TOWARDS PREDICTABLE TRANSACTION EXECUTIONS IN REAL-TIME DATABASE SYSTEMS

P.E. O'NEIL, K. RAMAMRITHAM, C. PU

COINS Technical Report 92-35
May 1992

Towards Predictable Transaction Executions in Real-Time Database Systems

Patrick E. O'Neil
Dept. of Math & Comp. Sc.
Univ. of Massachusetts
Boston, MA 02125

Krithi Ramamritham[1]
Dept. of Computer Sc.
Univ. of Massachusetts
Amherst, MA 01003

Calton Pu[2]
Dept. of Computer Sc.
Columbia University
New York, NY 10027

Abstract

Even though considerable research has been done on concurrency control protocols that take the timing constraints of transactions into account, most of these protocols do not predict – before a transaction begins execution – whether the transaction will meet its deadline. Hence the protocols do not directly tackle the problems introduced by the various sources of unpredictability encountered in typical database systems, including data dependence of transaction execution, data and resource conflicts, dynamic paging in virtual memory systems, disk I/O, and transaction aborts with the resulting rollbacks and restarts. On the other hand, the approach described in this paper has the potential to provide predictable transaction executions.

This approach exploits the *access invariance* property that many transactions possess or can be designed to possess. The execution path of a transaction with this property is unlikely to change as a result of data modifications by other concurrent transactions. In this approach, a transaction goes through two phases. In the first phase, called the *prefetch* phase, a transaction is run once, bringing the necessary data into main memory that is not in memory already. No writes are performed in this phase and data conflicts with other transactions are not considered. The overall computational demands of the transaction are also determined during this phase. At the end of the prefetch phase, the system attempts to guarantee that the transaction will complete by its deadline. This is done by planning the execution of the transaction – taking into account data and resource conflicts with the transactions already guaranteed – such that the transaction meets its deadline. If such a plan can be constructed, the transaction's *execution phase* begins and if access invariance holds then the transaction is committed at the end of this phase. A number of variations on this approach (such as using optimistic concurrency control in the prefetch phase or handling occasional failures of access invariance) are also investigated. The benefits of the approach are also discussed.

[1]partially supported by the National Science Foundation under grant IRI-9109210.
[2]partially supported by NSF, IBM, DEC, AT&T, Oki Electric Ind. and Texas Instruments.

Contents

1 Introduction

Many real-world activities involve time constrained access to data. For example, consider the following: looking up the "800 directory", radar tracking and recognition of objects and determining appropriate response, as well as the automatic tracking and directing of objects on a conveyor belt. Recently, attempts have been made to apply the database notion of transactions to model such activities. (Transactions have ACID properties, i.e., Atomicity, Consistency, Isolation, and Durability.) Meeting transaction timing constraints in a *predictable* manner demands new approaches to data and transaction management. Predictability refers to the ability to know whether a transaction will be able to meet its deadline so that, otherwise, appropriate corrective action can be taken before the deadline. This paper considers the predictable processing of dynamically arriving transactions that have deadlines. It assumes that transactions operate on a disk-resident database, where only a small fraction of accessed data can be cached in memory.

Although considerable research has been done recently on processing transactions with time constraints [1, 2, 4, 10, 13, 14, 15, 16, 20] for the most part, this work has not directly tackled the problems introduced by the various sources of unpredictability (discussed in Section 2). Whereas concurrency control protocols have been proposed to execute transactions taking their time constraints into account, most of them do not predict – before a transaction begins execution and starts making changes to the data – whether the transaction will meet its deadline. (Because of this, transactions may be aborted when their deadlines expire and this in turn can have adverse effects on other ongoing transactions.) Perhaps the only exceptions are the ideas discussed in [21] and [4]. Both consider transactions whose execution times as well as data requirements are assumed to be known a priori. Reference [21] considers periodic transactions operating on a main memory database. In [4], an approach is suggested where the system creates schedules that take into account the data requirements of contending transactions. Unfortunately, not all applications can predeclare data requirements, since in many cases transaction executions are data dependent. Secondly, main memory database systems limit the size of the database. Finally, periodicity constraints are just one type of timing

constraints.

On the other hand, our approach has the potential to deal with the many sources of unpredictability encountered in typical database systems. This approach exploits the *access invariance* property that many transactions possess or can be designed to possess. The execution path of a transaction with this property is unlikely to change as a result of data modifications by other concurrent transactions.

In this approach, a transaction goes through two phases. In the first phase, called the *prefetch* phase, a transaction is run once, bringing the necessary data into main memory that is not in memory already. No writes are performed and data conflicts with other transactions are not considered in this phase. The overall computational requirements of the transaction are also determined during this phase. At the end of the prefetch phase, the system attempts to guarantee that the transaction will complete by its deadline. This is done by planning the execution of the transaction – taking into account data and resource conflicts with the transactions already guaranteed – such that the transaction meets its deadline. If such a plan can be constructed, the transaction's *execution phase* begins and if access invariance holds then the transaction is committed at the end of this phase.

