Pergamon

Information Systems Vol. 23, No. 2, pp. 109–125, 1998

Printed in Great Britain
0306-4379/98 $17.00 + 0.00

A REAL-TIME CONCURRENCY CONTROL PROTOCOL FOR MAIN-MEMORY DATABASE SYSTEMS†

Özgür Ulusoy[1] and Alejandro Buchmann[2]

[1]Department of Computer Engineering and Information Science, Bilkent University, Ankara, Turkey
[2]Department of Computer Science, Technical University Darmstadt, Darmstadt, Germany

(Received 1 April 1997; in final revised form 3 February 1998)

Abstract — Protocols developed so far for concurrency control in real-time database systems have assumed a dynamic acquisition of data resources as it is impossible to predict which instances of relations will actually be accessed by each transaction. In this paper‡, we present a new concurrency control protocol for main-memory real-time database systems, which is based on predeclaration of data requirements at a relation granularity. When a transaction is submitted, it is possible to detect with simple syntactical means which relations will be accessed by the transaction. Preacquisition of data resources enables the system to execute transactions in a conflict-free manner. The protocol also offers the possibility of determining execution times without the effects of blocking and I/O, thereby allowing us to give guarantees for the execution of high-priority transactions. Through a series of simulation experiments, the protocol is compared against some typical concurrency control protocols proposed recently for real-time database systems. Copyright ©1998 Elsevier Science Ltd

Key words: Real-Time Database Systems, Main-Memory Database Systems, Concurrency Control.

1. INTRODUCTION

Predictability of transaction execution is a basic issue in Real-Time Database Systems (RTDBSs). Without predictability the only performance metric that can be defined is based on throughput but little can be said about individual transactions meeting their deadlines. The main reasons for the lack of predictability are:

- transaction execution depends on the state of the database,
- transactions acquire resources dynamically which may lead to one transaction blocking another or even deadlocking, with the corresponding rollbacks and restarts, and
- page faults are data- and load-dependent and unpredictable.

A solution to the predictability problem must therefore address these issues simultaneously.

There has been a considerable amount of work in the area of RTDBSs. An extensive exploration of the issues is presented in [11, 19, 21]. Most of the work in the RTDBS area has focused on the development and evaluation of priority-cognizant concurrency control protocols (e.g., [1, 2, 4, 6, 8, 10, 12, 17, 18, 20]). Many proposed concurrency protocols have transferred assumptions and results from non-RTDBSs in an unchallenged form to the realm of RTDBSs.

The chain of thought underlying conventional concurrency control is the following: I/O is 3-4 orders of magnitude slower than a main memory access. Therefore, throughput is improved if the system continues executing another transaction while the one that was previously executing performs an I/O operation. If you have many page faults and a transaction only executes briefly before requiring another I/O you improve the throughput by having many transactions ready to run. This will be easier to achieve if a transaction locks only the minimal amount of data needed to execute. Therefore, small-granule locks increase concurrency and performance in disk-resident database systems. However, small-granule locks are more effective if a transaction locks only the

†Recommended by Patrick O'Neil

‡A summary of the ideas presented in this paper has appeared in *SIGMOD Record*, Special Issue on Real-Time Databases, vol.25, no.1, March 1996.

minimal number of granules. Since a transaction's execution path depends on the state of the database at the moment of execution, it is difficult to determine at compile time what instances will be touched by a transaction. Therefore, to exploit small granularity and increase concurrency a transaction should acquire dynamically the minimal number of tuples.

It becomes clear that having slow I/Os as part of a transaction is an important reason for having small grained locks and indirectly for acquiring data resources dynamically.

On the other hand, eliminating I/O operations from the execution of a transaction reduces drastically the significance of concurrency because a transaction will never have to wait for 20 mseconds until an I/O is executed. If concurrency is less important, so are the tuple-level locks. In fact, it has been demonstrated that very large lock granularities (e.g., relations) are most appropriate in main memory databases [5, 13]. Finally, if small locks are not as important, the main reason for acquiring data resources dynamically becomes questionable.

We think that eliminating I/O operations through the use of main-memory databases and the impact this has on the preferred lock sizes requires that we revisit and analyze concurrency control and scheduling with predeclaration of resources. It is simple to determine by purely syntactic means during transaction compilation the read and write sets of the transaction at the relation level. If the lock granularity is the relation, it becomes feasible to use simple but efficient concurrency control and scheduling mechanisms based on resource predeclaration.

