and at 1:1 for the transactions in class 2. The read/write behavior of the data was designed such that when a transaction accesses a data item for write, at most one transaction has a read lock on it.

The transactions in class 1 are of shorter duration than those in class 2. Also, the read/write ratio of the transactions in class 1 is more than that of the transactions in class 2. Consequently, the level of conflicts in the system in class 1 is much less than the conflict level in class 2. As a result of this the relative performance of the locking protocols differs across the classes.

The performance of the different protocols is shown in Figures 4 and 5. (These figures also contain the graphs for the two-phase protocol discussed in Section 8.)

7.2.1 Analysis

In Figure 4, priority inheritance (LI) outperforms the other two protocols. Since the level of conflict is low, LA which tends to abort a transaction at every lock conflict, aborts too many transactions. All the successful transactions tend to finish much ahead of their deadlines. LI on the other hand does not abort any transaction but schedules the transactions in an intelligent manner, which enables most of the transactions to complete before their deadlines. What we mean is that the blocking times for locks, for the transactions with urgent deadlines and hence higher priority, is reduced due to priority inheritance. Thus priority inversion is avoided in this case. Under conditions of low conflict, which imply that the sustainable concurrency of the system is high, LA under-utilizes the system resources by the unnecessary abort of many transactions.

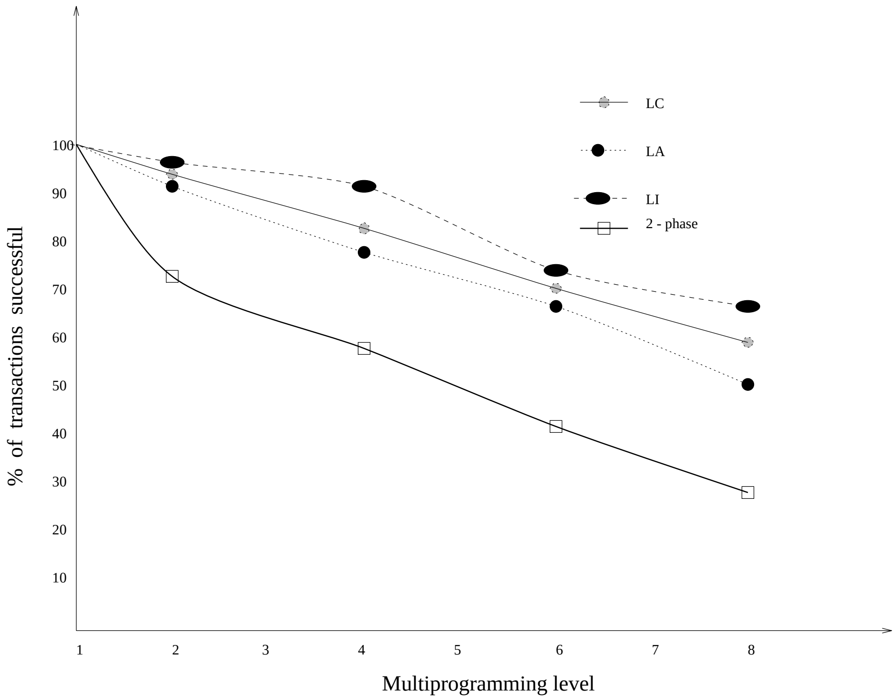

Figure 4: Performance of locking policies for class 1 transactions

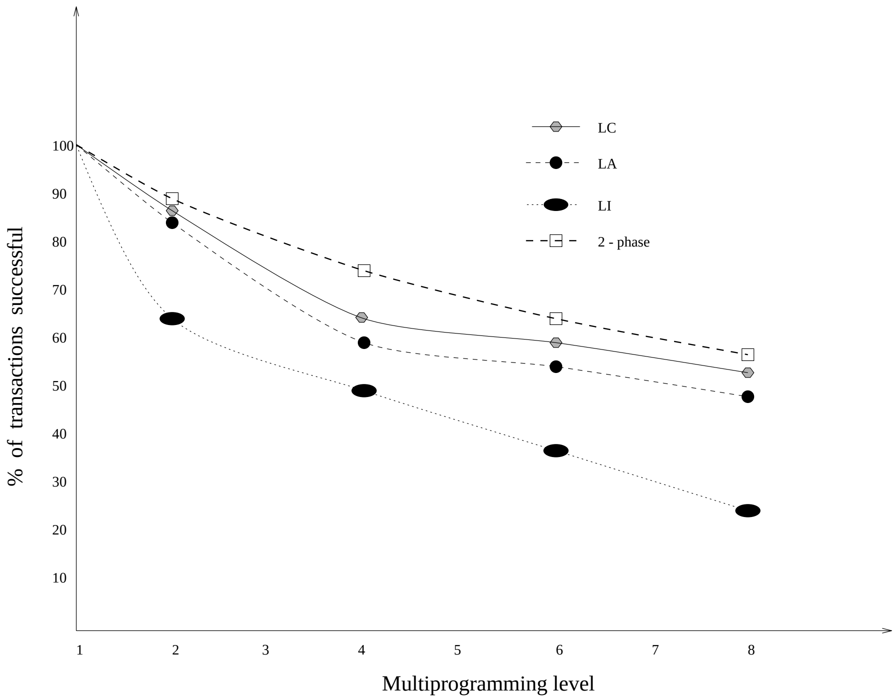

Figure 5: Performance of locking policies for class 2 transactions

In the case of class 1 transactions, LC performs better than LA but worse than LI. From this we can infer that although LC does not abort as often as LA, it still aborts too many transactions.

In the case of class 2 transactions as shown in Figure 5 LC comes on top. In this case due to the high conflict levels, the sustainable concurrency of the system is greatly reduced. Due to this reduced concurrency (caused by many transactions being blocked, sometimes all), any attempt made at the scheduling of all transactions in the system, leads to almost all of them missing their deadlines. By aborting certain transactions, the attempt to over-utilize resources is avoided and the resources are better utilized by the transactions left in the system. Thus the success rate in this case improves with the abort-based policies LA and LC.