Section 3 presents the details of the approach and then points out how it overcomes the problems introduced by the various sources of unpredictability. This section also discusses how many transactions can be designed to possess the access invariance property. How to deal with those transactions that do not possess access invariance is also discussed. Scheduling related issues are the subject of Section 4. Section 5 presents several extensions to the approach. A summary of the paper and ideas for future work are presented in Section 6.

2 Sources of Unpredictability in Transaction Executions

A number of factors contribute to the unpredictability of transaction executions. They include

- data dependence of transaction execution,
- data and resource conflicts,
- data oriented disk I/O, and
- transaction aborts and the resulting rollbacks and restarts.

Below we elaborate upon these.

Since a transaction's execution path can depend on the state of the data items it accesses, it may not be possible in general to predict the behavior of a transaction in advance as can be done under the assumption of predeclared data Read/Write sets.

Since a typical transaction dynamically acquires the data items it needs, it may be forced to wait until a data item is released by other transactions currently using it. Consider a database that employs strict two phase locking for concurrency control. In this case, a transaction may wait, in the worst case for an unbounded amount of time, when it attempts to acquire a data item. Similarly, a transaction may be forced to wait for resources, such as CPU and I/O devices.

Whereas unpredictable variations in behavior due to virtual memory can be easily dealt with by using real memory for buffered data and time-critical logic, in the case of disk-resident databases, delays can still occur when fetching data from disk. Such unpredictable variations can lead to very pessimistic worst-case assumptions. Such variations will also depend on the disk scheduling and buffer management algorithms used.

Turning to the effect of transaction aborts, consider two phase locking, where data is read in from disk and locked, then locks are held over a relatively extended period while

further I/Os are performed to access subsequent data for the transaction. The extended period of locking increases the likelihood of an abort. The same consideration applies to optimistic concurrency control methods, in that transactions executing over an extended period are unlikely to succeed. Transaction aborts have two negative implications. The total execution time for the transaction increases and, with naive design, the number of aborts can be unbounded. Secondly, the resources and time needed to handle the rollbacks necessitated by transaction aborts imply that ongoing transactions may be affected.

Even assuming that one can quantify in advance the computation time as well as data and resource needs of transactions, the use of these worst-case estimates implies a very pessimistic approach to processing transactions. This is inappropriate and will lead to poor transaction performance because the worst case scenarios, especially in the presence of I/O, are extremely improbable.

3 Our Approach

We first discuss the basic approach assuming that access invariance of transactions holds. We then show how with some help from the compiler and by proper design and coding of transactions we can attempt to ensure access invariance and, where this is not possible, detect the violation of access invariance. Finally we discuss the advantages of our approach. Scheduling related issues are presented in Section 4.

3.1 The Basic Idea Assuming Access Invariance

In our approach, a typical transaction goes through two phases, the prefetch phase and the execution phase. In the *prefetch* phase, a transaction is run once, bringing the necessary data into main memory that is not in memory already. No writes are performed in this phase and conflicts with other transactions are not considered. The overall computational demands of the transaction are also determined during this phase. (Computation-intensive parts of the transactions, such as match finding algorithms that relate two patterns in vision systems as well as risk analysis in financial applications [23], need not be run during

the access phase as long as (1) the execution of these parts is not necessary for determining the subsequent data requirements of the transactions, and (2) the execution times of these portions can be estimated a priori.)

Let us first assume that the data dependent portions of the transactions are such that a transaction's execution path, and hence its data and computational requirements, are unlikely to change due to possible concurrent changes to the data by other transactions during the time between prefetch and subsequent execution. This is a property known as *access invariance*, which is investigated, for instance, in [8]. Thus we assume, at the end of the prefetch phase, that all the necessary data is in memory. At this point, the system attempts to guarantee that the transaction will complete by its deadline. This is done by planning the execution of the transaction – taking into account data and resource conflicts with other transactions already guaranteed – such that the transaction meets its deadline. Basically, resource conflicts are resolved at guarantee time by scheduling transactions to execute in different time intervals if they contend for the same data item. That is, resource contention is handled so that transaction performance with respect to deadlines is predictable, rather than letting resource contention occur at arbitrary points in time resulting in an unpredictable system. If such a plan cannot be constructed, the system aborts the transaction. Otherwise, the transaction's *execution phase* begins. The transaction is committed at the end of the execution phase. The notion of guarantee and the scheduling algorithm are based on the resource constrained scheduling approach proposed for real-time systems [19]. Salient aspects of the algorithm are summarized in Section 4. It should be pointed out that even though a transaction is guaranteed with respect to its worst case computation time and data requirements, it is possible to reclaim the time allocated to a transaction's execution phase should it terminate early [22].

Now let us examine some of the details of the approach. In the next subsection, we study the ramifications of the access invariance property.

The normal execution of a transaction can be modeled as consisting of alternating periods of CPU execution and I/O access denoted as follows:

CPU_1, IO_1, CPU_2, IO_2, CPU_3, IO_3, ...

Any IO_k will usually consist of a single disk page access; however, it may represent a burst of several such page accesses. The page(s) referenced by IO_k might already be buffer resident but often will have to be retrieved from disk. Each CPU_k period can be factored into a number of different types of activities, and we wish to identify activities that must be performed during the prefetch phase, activities that must be performed during the execution phase, and activities that must be performed during both phases. We identify three types of CPU use.