In this paper, we propose a simple, predeclaration-based concurrency control protocol for main-memory RTDBSs. Predeclaration protocols require the knowledge of what resources will be used ahead of time and the granularity at which resources are locked. A transaction, when submitted, is always parsed and compiled into an internal, optimizable form. During parsing it is possible to detect in the same parsing step whether the proposed access is for reading or writing purposes. Under these conditions the read and write sets for a transaction can be established a priori and off-line, and the transaction can be scheduled in a conflict-avoiding manner by preacquiring the necessary resources. By doing this, a transaction will execute without blocking and will minimize its time in the system. Data resources, although in bigger granules, will be locked only the time required for the actual computation of a transaction. The CPU as the limiting resource will be fully used and less time will be wasted on lock management. We claim that our protocol is much more efficient than previously proposed protocols with dynamic resource acquisition. It also offers the possibility of determining execution times without the effects of blocking and I/O thereby allowing us to give end-to-end guarantees for the execution of high-priority transactions.

The approach proposed by O'Neil, Ramamritham and Pu [15] also recognizes the fact that I/O and dynamic acquisition of data resources leads to unpredictability. In that approach the I/O phase and computations necessary for determining the needed data resources are executed in a first prefetch phase. Under the assumption of access invariance the actual transaction is executed on the prefetched data in a second phase, the execution phase. In this approach the duration of the execution phase can be guaranteed, but not the duration of the prefetch phase. Therefore, while the approach is somewhat more general than ours and requires smaller locks, it can only give guarantees for the execution phase but not end-to-end guarantees.

Being able to give guarantees for individual transactions rather than measuring throughput and determining a posteriori the number of transactions that did miss their deadline is important in applications such as simulations with hardware in the loop. Combining simulated components with actual, physical components imposes some hard timing constraints for individual transactions. Furthermore, many applications requiring timing guarantees are event driven. This means that transaction arrival is dynamic but the transactions that are executed typically belong to a finite set of predefined and precompiled transactions. The elimination of blocking and I/O uncertainties makes it possible to predict worst case execution times for transactions. This is a necessary condition for implementing admission control.

In this paper, we also describe a performance model designed for studying various issues in main-memory RTDBSs. A series of experiments was carried out using this model to compare our protocol against some other concurrency control protocols chosen from the literature. Relative performance of the protocols was evaluated under a range of workloads and system configurations. The results obtained indicate that, as expected, the proposed protocol can provide significant performance gains over the other protocols.

In the next section, the new concurrency control protocol that we call "Predeclaration Protocol" is described in detail. Section 3 provides a description of the simulation model of a RTDBS which has been used to obtain the performance results presented in this paper. The performance experiments and results are presented in Section 4. The last section summarizes the conclusions of our work.

2. THE PREDECLARATION PROTOCOL

The data structures associated with the protocol are:

- *read_set[T]*: The set of relations to be read by transaction T;
- *write_set[T]*: The set of relations to be updated by transaction T;
- *conflicting_transactions[T]*: The list of transactions that are conflicting with T;
- *ready-queue*: The list of transactions that are ready to execute;
- *wait-queue*: The list of transactions that have some access requests conflicting with the scheduled transactions.

An 'already scheduled transaction' in the following description corresponds to a transaction which is either executing, or in the *ready-queue* or *wait-queue*.

For each transaction submitted to the system, the set of relations to be accessed by the transaction, and the mode of each access (i.e., either read or write) are determined. Then, a conflict check is performed between the relations to be accessed by the new transaction and the relations in the access list of already scheduled transactions. If no conflict is detected, the transaction is inserted into the *ready-queue*. Otherwise, the transaction is inserted into the *wait-queue*, and the *conflicting_transactions* list is established for the transaction. The *ready-queue* is organized on the basis of transaction priorities. A variety of criteria can be used to determine a transaction's priority.

When a transaction is committed, its read and write locks are released. The id of the committed transaction is removed from the *conflicting_transactions* list of each transaction in the *wait-queue*. If, for any transaction in the *wait-queue*, the list of *conflicting_transactions* becomes empty, that transaction is transferred to the *ready-queue*. Following each commitment, the first transaction in the *ready-queue* is started.

Following is a formal description of the Predeclaration Protocol.

For each transaction T submitted to the system,
  OFF-LINE:
    Parse transaction T, identify the relations to be accessed, and construct *read_set[T]* and *write_set[T]*.
  ON-LINE:
    Set *conflicting_transactions[T]* to empty.
    For each transaction T' which has already been scheduled,
      If $(read\_set[T] \cap write\_set[T']) \neq \emptyset$
      Or $(write\_set[T] \cap (read\_set[T'] \cup write\_set[T'])) \neq \emptyset$
        insert T' into *conflicting_transactions[T]*
    EndFor.
    If *conflicting_transactions[T]* is empty
      insert T into *ready-queue*
    else,
      insert T into *wait-queue*
    EndIf.

When a transaction T commits,
  The locks on the relations in *read_set[T]* and *write_set[T]* are released.
  For each transaction T' in the *wait-queue*,
    conflicting_transactions[T'] = *conflicting_transactions[T']* $- T$
    If *conflicting_transactions[T']* becomes empty
      T' is transferred to the *ready-queue*
    EndIf.
  EndFor.
  The first transaction in the *ready-queue* is started to execute after being granted its locks.