The improved performance of LC over LA is due to the use of sound heuristics to decide whether or not to abort. The abortion of transactions which have acquired more locks than the lock requesting transaction is always avoided. Thus, some of the unnecessary aborts of transactions close to completion are avoided.

8 Integrated Performance Evaluation

Having individually studied the performance of the three classes of algorithms, we are in a position to study their combined performance effects. Here we evaluate the performance of the real-time database system with lock and buffer contention under the dynamic scheduling algorithm. Unlike the isolated performance studies, the test load here is associated with both lock conflicts and buffer conflicts.

Since the dynamic scheduling algorithm outperformed the static scheduling algorithms under all the conditions that we considered as shown earlier, we consider only the dynamic scheduling algorithm for integrated performance. We have used different combinations of locking and buffer policies and compared their performance. The policies tested include combinations involving inheritance(BI) and abort(BA) policies for buffers and inheritance(LI), abort(LA) and the lock count(LC) policy for locks. So, a policy of inheritance for buffers and abort for locks would be denoted as BILA. The other policies are denoted similarly.

Here again, we classify transactions into two classes, 1 and 2, based on the execution time and the contention for locks and buffers. For buffer we use the contention factor f explained earlier. For locks we vary the read/write ratio. For class 1 transactions the read/write ratio is set to be 3:1. The buffer conflict factor f, explained earlier, is set to around 1.15 for all MPL by appropriately varying the total number of buffer pages for each MPL. For class 2 transactions the read/write ratio for each transaction is maintained at 1:1. The buffer conflict factor f is set to around 1.5 by appropriately varying the total number of buffer pages in the shared memory. Given this, class 2 transactions will encounter more conflicting situations than class 1.

For these values of f, from the results on buffer protocols, we expect BI to work better. So we combined BI with the three locking protocols. To make sure that our decision to focus on BI is right, we also compared with a completely abort based protocol, namely BALA.