- *CPU-rw*: Transaction logic to evaluate expressions that determine the data read/write set. This is part of the prefetch phase.
- *CPU-comp*: Pure computations. Do not involve any *CPU-rw* type activities. Typically, *CPU-comp* applies to the execution phase.
- *CPU-SYS*: System logic needed during the prefetch phase, e.g., for query planning, and during the execution phase, e.g., commit processing.

Access invariance, when logically guaranteed, implies that the values for the expressions that determine the read/write set don't change between the prefetch phase and the execution phase. In this case, we don't need to perform *CPU-rw* a second time during the execution phase. Similarly, if portions of a transaction's execution involve *CPU*-intensive computations none of which deal with *CPU-rw* type activities, then such computations constitute *CPU-comp*. They can be postponed till the execution phase. This reduces the CPU overheads of the fetch phase and furthermore reduces resource waste in case the execution phase is not pursued. As mentioned earlier, match finding algorithms that relate two patterns, correlation schemes used in weather forecasting, etc. can be modeled as *CPU-comp* type activities.

We believe that it should be relatively easy to write applications, perhaps with some specialized compiler aids, that support the distinction between *CPU-rw* and *CPU-comp*. Expressions can be identified during compilation that determine which data items are in the read/write set, and logic to evaluate these expressions are placed in *CPU-rw*,

while logic to evaluate other expressions are placed in *CPU-comp*. Estimates of *CPU-comp* execution time can also be provided by the compiler.

CPU-SYS includes a number of significant CPU activities:

- Query plan creation to determine the access methods to use for the transaction, e.g., what index accesses to perform.
- Lookaside, to look for the desired page in memory buffer. After doing this in the prefetch phase we should be able to retain a pointer to the appropriate buffer for use in the execution phase, a practice known as *swizzling* in object-oriented databases [5].
- CPU overhead for physical I/Os, for example creating channel programs for reading and writing.
- Process switching. When a process becomes blocked because of a need for I/O in the prefetch phase, a process switch is required.
- Transaction Commitment, which is part of the execution phase. Log records are written as part of the commit process.

Based on the above, we can divide *CPU-SYS* into $CPU\text{-}SYS_p$ and $CPU\text{-}SYS_e$ corresponding to the CPU system overheads during the prefetch and execution phases. The optimal situation would be a small $CPU\text{-}SYS_p$ overhead so that we would be able to perform all I/O and then evaluate the feasibility of the transaction with only a small investment of resources. Unfortunately, this is not the case: Typically, $CPU\text{-}SYS_p$ will represent the largest share of *CPU-SYS*.

With the access invariance assumption, there is no need to perform the *CPU-rw* type computations in the execution phase and no need to perform *CPU-comp* type computations in the prefetch phase. Hence, assuming that the transaction was written according to the rules of 2PL (Two Phase Locking), a transaction execution will take the following form during the prefetch phase:

CPU-rw$_1$, *CPU-SYS*$_{p1}$, *IO*$_1$, ..., *CPU-rw*$_i$, *CPU-SYS*$_{pi}$, *IO*$_i$, ..., *CPU-rw*$_m$, *CPU-SYS*$_{pm}$, *IO*$_m$.

and the following pattern during the execution phase:

CPU-comp$_1$, ..., *CPU-comp*$_j$, ..., *CPU-comp*$_n$, *CPU-SYS*$_e$, *IO*

Each of the IO's in the prefetch phase reads in the pages/data items needed for subsequent computations. All the pages are released (as in strict 2PL) when the transaction commits. The implication of this is that during the execution phase as the transaction proceeds from one *CPU-comp* activity to the next, it uses more and more data items. All of these are released at the end of the execution phase. The *IO* at the end of the execution phase is for writing log records.

Given the above characterization of transaction processing, the following questions arise: How are the execution phases scheduled to execute? How are the prefetch phases executed? How are the two allowed to execute in a manner that enhances the performance of real-time transaction processing? These questions are answered in the section 4.

Several extensions are possible to this basic approach. For example, in some situations, there may not even be a need to go to the execution phase. Assuming that all parts of a transaction are executed during the prefetch phase, using an optimistic concurrency control approach during prefetch, this will be possible if the data items accessed by the transaction were not used by any other concurrent transaction. These extensions are discussed in Section 5.

3.2 Issues Related to Access Invariance

Given the many benefits of the access invariance property, it is clear that, if at all possible, one should attempt to write transaction code such that transactions will be access invariant. Below, we discuss some of the possible techniques. Clearly, it is necessary to detect violations of the access invariance property and recover from it. The main motivation for the latter is to ensure that those that possess access invariance are not jeopardized by those that do not. Hence, we discuss how one can detect and then deal with the violation

of access invariance in those applications where it cannot always be guaranteed.

Access invariance of a transaction T will not hold if (a) data conflicts occur between T and other transactions whose execution phases overlap the prefetch and execution phases of T *and* (b) the changes made to these data by the other transactions change the data and computational requirements of T. In our case, where the execution phases of transactions are planned so that a set of transactions execute without conflicts, the violation of access invariance is harmful only if the changes to the data and computational requirements of T are additive in nature, i.e., data items not accessed in the prefetch phase and/or more CPU time than initially assumed are needed for T.