In the algorithm described above, conflicts among transactions are always detected between the newly arriving transaction and the older transactions already in the system. Since only the conflicting transactions set of the newly arriving transaction is updated, it inherently favors the older transactions. A variant of this protocol, that executes a transaction always before a lower priority transaction requiring the same data resources, is discussed in Section 4.5. In that variant, the resources already allocated to a ready transaction are taken back if those resources are requested by a higher-priority transaction.

3. A REAL-TIME DATABASE SYSTEM MODEL

In this section, we briefly describe the RTDBS simulation model that we used to evaluate the performance of the proposed concurrency control protocol. The model is based on an open queuing model of a multiprocessor, memory-resident database system.

The entire database is kept in main memory, while a stable copy, possibly out of date, is kept on disk. For each transaction the disk is only accessed once to write the log record onto disk and guarantee write-ahead logging. The disk-resident copy of the database can be updated asynchronously by applying the log (possibly on a separate processor). In the case of system failure, the database can be recovered from the stable copy and the log. Since application of the log records to the disk-resident version of the database can be done off-line, it is safe to assume that this process does not interfere with regular transaction processing. Therefore, the only I/O cost paid by a transaction is the writing of the log. A feasible alternative for writing to the disk is broadcasting the log to other machines, thus reducing the delay by almost an order of magnitude.

Transaction arrivals are assumed to be Poisson. Each transaction is associated with a real-time constraint in the form of a deadline. The transactions are prioritized based on the *Earliest Deadline First* (EDF) policy; i.e., a transaction with an earlier deadline has higher priority than a transaction with a later deadline. To ensure uniqueness of priorities, if any two of the transactions have the same deadline, the one that has arrived at the system earlier is assigned a higher priority. The transaction deadlines are *soft*; i.e., each transaction is executed to completion even if it misses its deadline.

The set of parameters described in Table 1 is used in specifying the configuration and workload of the RTDBS. The parameter *database_size* determines the number of relations stored in the database. The relation size is uniformly distributed within the range $0.5 *$ *relation_size* through $1.5 *$ *relation_size*. Number of relations accessed by a transaction is chosen from an exponential distribution and the actual relations are chosen uniformly from the database. For each relation accessed, the number of pages processed by the transaction is again chosen from an exponential distribution.

As described in the preceding section, one transaction in the *ready-queue* is started to execute each time a transaction commits. This means that, the total number of transactions executing in the system at any time is limited by the total number of processors the system has (i.e., *num_CPU*). If there is no ready transaction in the system when a transaction commits, the processor that has been running the committed transaction becomes idle. When a new transaction arrives at the system, if at least one processor is available (i.e., is not running a transaction), the arriving transaction is assigned to any of the available processors if it does not conflict with the transactions that have already been scheduled.

Configuration Parameters	
num_CPU	Number of CPUs
CPU_rate	Instruction rate of CPU (MIPS)
database_size	Number of relations in the database
relation_size	Average number of pages in each relation
instr_lock	Number of instructions for a lock/unlock operation
instr_data_read	Number of instructions per accessed page
instr_data_write	Number of instructions to modify a page
instr_context_switch	Number of instructions for switching between transactions and the scheduler
Transaction Parameters	
iat	Mean interarrival time of transactions
relation_access	Average number of relations accessed by each transaction
page_access_per_relation	Average number of pages accessed per relation by each transaction
update_prob	Probability of updating the accessed page
instr_xact_start	Number of instructions to initiate a transaction
instr_xact_terminate	Number of instructions to terminate a transaction
slack_rate	Average rate of slack time of a transaction to its processing time

Table 1: RTDBS Model Parameters

Concurrency control is implemented at a relation granularity.

The slack time of a transaction is chosen randomly from an exponential distribution with a mean of *slack_rate* times the estimated processing time of the transaction. The deadline of a transaction is determined by the following formula:

$$deadline = start\ time + processing\ time\ estimate + slack\ time$$

where

$$slack\ time = expon(slack\_rate * processing\ time\ estimate)$$

Let *#relations* denote the actual number of relations accessed by the transaction.

$$processing\ time\ estimate = \frac{1}{CPU\_rate} * (instr\_xact\_start + \#relations * (2 * instr\_lock$$
$$+page\_access\_per\_relation * (instr\_data\_read + update\_prob * instr\_data\_write))$$
$$+instr\_xact\_terminate)$$

4. PERFORMANCE EVALUATION

The details of the main-memory RTDBS model described in the previous section were captured in a simulation program. The default values of configuration and workload parameters used in the simulation experiments are presented in Table 2. The parameters were chosen to yield a transaction load and data contention high enough to observe the differences between performances of the evaluated protocols.