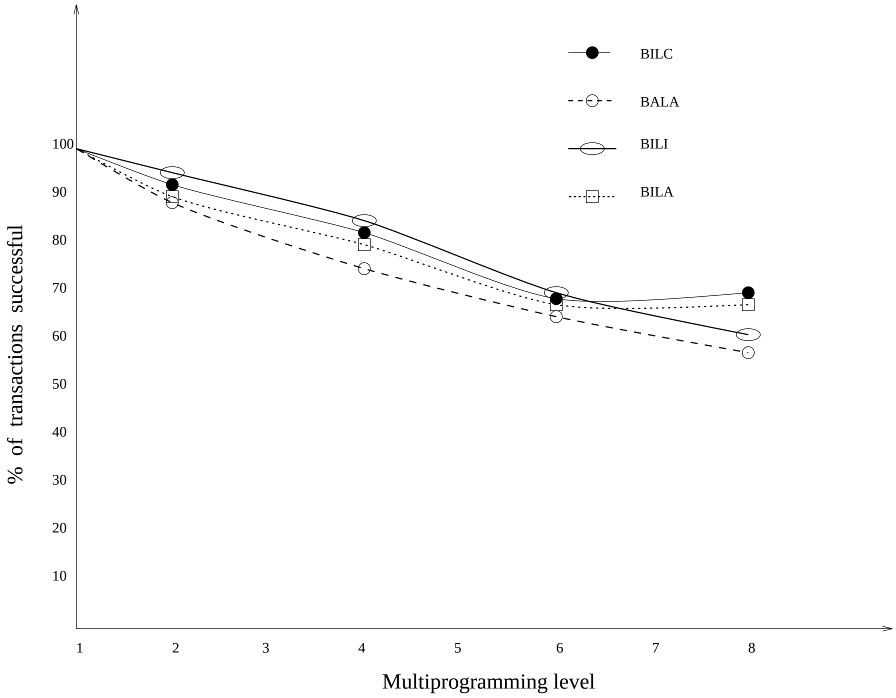

Figure 6: Integrated Performance of Class 1 Transactions

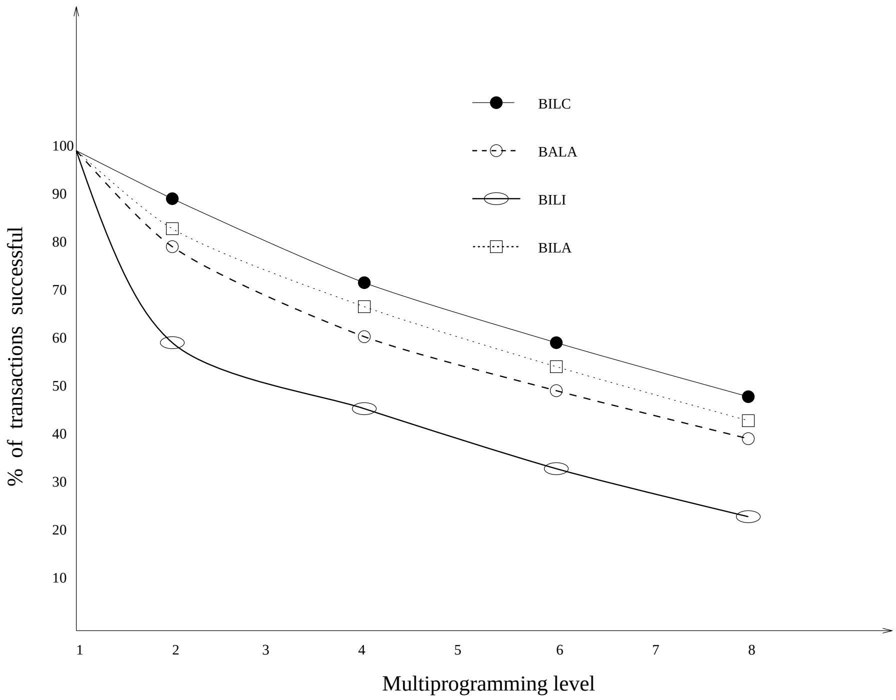

Figure 7: Integrated Performance of Class 2 Transactions

8.1 Analysis

For class 1, priority inheritance is observed to perform better than abort-based algorithms as seen in Figure 6. Raising the priority of a blocking transaction causes it to complete earlier without causing further priority increases and delays in most cases. This may lead to the successful completion of the blocked and the blocking transactions leading to better performance. This is true especially at low MPL's. At MPL of 8, corresponding to an overloaded condition, the blocked transactions may not be successful and hence the performance falls. In this case, as shown by the slightly better performance of BILC and BILA, a small amount of forced abortions helps. However, at low loads BILC aborts transactions needlessly at times and thus has worse performance than the BILI algorithm. Thus in this case, the BILI algorithm performs the best.

For class 2, lock conflict resolution algorithms that induce some aborts when combined with BI are observed to outperform priority inheritance as shown in Figure 7. The abort-based algorithms release valuable resources facilitating successful completion of more transactions. The policy of inheritance for buffers and aborts for locks (BILA) is better than both BILI and BALA for class 2 because at the buffer conflict level 1.5, inheritance is the best as shown in Figure 3 and for locks abort policy is better than inheritance at the data conflict levels in class 2. The BILI algorithm in this case attempts to over-utilize system resources.

While the BILA algorithm makes better use of the resources than the BALA algorithm, the proposed BILC algorithm outperforms even the BILA algorithm. Since BILC bases its decision to abort on the number of locks held by the conflicting transactions, a few unnecessary aborts of transactions nearing completion are avoided.

Summarizing the results of the integrated tests, we can say that when the conflict level is low, inheritance is the best policy. When the conflicts are high, it is better to forcibly reduce the probability of conflicts by aborting some of the transactions. Between these two extremes when one needs to judiciously decide which transaction to abort. A decision based on which transaction has done more work, such as with LC, has good payoffs.