First of all, it is worth noting that since most changes to data that are concurrently updatable are likely to be harmless, that is, not likely to result in additional data or computational needs for a transaction, a change does not immediately imply violation of access invariance. Also, a transaction's logic can be written to be *fail-fast*, so that when it sees a value (say, of a warehouse inventory that is insufficient to fill an order), the transaction itself completes earlier than planned, rather than taking some alternative path. Any alternative (such as, filling the order by checking the inventory in a different warehouse) can be achieved by a separately scheduled transaction. Finally, it should be mentioned that in those cases where a transaction might miss its deadline if its execution phase is interrupted due to concurrent changes to data that alter its path, a transaction might be in a position to trade-off quality for timeliness. That is, a transaction (such as, a query about the remaining inventory in a warehouse) might complete with the previous version of the data, by following the original path that was encountered during its prefetch phase. This again demands special coding of transaction logic but is necessary to achieve predictable executions.

Note that a transaction's execution path will change between prefetch and execution phases only if it takes alternative paths based on the value of some data item that is subject to change and in fact has changed. Of course, it can take alternative paths based on program variable arguments that were input to the transaction and are not subject to concurrent update. Thus, we can differentiate between data items that can be

updated concurrently and those that can't; Constraints can be placed on alternative logic paths in a transaction to *guarantee* access invariance. The idea of building a constraint, to be checked by the compiler, is to disallow logic that makes the path dependent on concurrently changing data. That is, we can read a data item based on a data item read earlier, only if the data is approved for that purpose, a data item that is not subject to concurrent update, for example, the record containing information about the a priori known characteristics of a warehouse, such as its capacity.

Let us consider another example, to execute a buy order in a stock trading system. The set of relevant sell order records, stock records, customer and agent records, as well as indexing information to locate these records must be referenced in this case. If concurrent trading transactions do not affect the buying transaction, its execution phase will complete successfully. Suppose concurrent buy orders are executed. If there are sufficient sell orders to satisfy these buy orders plus the original buy order, the original buy order will have access invariance and execute successfully. If, however, a concurrent buy order invalidates one of the few existing sell order records, then the original buy transaction will terminate without completing the buy. In the latter case, the transaction completes without accessing the data it encountered during its prefetch phase (that is, it has the fail-fast behavior mentioned above). Of course, a third possibility is that because of the nature of the concurrent sell and buy orders, the transaction takes a different path, with different I/O and computational requirements. In this case, the transaction will repeat its prefetch phase.

While it is always possible to construct pathological counter-examples, we maintain that program logic can usually be written to support access invariance, and only the occasional exceptions will remain as sources of unpredictable behavior. In any case, such exceptions can easily be detected, and can be handled as discussed below.

The basic approach outlined earlier is extended as follows: (1) During the execution phase, the transaction management system keeps track of the data and time used by a transaction T to check if T deviates from the guaranteed behavior. If more time is used or (a) any new items are used *and* (b) these data items are used by any other guaranteed

transaction's execution phase, then T's execution phase is interrupted and the rest of T executes as though the transaction was going through its prefetch phase; (2) When a transaction's execution phase is interrupted, it releases all the data items allocated to it and discards all the changes done to these data items. A variation of the scheme proposed in [7] can be used to accomplish this. In our case, a transaction makes changes only to copies of the data pages and discards the copies if its execution phase terminates prematurely.

Thus, in some cases, a transaction may go through the prefetch phase multiple times. Knowing the elapsed time for previous prefetch phases and the time required for its execution phase, a transaction should be terminated before restarting its prefetch or execution phases in case its deadline cannot be met.

3.3 Advantages of The Approach

Let us see how our approach tackles the four major sources of unpredictability mentioned at the beginning of Section 2.

- By using the prefetch phase to bring in the data pages, the data dependent requirements are determined during this phase.
- Data and resource conflicts during execution are avoided by the use of explicit planning of the execution phase of transactions.
- Since necessary pages are brought into memory during the prefetch phase, I/O is avoided during the execution phase, shortening the scheduled execution period and the duration of its exclusive access to needed data items.
- Finally, as long as access invariance holds, transaction aborts and rollbacks are avoided because all changes are done during the execution phase and this phase is not begun unless it is known that it will complete in time.

Note that in this approach we are saving an important system overhead for data locking and deadlock detection, present in most database systems. Correct transaction

executions, achieved via locking in traditional approaches, is achieved here via proper scheduling of transaction activities. A priori, it is not clear that such scheduling represents a greater overhead.

Note also that the work done in prefetching data from disk is in fact the minimum we must perform to achieve predictability. For transactions that perform data-dependent computations, reading in the pages as the transaction executes, subsequent computations and disk accesses of the transaction will not be known. Also, disk access delays constitute one of the primary sources of unpredictability of transaction executions. Naturally, the approach to I/O scheduling should also be deadline sensitive (see for example [6]).

We point out that our approach keeps track of deadlines during the prefetch phase and will abort a transaction if it is already past its deadline. This gives us an important advantage in situations where deadlines are much too small. (As mentioned earlier, we might be able to provide a contingency approach in this case, giving a low quality answer with the data already present.) Thus, if we find that we are going to miss our deadline because of I/O, we will at least have saved the CPU time needed to perform the computationally intensive parts of the transactional computations.