The primary performance metric used in the experiments is *success_ratio*; i.e., the fraction of transactions that satisfy their deadlines. The other metric that helped us analyze the results is *useful_CPU*, which is defined as follows:

$$useful\_CPU = \frac{CPU\ time\ spent\ for\ processing\ the\ operations\ of\ committed\ transactions}{Total\ CPU\ time\ used}$$

In determining *useful_CPU*, only reading and updating data pages are considered to be useful operations while the implementation overheads of the protocol, such as locking and context switching are considered not useful.

The simulation program was written in CSIM [16], which is a process-oriented simulation language based on the C programming language. For each configuration of each experiment, the final results were evaluated as averages over 20 independent runs. Each run was continued until 1000 transactions were executed. 95% confidence intervals were obtained for the performance results. The width of the confidence interval of each data point is within 4% of the point estimate. In displayed graphs, only the mean values of the performance results are plotted.

num_CPU	3
CPU_rate	100 MIPS
database_size	50 relations
relation_size	1000 pages
instr_lock	300 instructions
instr_data_read	30000 instructions
instr_data_write	20000 instructions
instr_context_switch	5000 instructions
iat	2 - 12 mseconds
relation_access	3
page_access_per_relation	5
update_prob	.5
instr_xact_start	30000 instructions
instr_xact_terminate	40000 instructions
slack_rate	5

Table 2: Performance Model Parameter Values

instr_conflict_check	300 instructions
instr_deadlock_check	1000 instructions
instr_xact_start	10000 instructions
instr_xact_valid	20000 instructions

Table 3: Values of Parameters for Protocols PA, PI, and OPT

4.1. Impact of Transaction Load

In this section, we provide the performance results of the Predeclaration Protocol (PRED) under varying transaction loads in the system. Mean time between successive transaction arrivals (i.e., *iat*) was varied from 2 mseconds to 12 mseconds in steps of 1. We present our findings together with a comparison to the performance of some other protocols which were chosen as representatives of different types of concurrency control protocols proposed for RTDBSs. In the following, we first provide a brief description of three protocols that were selected for comparison, and then discuss the performance results obtained using our RTDBS model. In all three protocols described below, the transactions acquire their locks dynamically on individual relation pages; in other words, the protocols are implemented at a page level granularity. However, the results obtained by executing the protocols at a relation granularity are also provided at the end of this section.

Priority Abort (PA) Protocol: This protocol resolves data conflicts always in favor of high-priority transactions [1]. At the time of a data lock conflict, if the lock-holding transaction has higher priority than the priority of the transaction that is requesting the lock, the latter transaction is blocked. Otherwise, the lock-holding transaction is aborted and the lock is granted to the high priority lock-requesting transaction. Assuming that no two transactions have the same priority, this protocol is deadlock-free since a high priority transaction is never blocked by a lower priority transaction.

Priority Inheritance (PI) Protocol: The priority inheritance method, proposed in [17], ensures that when a transaction blocks higher priority transactions, it is executed at the highest priority of the blocked transactions; in other words, it inherits the highest priority. The aim is to reduce the blocking times of high priority transactions.

Optimistic Wait-50 (OPT) Protocol: OPT is an optimistic concurrency control protocol incorporating real-time priorities of transactions [6]. The validation check for a committing transaction is performed against the executing transactions and if the write-set of the validating transaction intersects with the read-set of one of the executing transactions, these two transactions are said to be in conflict. The proposed protocol uses a *50 percent* rule as follows: If half or more of the transactions conflicting with a committing transaction are of higher priority, the transaction is made to wait for the high priority transactions to complete; otherwise, it is allowed to commit while the conflicting transactions are aborted. While the transaction is waiting, it is possible that it will be restarted due to the commit of one of the conflicting transactions with higher priority.

For protocols PA and PI, we simulate the cost of data conflict check at each data access request, by using a parameter *instr_conflict_check* and setting its value to 300 instructions. On the other hand, since no conflict check is performed during the initiation of a transaction, the value of *instr_xact_start* is set to 10000 instructions, rather than 30000 instructions, in executing these two protocols.

Protocol PI is prone to deadlocks. For this protocol, deadlock detection is performed through a wait-for graph each time a transaction is blocked. The processing cost of checking for a deadlock is simulated by using a parameter *instr_deadlock_check* with a value of 1000 instructions. A detected deadlock is recovered from by selecting the lowest priority transaction in the deadlock cycle as a victim to be aborted. An aborted transaction is restarted with its original priority, and it accesses the same data as before. The overhead of aborting a transaction is the time wasted by the aborted transaction.

For the optimistic protocol OPT, we again set the initiation cost of a transaction, *instr_xact_start*, to 10000 instructions. The validation cost of a transaction at the end of its execution is simulated by a parameter *instr_xact_valid* with a value of 20000 instructions. Values of parameters that are specific to protocols PA, PI, and OPT are listed in Table 3.

As detailed above, protocol PRED limits the total number of transactions executing at any time (i.e., the multiprogramming level) by the number of processors in the system. In order to have a fair comparison of the protocols, the same level of multiprogramming is used with protocols PA, PI, and OPT as well. When this level is reached, arriving transactions are temporarily blocked.