9 Two Phase Transaction Execution

We have also implemented a two-phased approach to transaction execution [11, 5]. The motivation for this approach is to decrease the effect of lock conflicts when combined with I/O delays. Specifically, because of the delay involved in bringing the data into the buffer, a transaction holds its locks for a longer duration, resulting in increased conflicts.

We briefly describe the implemented scheme here. Note that this is a much simplified version of the protocol detailed in [11].

In the first phase, the transaction is run once bringing the data into main memory if it is not already present. No writes are performed in this phase and no locks are held, i.e., conflicts with other transactions are ignored. Reads are performed as usual but no writes are performed. In the second phase the transaction is run again, this time acquiring locks and performing writes.

The second phase of the transaction's execution is begun after the first phase, with the transaction holding the locks as usual. But, this second phase is not begun if the time left before the deadline is less than the warm start execution time of the transaction. Assuming that the buffer is large enough to hold the data needed by the concurrently running transactions, data required by the transaction will be in memory at the end of the first phase and so a second run of the transaction is unlikely to face I/O. Thus, although the second phase is just like the execution of a transaction in a non-two-phase environment, the lack of disk latency will reduce the lock holding time and hence increase the concurrency of the system.

In our tests, the number of buffer pages has been kept sufficiently high to avoid buffer contention. The two-phase execution algorithm is tested on both class 1 and class 2 transactions. The performance evaluation for two-phase execution is shown in Figures 4 and 5.

9.1 Analysis

For class 1, two phase execution performs worse than priority inheritance. The execution times of transactions and lock conflicts are low and so the slight reduction in lock-holding time does not improve the performance. In two phase execution it is virtually equivalent to running each transaction twice within the same deadline.