Finally, we should mention other advantages to our approach. By performing the guarantee calculation before the computationally intensive parts of the transaction are executed, we avoid wasting CPU resources. Attempts to guarantee the execution phases provides early notification that a deadline is likely to me missed. Early notification has *fault tolerance* implications in that it is now possible to run alternative error handling transactions early, before a deadline is missed.

Before we end this section, we give a simplified quantitative example to illustrate the benefits of our approach. Specifically, we show how a reasonable slack time for a transaction with I/O can become an enormous slack time after pages have become memory resident. If one were to guarantee in advance that failure to complete I/O in time is unlikely, because of pessimistic assumptions that must be made about I/O, a non-trivial multiplying factor must be applied to standalone I/O elapsed time. This will be true even in cases where I/O is not a bottleneck and *mean* I/O service time is less than twice the

standalone elapsed time. Thus, assuring with, say, 99% certainty that enough time is available for I/O will require I/O time that is at least three times the I/O service time. Thus, attempting to schedule, in advance, the I/O for a transaction will make even a relatively large slack time seem small. Assume (say) that elapsed time needed to execute I/O in a standalone situation is 9 times greater than CPU time required for a typical transaction. Assume too that the mean slack time for such transactions is 2 times the standalone elapsed time. Thus, with CPU time of 0.1 and the standalone I/O time of 0.9 such a transaction has 3.0 time units to complete. With a 99% certainty requirement for I/O, at least (3×0.9) must be allocated for I/O, a tight situation, given the overall time of 3.0. On the other hand, assuming negligible prefetch CPU costs, after the I/O is completed, the mean slack time for CPU will be the ratio of elapsed time remaining (on average, $3.0 - (2 \times 0.9) = 1.2$) to CPU time remaining (0.1), or a slack time that is 12 times the remaining expected elapsed time. This eases the scheduling of the execution phase of the transaction. In the unusual case that time to perform needed I/O actually exceeds the deadline, this fact will be apparent before the scheduling algorithm is applied to plan the execution phase.

4 Scheduling Issues

A transaction is guaranteed by constructing a plan for its execution phase, together with the execution phases of other transactions whereby all transactions are scheduled to execute so as to meet their timing constraints. The guarantee is subject to a set of assumptions, for example, about its worst case execution time, and data requirements. As we saw earlier, the prefetch phase determines the computational and data requirements of transactions. When a transaction is guaranteed, the scheduler attempts to plan a schedule for it and the previously guaranteed transactions so that all transactions can make their deadlines. In Section 4.1 we briefly discuss how the scheduler works. Further details can be found in [19].

Executing the prefetch phase and planning for transaction guarantees can be done in parallel with the execution of previously guaranteed transactions. How this can be

accomplished is discussed in Section 4.2.

4.1 Scheduling the Execution Phases of Transactions to Meet Deadlines

The planning algorithm attempts to schedule transactions in a non-preemptive fashion given their deadline T_D, worst case computation time T_C, and resource requirements $\{T_R\}$. Resources include the pages or data items accessed by a transaction as well as resources such as buffers and CPUs. Note that the algorithm described below works for a multiple CPU system; a single CPU is a simple instance of this. A transaction uses a resource either in shared mode or in exclusive mode and holds it from time t_b to t_e. t_b can be thought of as the earliest time when a transaction would obtain its lock for the data item and t_e is the latest time when the data would be unlocked. The guarantee algorithm computes the earliest start time T_{est}, at which transaction T can begin execution.

The heuristic scheduling algorithm tries to determine a full feasible schedule for a set of transactions in the following way. It starts at the root of the search tree which is an empty schedule and tries to extend the schedule (with one more transaction) by moving to one of the vertices at the next level in the search tree until a full feasible schedule is derived. To this end, we use a heuristic function, H, which synthesizes various characteristics of transactions affecting real-time scheduling decisions to actively direct the scheduling to a plausible path. The heuristic function, H, is applied to at most k transactions that remain to be scheduled at each level of search. The transaction with the smallest value of function H is selected to extend the current schedule.

While extending the partial schedule at each level of search, the algorithm determines if the current partial schedule is *strongly-feasible* or not. A partial feasible schedule is said to be *strongly-feasible* if *all* the schedules obtained by extending this current schedule with any one of the remaining transactions are also feasible. Thus, if a partial feasible schedule is found not to be *strongly-feasible* because, say, transaction T misses its deadline when the current schedule is extended by T, then it is appropriate to stop the search since none of the future extensions involving transaction T will meet its deadline. In this case,

a set of transactions can not be scheduled given the current partial schedule. (In the terminology of branch-and-bound techniques, the search path represented by the current partial schedule is *bound* since it will not lead to a feasible complete schedule.)

However, it is possible to backtrack to continue the search even after a non-strongly-feasible schedule is found. Backtracking is done by discarding the current partial schedule, returning to the previous partial schedule, and extending it using a different transaction. The transaction chosen is the one with the *second* smallest H value. Even though we allow backtracking, the overhead of backtracking can be limited either by restricting the maximum number of possible backtracks or by restricting the total number of evaluations of the H function.