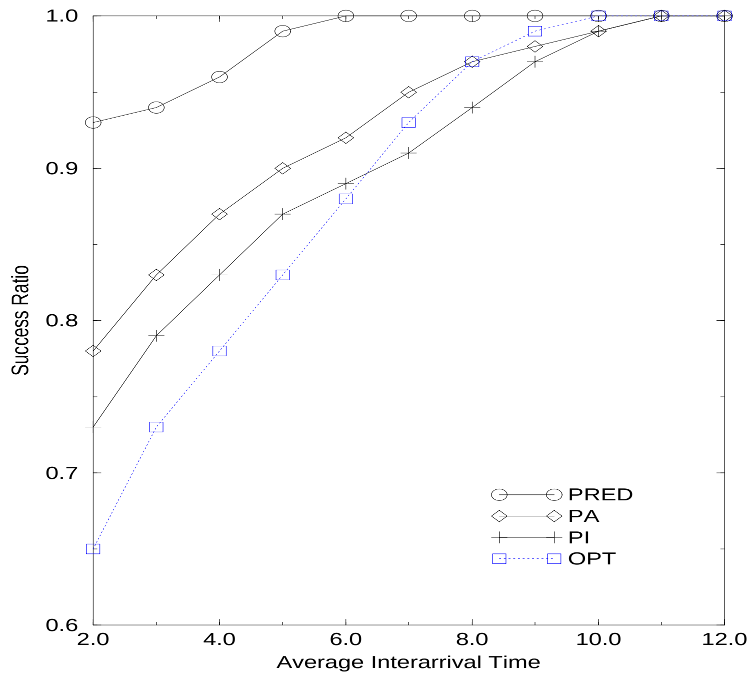

Fig. 1: Real-Time Performance of the Protocols as a Function of the Average Interarrival Time of Transactions

Figure 1 displays the *success_ratio* results for the concurrency control protocols as a function of the average interarrival time of transactions. Our simulation program captures the effects of both

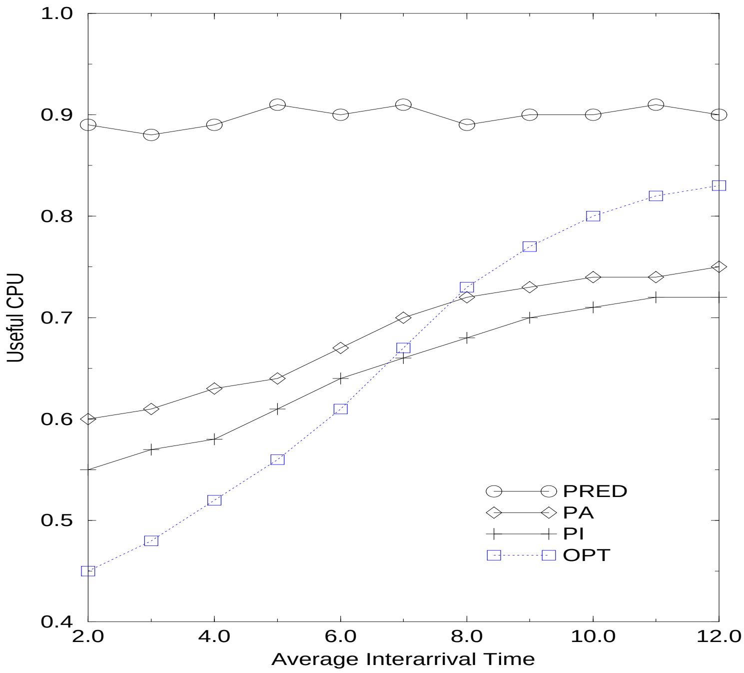

Fig. 2: Useful CPU Time Results

data contention and resource contention. Data contention exists due to conflicting data access requests of transactions. Resource contention is due to the limited number of CPUs in the system, which results in queuing delay at each of those CPUs. The transaction load has an impact on both data and resource contention in the system. The number of data access conflicts among concurrent transactions and the average length of CPU queues increases as more transactions are processed. Therefore, increasing the value of *iat* (i.e., decreasing the level of transaction load) leads to better performance for all concurrency control protocols tested in this experiment.

The comparative performance results of protocols PA, PI, and OPT are similar to what we obtained with a disk-resident RTDBS [20]: Protocol PA works in general better than protocol PI for a wide range of mean interarrival time. The optimistic protocol OPT performs well for large values of interarrival time; i.e., when the system is lightly loaded. Since the number of conflicts is small under low load levels, only a few transactions fail to be validated at commit time. However, under high levels of transaction load, the performance of protocol OPT is worse than that of other protocols. This result can be contributed to the increased number of restarts due to increased conflicts among transactions. The transactions that are in conflict with a committing transaction are aborted and restarted from the beginning. The wasted execution time due to the large number of restarts substantially increases the number of missed deadlines.