But for class 2, two-phase execution is observed to outperform abort-based algorithms and priority inheritance. Here the competition is between two-phase execution and abort-based algorithms. In the abort-based algorithms transactions which are aborted eventually will hold onto locks till their abortion, preventing the effective use of resources. But in two-phase execution, the number of aborts in the second phase is observed to be very low compared to that in the first phase. So the number of locks which are released by aborting transactions in two-phase execution is much less. Most of the aborts due to scheduling delays are caused immediately after the first phase. Since locks are held by transactions only in the second phase, the time for which some locks are held is reduced. This reduction in lock-holding time is significant only for Class 2 as the lock conflicts are high. Also, transactions waiting to acquire locks will be waiting on transactions which are in their second phase. At any given time, the number of transactions in their second phase will be less than the MPL (it will be around MPL/2). So the effective MPL as far as lock-contention is concerned is much less for two-phase execution. This reduction in MPL for lock-contention is significant only for Class 2 as the lock conflict levels are high and not for Class 1. So lock-holding is more efficient for two-phase execution. The probability that a transaction which has entered its second phase completes successfully is very high.

The above study shows that the two-phase approach can be recommended when we have long transactions accessing many data items or encountering many conflicts. In other cases, the overheads of executing a transaction twice may overwhelm the performance improvement due to reduced lock holding times.

10 Lessons Learned from Building a RTDBMS

All four major components of a transaction processing system, namely, the scheduler, the transaction manager, the buffer manager, and the lock manager need to have hooks to make decisions that are priority-cognizant. In this section we summarize the needed facilities and describe how we managed when they were not readily available.

10.1 The Scheduler

The scheduler should provide hooks to dynamically change priorities and to execute real-time transactions at priorities that are higher than transactions without deadlines. Fortunately, the real-time features provided by System V in the form of the real-time class is very convenient to schedule real-time transactions by adjusting their priorities, and to run real-time transactions without being preempted by non-real-time transactions. However, this means that the system processes also have a lower priority than real-time transactions which, as we explain later, makes it tricky to abort transactions immediately upon the expiry of their deadlines.

10.2 Transaction Manager

The transaction manager must support efficient and timely transaction aborts and restarts. In fact, perhaps the biggest limitation we faced was the lack of a proper and timely transaction abort mechanism: for aborting transactions that have missed their deadlines and for restarting transactions – that are aborted for releasing resources – if their deadlines have not been missed.

In our implementation, transactions that miss their deadlines are not aborted immediately, neither do all such transactions run to completion. Such transactions are aborted only when the resources needed by the transaction become necessary for another transaction. This is achieved as follows: (1) When a transaction arrives the priority of any transaction which has missed its deadline is set to the lowest value. This ensures that transactions which have missed their deadlines do not stand in the way of other transactions' use of the CPU. (2) Whenever there is a conflict for a buffer/lock, the system first checks if one of the conflicting transactions has missed its deadline. If so, that transaction is aborted. If not, abort or inheritance policies are used to resolve the conflicts. This policy ensures that buffer and lock resources are not consumed unnecessarily. The two together prevent a transaction which missed its deadline from consuming resources which otherwise could be used by other transactions.

The reason that the abort is not immediately effected is as follows: The main issue is to find a low cost means to detect missed deadline transactions. In our implementation, we detected missed deadlines upon the occurrence of the following events: (1) arrival of a new transaction – whereupon the scheduler is invoked; (2) when contention arises for buffer/lock resources. This implies that CPU time may be consumed unnecessarily because a transaction priority is not lowered (or the transaction is not aborted) until event 1 (or event 2) occurs.

We considered two alternatives:

1) Abort a transaction as soon as its deadline expires by having an alarm for each transaction which rings when the deadline is missed.

2) Have a periodic process which checks the deadlines of transactions at periodic intervals and aborts transactions which missed their deadlines. As in the method we did implement there will be a delay between when a deadline expires and when it actually is aborted and so we did not consider this method further. Our method has the advantage that abort/restart overheads are incurred only when the resources held by a delayed transaction are needed by another.

However, there are difficulties in realizing either alternative. Since real-time processes are not preempted, all system processes are scheduled only if the real-time processes are blocked. Thus, any periodic time check to abort transactions which have crossed their deadlines will have to be done as part of some real-time process. Also, the scheduler is a system process and by adjusting the priorities we only control the scheduling policy. The mechanism is still in the operating system. In order to have absolute control over our real-time processes we will have to have more control over the scheduling mechanism.