The algorithm works as follows: It starts with an empty partial schedule. Each step of the algorithm involves (1) determining that the current partial schedule is strongly-feasible, and if so (2) extending the current partial schedule by one transaction. In addition to the data structure maintaining the partial schedule, transactions in the transaction set S are maintained in the order of increasing deadlines. This is realized in the following way: When a transaction arrives at a node, it is *inserted*, according to its deadline, into a (sorted) list of transactions that remain to be executed. This insertion takes at most $O(n)$ time where n is the transaction set size. Let $n_k = \min(k$, number of transactions yet to be scheduled). Then when attempting to extend the schedule by one transaction, three steps must be taken: (1) strong-feasibility is determined with respect to the first (still remaining to be scheduled) n_k transactions in the transaction set, (2) if the partial schedule is found to be strongly-feasible, then the H function is applied to the first n_k transactions in the transaction set (i.e., the n_k remaining transactions with the earliest deadlines), and (3) that transaction which has the smallest H value is chosen to extend the current schedule. Given that only n_k transactions are considered at each step, the complexity incurred is $O(n \times k)$ since only the first n_k transactions are considered each time. If the value of k is constant (and in practice, k will be small when compared to the transaction set size n), the complexity is linearly proportional to n, the size of the transaction set [19]. While the complexity is proportional to n, the algorithm is programmed so that it incurs a fixed worst case cost by limiting the number of H function evaluations permitted in any one

invocation of the algorithm. See [19] for a discussion on how to choose k.

The heuristic function H can be constructed by simple or integrated heuristics. In the context of real-time tasks, extensive simulation studies of the algorithm [19] show that an H function based on T_d, a task's deadline and T_{est}, its earliest start time, has superior performance. Given a partial schedule, T_{est} can be computed from the earliest times resources will be available – derivable from the t_e's associated with tasks in the partial schedule, the resource needs of tasks yet to be scheduled, as well as t_b's of these tasks. We believe that this heuristic will have good performance even in the database context.

Finally, recall that a guarantee is done with respect to a transaction's worst case computational needs. *Resource reclaiming* refers to the problem of *correctly* and *effectively* utilizing the resources unused by a transaction when a transaction executes less than its worst case computation time. [22] discusses two resource reclaiming algorithms that effectively reclaim the unused times. One of the positive fallouts of reclaiming is that now we can afford to be pessimistic about the computation times of transactions. This is because even if the dynamic guarantees are done with respect to transactions' worst case computation times, since any unused time is reclaimed, the negative effects of pessimism are considerably eliminated.

4.2 Concurrent Processing of Prefetch and Execution Phases of Transactions

An important design question concerns the relative amount of CPU resources that should be allocated to perform the prefetch and execution phases of the incoming stream of transactions. Note that when prefetch phases are executed, they produce later work for the system in performing the execution phases. A naive approach to scheduling these different types of work can lead to an unstable behavior and harm the capability of the system to meet transaction deadlines. In workload environments where too many transactions are submitted for the system to handle, the question of how we are to throttle execution to a rate that the system can handle is not a simple one.

A naive approach is not to earmark any time to execute the prefetch phases of transactions but to execute them only when the processors are free, i.e., when no execution phases are scheduled to execute. This approach will lead to alternating bursts of prefetch phases followed by bursts of execution phases. During the periods when we are performing prefetch phases, it is possible that the CPU will be badly under-utilized, since there is little complex calculation to perform. Even more significant, a transaction arriving just as the system goes into its execution phase will have to wait through a cycle of execution and prefetch before its execution phase can be planned. This means that the transaction deadline is less likely to be met in such an unstable system.

Suppose we ensure that prefetch phases get a chance to execute at regular intervals. A straightforward way is to guarantee a periodic activity which is guaranteed along with the execution phases of transactions. The period and length of this activity can be adjusted to transaction characteristics and arrival rates. This technique is applicable to both uniprocessor as well as multiprocessor systems. Of course, prefetch phases can also execute when a CPU is idle.

Another approach, applicable in the context of a shared memory multiprocessor system, is one where a subset of the processors is allocated for executing the prefetch phases while the rest are used for the execution phases. The allocation of processors to each type of processing will depend on the computational needs of the prefetch phases vs. the execution phases, the arrival rate of transactions, elapsed time for I/O, etc. Since these parameters are likely to vary with dynamic arrivals, it should be possible to switch a processor from being used for one type of processing to another.

If the first approach can be thought of as a temporal partitioning of CPU time, the second one can be regarded as a spatial partitioning approach. Note that it is important that I/O interrupts that occur during the prefetch phase be serviced as they occur, even while a transaction is running in its execution phase. In uniprocessor environments, a constant background cost for servicing a certain maximum number of interrupts is easy enough to factor in to the execution scheduling, and simply means that the scheduler needs to take such costs into account. Note that it is acceptable to occasionally disable

I/O interrupts in finishing an execution with a tight deadline, but there is a hardware limit to the queue of unserviced interrupts. One advantage of the spatial partitioning approach is that the execution phases are shielded from the interruptions needed to handle new transaction arrivals.