The performance of these three protocols, in general, is considerably worse than that of our new protocol PRED. With protocol PRED, except under very high transaction loads, almost all transaction deadlines are satisfied. This result is due to the fact that the CPU time is not wasted due to conflict checks at each data access, or on context switches between transactions and the scheduler. Some of data conflicts in protocols PA and OPT can result in transaction aborts, which is another source of the CPU waste. With protocol PI, on the other hand, there is the possibility and CPU cost of deadlocks. Protocol PRED enables the system to spend more useful CPU time on transaction processing, and as a result, even under very high loads only a few transactions miss their deadlines. Figure 2 presents the *useful_CPU* results for all three protocols. Remember that *useful_CPU* specifies the fraction of CPU time that is not wasted (i.e., used for processing the operations of committed transactions).

In order to see how the performance results of the locking protocols PA and PI are affected by a change in the lock granularity, we also implemented these protocols at a relation granularity. The results obtained are displayed in Figure 3. Each of these two protocols provides better performance

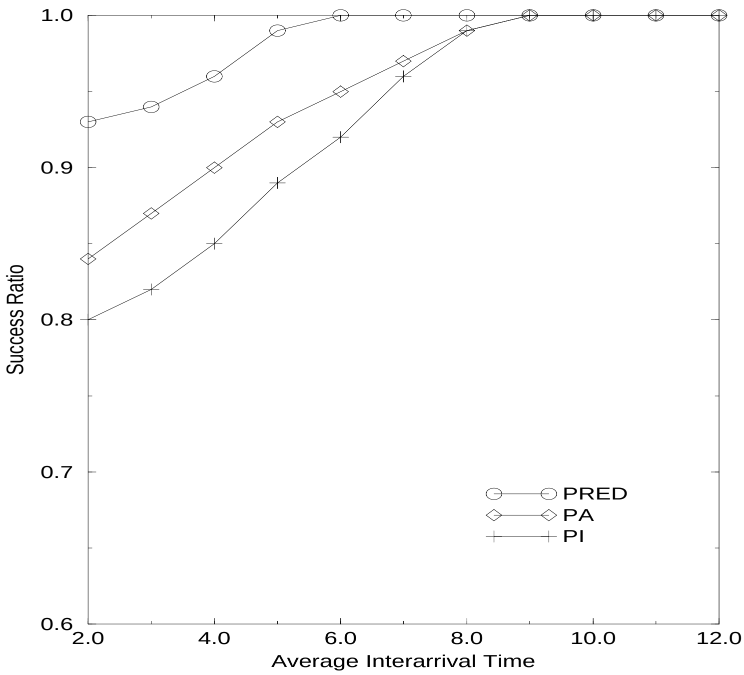

Fig. 3: Real-Time Performance of the Locking Protocols When They are All Implemented at a relation granularity

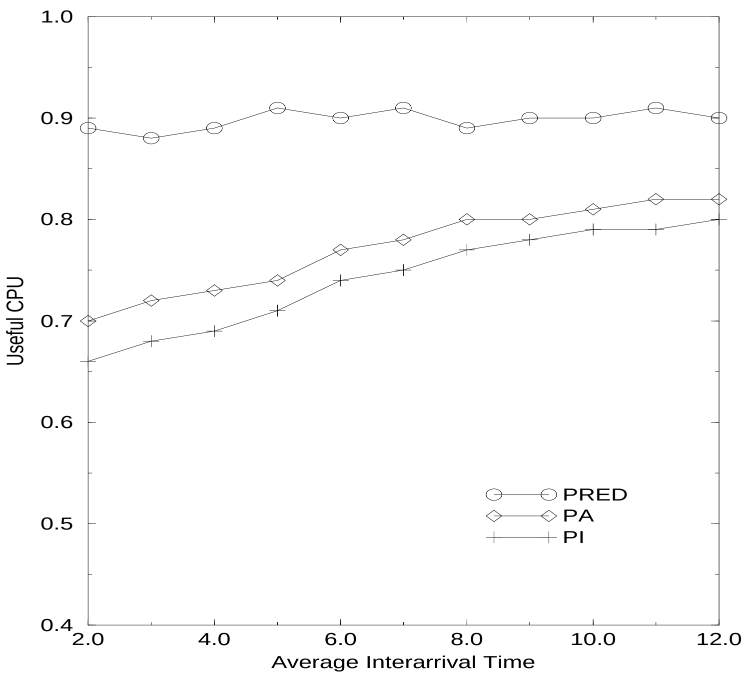

Fig. 4: Useful CPU Time Results When All the Protocols are Implemented at a Relation Granularity

than that obtained by executing it at a smaller (page level) granularity. This result confirms our intuition that using large lock granularity with any concurrency control protocol can lead to an improvement in the performance of main-memory RTDBSs. Comparing the performance results of these protocols against those of protocol PRED, it can be observed that protocols PA and PI still cannot reach the performance level achieved by PRED. This is because the CPU waste experienced with each of these two protocols is still higher than that of protocol PRED (see Figure 4).