Another difficulty we faced was with regard to effecting the actual abortion and restart of a transaction. In our database system, associated with each running transaction was an SQL-server. In order to abort a transaction we had to kill the server. To restart a transaction a new server had to be created. In a Unix-based operating system, this has the potential to have very large costs. We overcame the problem in the following tortuous way. We had many clients and SQL servers created and remaining dormant. When a transaction was aborted, the SQL-server corresponding to the aborted transaction awakened another SQL server by sending a message to that SQL server which in turn started executing the transaction. This server as well as the SQL server of the aborted transaction execute the same SQL transaction code. This obviously added to the overheads and to the latency of transaction aborts.

10.3 Lock Manager

The lock manager must allow for conflict resolution based on transaction characteristics such as deadlines, priorities, etc. It was easy to convert the locking mechanism of Genesis. Conflict resolution using transaction priorities was also easy to realize. We introduced a lock count with each transaction to implement the LC protocol in a straight-forward way.

10.4 Buffer Manager

Buffer management plays an important part in achieving deadline cognizant resource management, in particular, in taking buffers away from low priority transactions and allocating them to those with higher priorities. Unfortunately, in Genesis, the buffer manager structure was difficult to change. In particular, if we had been able to implement Steal buffer policies we need not have aborted transactions to get a free page. We would have also liked to experiment by varying the page set sizes for transactions dynamically – depending on the priority. Recall that when the execution plan for a transaction is generated, the page set size is fixed. The transactions do replacements within their page set when conflicts for pages arise. If we had a method by which the page set size could be varied dynamically, then we could increase the page set size of a high priority transaction to avoid replacing some of the pages it is using currently when a conflict arises. This way, we could try to eliminate conflicts within the page sets of high priority transactions.

11 Conclusions

We have implemented RT-Genesis, a fully operational RT-DBMS with an SQL interface which has been suitably modified, to execute real-time and non real-time transactions. We implemented and made a comparative study of various algorithms used in real-time databases especially for issues like scheduling, buffer management and lock management. We have studied the isolated performance and integrated performance of these algorithms.

- The dynamic priority assignment algorithm is observed to perform better than static algorithms for priority assignment under all conditions for firm-deadline transactions, each having a unit value if successful and zero, otherwise. Among the two static algorithms, EDREL is better than EDABS under all conditions.

- The buffer management performance evaluation confirms the necessity of a real-time buffer manager which performed better than the non real-time buffer manager. The priority inheritance policy performs better than the abort policy when the contention is low and the abort policy performs better than the priority inheritance policy when the contention is high.

- The newly proposed Lock Count (LC) protocol performed better than both the abort protocol(LA) and the inheritance protocol (LI) under conditions involving high data conflict. At lower conflict levels, inheritance(LI) is the best.

- The integrated performance of these algorithms was also studied under different conditions. BILI is found to have the best performance under conditions involving low buffer and lock conflicts. BILC is preferable when more conflicts occur.

- Two-phase transaction execution was implemented and its performance shown to be better than the other algorithms under high conflict conditions.

Based on the experience gained from building this real-time version of a commercial DBMS we can say that that:

- It is feasible to convert a non-real-time DBMS into a Real-Time DMBS as long as the underlying operating system allows for the proper assignment and management of priorities. In our case, the operating system was System V Unix.

- A very simple change is needed in the SQL interface to specify the deadlines.

- It is important to have a repertoire of time cognizant priority assignment, locking, and buffer management algorithms for meeting the needs of time-constrained transactions. A monolithic policy does not suffice.

- Whether to abort a conflicting transaction, or to allow a resource holding transaction to inherit a higher priority depends on the load on the system and the level of conflicts. As one moves from light to medium to heavy loads/conflicts, the choice moves from always-inherit, to sometimes-inherit, to always abort policies. In the second instance, the choice of which transaction to abort and when to resort to priority inheritance must be made judiciously, as in the case of LC, a new protocol.