These approaches might fail if the relative ratios of computational resources allocated for the two phases are imbalanced for the given workload. If the CPU resource allocated to execution is too high, we have the same kind of instability that we saw with the naive approach: bursts alternating between prefetch and execution. If the CPU resource allocated to execution is too low, we will prefetch for a large number of transactions we do not later execute, and waste CPU time we could be using for execution.

Ideally, we want to take a homeostatic approach to splitting up the work between prefetch and execution. This homeostasis should probably be based on the length of the queue of transactions waiting to execute. When the queue is long by historical measures, we can cut down on the prefetch activity and increase the resources going to execution. When the queue is short, but before it runs out entirely, we can increase the resources devoted to prefetch.

Another question of resource use is under what circumstances the system should drop a transaction from consideration in an overloaded schedule without performing the prefetch phase. That is, it will be beneficial to employ a filter at the beginning of a transaction's prefetch phase so that it enters the system only if its chances of being completed by its deadline are high [12]. We expect to put further thought into these questions in future work.

5 Extensions to the Basic Approach

In this section, we examine two extensions to the approach outlined earlier. One is applicable in situations where an optimistic concurrency control provides improved performance and another is applicable when a transaction's data requirements are known a priori.

In cases where there is a great deal of overlap (for some reason) between CPU

computations in the prefetch phase and CPU computations in the execution phase, we may want to perform all the CPU work the first time through, and then validate. If validation fails, we can retry and schedule the transaction for memory-resident execution. This is close to the approach taken in [8]. If we commonly have deadlines that will not be met because of I/O as discussed above, and we have a very large *CPU-comp* component, then we will be wasting a lot of CPU time as well before we find out that the deadline can't be met. An approach such as ours where *CPU-comp* components are executed only after confirming that the deadline can be met is much preferable.

Suppose it is possible to predeclare data needs of (some) transactions. We might conceivably want to guarantee such transactions then while I/O is in progress to bring the data into memory. We could pre-analyze by lookaside exactly what page I/Os have to be performed. Since this means that we can overlap CPU execution with necessary I/O and finish earlier, we will win in this case if deadlines are tight and *CPU-comp* is relatively large. Of course, we will have to make worst-case assumptions about completing the I/O in time for the transaction to use the data that is read into memory. We would lose, for example, if we used a relatively naive approach and started a long I/O transaction that conflicted with some memory-resident transaction almost ready to execute its execution phase so that we run out of buffer space. The latter case can be avoided if we start the transaction only when enough buffer space is available for this transaction as well as all ongoing transactions. These can be handled as part of a transaction's physical resource requirements and taken into account by the scheduling algorithm.

If we do not get to overlap *CPU-comp* and I/O, this could make a difference in cases with large *CPU-comp* and very tight deadlines. It would only make a difference however if some other method would work better, and the optimistic method discussed earlier might answer this problem. Note though that the large *CPU-comp* case is also the case where we can waste a lot of CPU if validation (with the optimistic case) fails frequently.

6 Conclusions and Future Work

The access invariance based approach discussed in this paper is motivated by a need to provide for predictable transaction executions. It involves a two-phase solution. In the first phase, refered to as the prefetch phase, a skeleton version of the transaction is executed, without data locking, to evaluate what data items and disk pages the transaction requires; these disk pages are brought in from disk and made resident in memory. The skeleton version can be considered to be equivalent to the actual version with respect to disk accesses but may not be equivalent to the CPU-intensive activities of the transaction. After all pages are resident, we know everything about the data read/write set for the transaction. Hence in the second phase, known as the execution phase, the transactions are executed so as to avoid conflicts and to complete before the respective deadlines. A number of variations on this approach (such as performing optimistic concurrency control in the prefetch phase) are also investigated.

We review some of the benefits of this approach for transactions that maintain access invariance. Firstly, transactions become more predictable because we don't have to make pessimistic I/O estimates before scheduling. Secondly, an important advantage over 2PL is that whereas in 2PL future I/O requests will be delayed by delays in obtaining locks, since the prefetch phase does I/O without locking, we can continue to make progress with I/O for all transactions in their prefetch phase. Thirdly, transactions that possess access invariance become more predictable, since they do not abort. In cases where predeclared read/write sets are impossible, the I/O work done to achieve this predictability is the minimum possible. It is especially important to transactions that interact with the real-world since a transaction will not be started unless it will complete all its interactions. Finally, we maintain that the approach is quite generally applicable, since many transactions can be designed to possess the access invariance property. Occasional exceptions can be detected and handled (although this means that some portions of the prefetch phase must be re-executed before the execution phase can begin). In any case, this approach provides a way by which, once guaranteed, a transaction will complete by its deadline and time-consuming recovery actions are not necessary if a transaction is unable to execute.

The price paid in the latter situation is the overhead of the prefetch phase.

Other researchers have used a prefetch based approach implicitly or explicitly. In [8], prefetch was used to improve *throughput* in high contention transactional environments. In [17], investigating time-critical stock trading situations, a type of prefetch was used to reduce contention and thus improve throughput. However, we believe that the current paper is the first one to explicitly use dynamic prefetching and conflict avoidance scheduling to execute real-time transactions predictably.