4.2. Sensitivity of the Results to Some Other Parameters

The experiments we have discussed so far looks at the sensitivity of performance results to the mean interarrival time parameter. In the experiments of this section, we performed some more sensitivity analyses to see the effects of other parameter choices. One of the system parameters that determines the level of data contention in the system is *relation_access* (average number of relations accessed by a transaction). By varying the value of this parameter we were able to evaluate the comparative performance of concurrency control protocols under different levels of data contention. Average number of pages accessed per relation by each transaction was fixed at 5 throughout this experiment while the default value used for the mean interarrival time parameter was 5 mseconds.

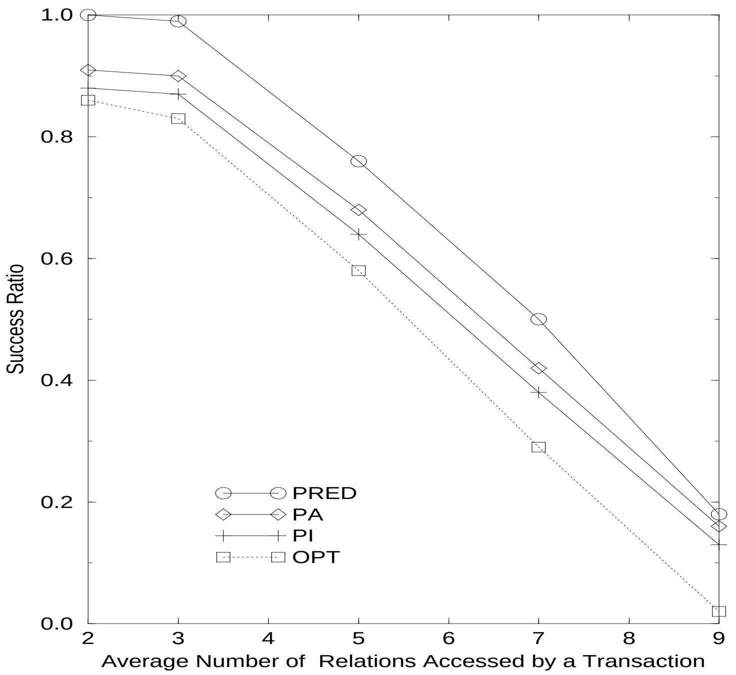

Fig. 5: Real-Time Performance Results under Varying Levels of Data Contention

It is indicated by the performance results displayed in Figure 5 that increasing the level of data contention by running longer transactions can significantly affect the system performance. For each concurrency control protocol, increasing the average length of transactions leads to a rapid degradation in the performance. Another observation in this experiment is that the relative performance of protocols PRED, PA and PI is not affected much by the change in the degree of data contention unless the system is characterized by an extremely high level of data contention in which case all the protocols perform almost equally bad (this was the case we observed for the *relation_access* value of 9). Under very high contention conditions, all the protocols suffer from the overhead of data conflicts (in terms of its resulting effects, like blockings and aborts) to a great extent. Although protocol PRED still provides a bit better performance, with any of the protocols the majority of transactions miss their deadlines when the level of data contention is very high. The effect of data contention on the performance of protocol OPT is worse than the others as this protocol is characterized by large number of restarts under high data conflict conditions as we explained before.

In the experiments we discussed so far, it has been assumed that the list of relations accessed by each transaction is chosen uniformly from the whole database. To evaluate the impact of data contention under a "hot spots" model, we partition the database into two regions: a "hot" region where most of the references are made, and a "cold" region that is accessed rarely. 20% of the relations stored in the database is designated as the hot region where 80% of the data accesses are directed. The rest of the database is specified as the cold region. Adapting this hot spots data access model to our system leads to a substantial increase in data contention among transactions, because most of data accesses are to a small portion of the database that corresponds to 10 relations in our system. Performance results of the protocols are displayed in Figure 6. Compared to the results presented in the preceding section, the performance of all the protocols obtained under this high contention environment is at a lower level, and also the performance results of PA and PI are closer to those of PRED. This result confirms our findings in the previous experiment of this section: in such a high contention environment, like any other protocol, protocol PRED experiences a large number of data conflicts and its resulting effects. The implementation overhead (i.e., CPU waste) of the protocols becomes less effective in determining the relative performance.