With regard to the last two items, in some sense, the conclusions are not startling. Similar observations have been made in earlier simulation-based studies (see [1, 10, 14, 15], for example) or the rarer implementation-based study [9]. Thus, in terms of performance, the actual performance trends in our results were similar to those in the literature, even though if one were to run our transactions in simulation mode the overheads assumed may affect the actual performance figures. Whereas scheduling overheads incurred by our algorithms were not large, they were not negligible. As we mention in the paper, our abort overheads were not high even though we did have message passing delays that delayed restarts.

What is significant here is our use of a commercial relational DBMS. These are not usually built keeping experimentation in mind. The "hooks" that are needed – to facilitate switching between algorithms or incorporating new algorithms – are absent. Still, we had to design the RT-Genesis system to do precisely this. Also, since we wanted to maintain compatibility with GENESIS we were constrained in several ways. For instance, (1) once a transaction is aborted, the server corresponding to that transaction gets killed. Hence restart of an aborted transaction with the same deadline and without considerable delay was difficult; (2) all our buffer management policies had to work within the confines of those built into Genesis. Thus, for example, when a buffer page was unavailable, it was not possible to preempt a running transaction, use a single page used by the transaction, and later allow it to resume. But for such constraints, we would have liked to study other protocols, e.g., priority-LRU [3], for buffer management.

References

[1] Abbott R. and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems,* pp. 513–560, Sept. 1992.

[2] B Adelberg, H. Garcia-Molina, B. Kao. Emulating Soft Real-Time Scheduling using Traditional Operating System Schedulers, *Real-Time Systems Symposium*, pp. 292-298, Dec 1994.

[3] M.J. Carey, R. Jauhari and M. Livny. Priority in DBMS Resource Scheduling, *Proceedings of the VLDB*, pp. 397-410, 1990.

[4] M. Dertouzos, Control Robotics: The Procedural Control of Physical Processes. *Proceedings of the IFIP Congress*, 1974.

[5] Franaszek P. A., J. T. Robinson, and A. Thomasian. Concurrency Control for High Contention Environments. *ACM Transactions on Database Systems*, pp. 304–345, June 1992.

[6] H-T Chou and D. J. DeWitt. An Evaluation of Buffer Management Strategies for Relational Database Systems, *Proceedings of the VLDB*, 1985.

[7] J. Huang, J. A. Stankovic, D. Towsley. Krithi Ramamritham Real-Time Transaction Processing : Design, Implementation and Performance Evaluation, *COINS 90-43*, Univ. of Massachusetts, May 1990.

[8] J. Huang, J.A. Stankovic, K. Ramamritham, D. Towsley and B. Purimetla. On Using Priority Inheritance in Real-Time Databases, *Real-Time Systems Journal*, Vol 4, No. 3, pp. 243-268, 1992.

[9] J. Huang and J. Stankovic, Real-time buffer management. *COINS TR 90-65*, 1990.

[10] J. Haritsa, M. Carey, M. Livny. Data access Scheduling in Firm Real-Time Database Systems, *The Journal of Real-Time Systems*, 4, pp. 203-241, 1992.

[11] P. O'Neil, K. Ramamritham, and C. Pu, A Two-Phase Approach to Predictably Scheduling Real-Time Transactions, in *Performance of Concurrency Control Mechanisms in centralized Database Systems*, V. Kumar, Ed. Prentice-Hall, pp. 494-522, 1995.

[12] K. Ramamritham, Real-time databases, *Distributed and Parallel Databases*, Vol 1, pp. 199-226, 1993.

[13] L. Sha, R. Rajkumar and J. Lehoczsky. Priority inheritance protocols: an approach to real-time synchronization, *IEEE Transactions on Computers*, 39, pp. 1175 - 1185, 1990.

[14] S. .H. Son, Y. Lin, and R. P. Cook, Concurrency Control in Real-Time Database Systems, in *Foundations of Real-Time Computing: Scheduling and Resource Management*, edited by Andre van Tilborg and Gary Koob, Kluwer Academic Publishers, pp. 185-202, 1991.

[15] O. Ulusoy and G.G. Belford, Real-Time Transaction Scheduling in Database Systems, *Information Systems*, Vol 18, No. 8, pp. 559-580, 1993.