In terms of work that remains to be done, we should note that we are beginning to undertake an implementation and evaluation of this approach. We also plan to test our assumptions about access invariance in the context of different database applications. In [8], the authors have shown via simulation studies that access variance can be exploited in traditional databases to improve performance, that is, to reduce response times. We hence believe that use of this property in real-time databases will lead not only to predictable transaction executions but also increase the number of transactions that meet deadlines. It is well known that relatively large slack times (a factor of 10 or more compared to the execution time) allow very good scheduling in the memory resident case. Simulations typically restrict themselves to smaller average factors in order to consider a region of interest, where different algorithms have different effect. We hope to show, as alluded to in Section 3, that when we deal with a disk resident database, given the high variance of the I/O, even with low slack when I/O is included, almost all transactions will have extremely large slack by the time they enter their execution phase. Thus, prefetch in realistic regions of time-criticality for disk-resident databases will lead to a relatively trivial CPU scheduling problem after I/O is complete (in the prefetch phase), leading to good real-time performance.

References

[1] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions: A Performance Evaluation," *Proceedings of the 14th VLDB Conference*, Aug. 1988.

[2] R. Abbott and H. Garcia-Molina, "Scheduling Real-Time Transactions with Disk Resident Data," *Proceedings of the 15th VLDB Conference*, 1989.

[3] P.A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems.* Addison-Wesley, Reading, MA, 1987.

[4] A.P. Buchmann, D.R. McCarthy, M. Chu, and U. Dayal, "Time-Critical Database Scheduling: A Framework for Integrating Real-Time Scheduling and Concurrency Control", *Proceedings of the Conference on Data Engineering*, 1989.

[5] R.G.G. Cattell, *Object Data Management.* Addison-Wesley, Reading, MA, 1987.

[6] S. Chen, J. Stankovic, J. Kurose, and D. Towsley, "Performance Evaluation of Two New Disk Scheduling Algorithms for Real-Time Systems", *Real-Time Systems*, Sept. 1991.

[7] K. Elhardt and R. Bayer, " A Database Cache for High Performance and Fast Restart in Database Systems", *ACM TODS*, Vol 9, No. 4, pp. 503-525.

[8] Peter A. Franaszek, John T. Robinson, and Alexander Thomasian, "Access Invariance and its Use in High Contention Environments", *Proceedings of the Sixth International Conference on Database Engineering*, 1990, pp 47-55.

[9] Jim Gray and Franco Putzolu, "The Five Minute Rule for Trading Memory for Disk Accesses and the 10 Byte Rule for Trading Memory for CPU Time", Proceedings of the 1987 ACM SIGMOD Conference, pp 395-398.

[10] J.R. Haritsa, M.J. Carey and M. Livny, "On Being Optimistic about Real-Time Constraints," *Proceedings of ACM PODS*, 1990.

[11] J.R. Haritsa, M.J. Carey and M. Livny, "Dynamic Real-Time Optimistic Concurrency Control," *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.

[12] J.R. Haritsa, M.J. Carey and M. Livny, "Earliest Deadline Scheduling for Real-Time Database Systems," *Proceedings of the Real-Time Systems Symposium*, Dec. 1991.

[13] J. Huang, J.A. Stankovic, D. Towsley and K. Ramamritham, "Experimental Evaluation of Real-Time Transaction Processing," *Proceedings of the Real-Time Systems Symposium*, Dec. 1989

[14] J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, "Experimental Evaluation of Real-Time Optimistic Concurrency Control Schemes", *Proceedings of the Conference on Very Large Data Bases*, Sep 1991.

[15] J. Huang, J.A. Stankovic, K. Ramamritham and D. Towsley, "On Using Priority Inheritance in Real-Time Databases," *Proceedings of the Real-Time Systems Symposium* December 1991.

[16] Y. Lin and S.H. Son, "Concurrency Control in Real-Time Databases by Dynamic Adjustment of Serialization Order," *Proceedings of the Real-Time Systems Symposium*, Dec. 1990.

[17] Peter Peinl, Andreas Reuter, and Harald Sammer "High Contention in a Stock Trading Database, a Case Study", Proceedings of the 1988 SIGMOD Conference, pp 260-268.

[18] C. Pu and A. Leff. Replica Control in Distributed Systems: An Asynchronous Approach. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 377–386, May 1991.

[19] K. Ramamritham, J. Stankovic, and P. Shiah, "Efficient Scheduling Algorithms for Real–Time Multiprocessor Systems," *IEEE Transactions on Parallel and Distributed Systems*, Vol. 1, No. 2, April 1990, pp. 184-194.

[20] K. Ramamritham, "Real-Time Databases" *International Journal of Distributed and Parallel Databases*, to appear.

[21] L. Sha, R. Rajkumar and J.P. Lehoczky, "Concurrency Control for Distributed Real-Time Databases," *ACM SIGMOD Record*, March 1988.

[22] C. Shen, K. Ramamritham, and J. Stankovic, Resource Reclaiming in Real-Time, *Proc Real-Time System Symposium*, December 1990.

[23] John R. Wilke, "Parallel Computing Finds Mainstream Use", Wall Street Journal Technology Column, January 6, 1992, pg B1.