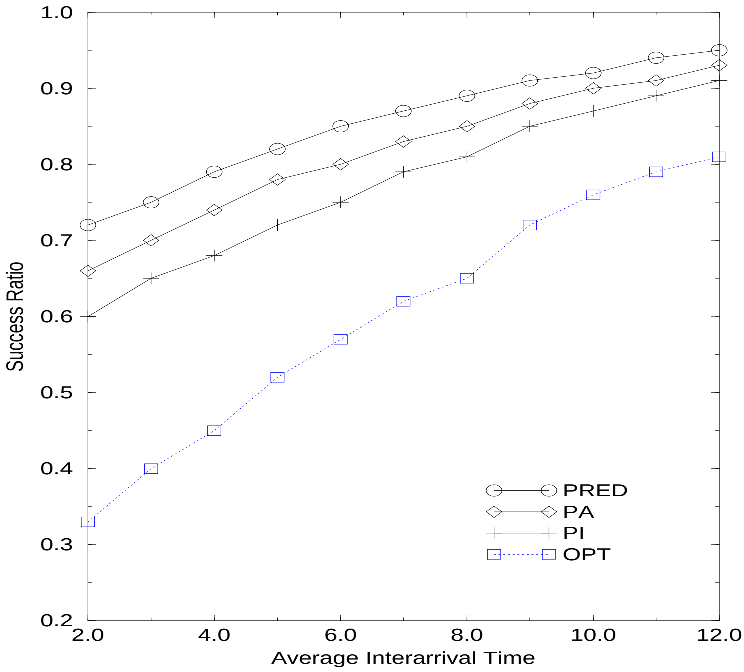

Fig. 6: Real-Time Performance Results under a Hot Spots Data Model

Another experiment that we conducted evaluates the impact of the CPU cost of processing data pages. Two related system parameters studied in that experiment are *instr_data_read* (the number of CPU instructions per accessed page) and *instr_data_write* (the number of CPU instructions to modify a page). Performance results obtained by varying the value of parameter *instr_data_read* are displayed in Figure 7. For low values of this parameter, all the protocols perform very well. The average lifetime of transactions is relatively short and therefore transactions do not experience many data conflicts. As the CPU cost of processing a data page is increased, data contention among transactions become high enough to bring out the differences between the performances of protocols.

When the processing cost becomes very high; i.e., for very large values of *instr_data_read*, all the protocols perform quite poor due to the large number of data conflicts experienced. Similar to the results we obtained under very high data contention environments (see the first experiment of this section), the performance results of the protocols (except OPT) are close to each other when the time spent for processing a page becomes extreme. This is not surprising since the cost of data conflicts dominates all the other factors that can affect the relative performance.

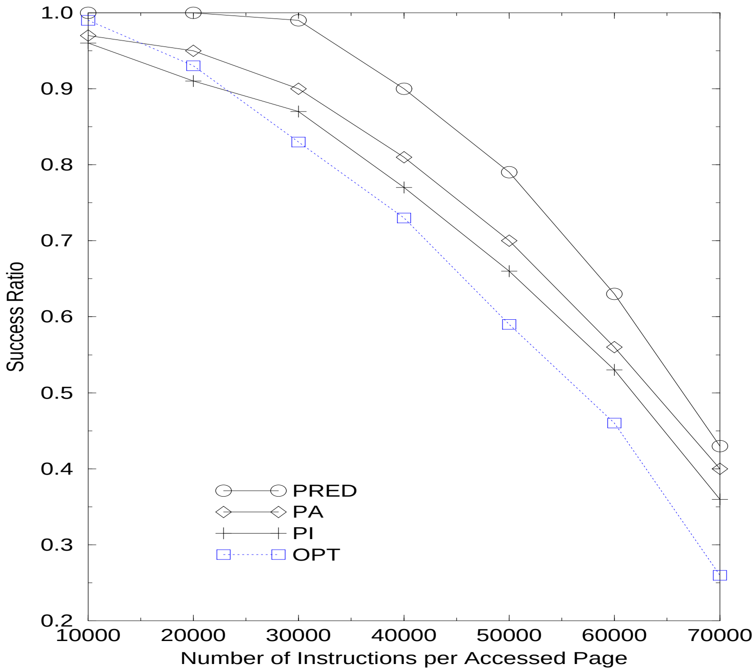

Fig. 7: Real-Time Performance of the Protocols as a Function of the Number of CPU Instructions Used for Each Accessed Page

The results obtained for different values of parameter *instr_data_write* are not presented because the qualitative behavior is similar to that we observed with parameter *instr_data_read*.

Some other experiments were conducted to evaluate the impact of various other system parameters on protocols' behavior. Those parameters include *instr_xact_start* (number of instructions to initiate a transaction) and *instr_xact_terminate* (number of instructions to terminate a transaction). The relative performance of the protocols is not sensitive to varying the values of *instr_xact_start* and *instr_xact_terminate*, except when extremely large values are assigned to these parameters, in which case almost all transactions miss their deadlines under any protocol. The results of another experiment that investigated the effects of parameter *slack_rate* are provided at the end of Section 4.5.

4.3. Sensitivity to Relation Locality

In the experiment of this section, we tested the sensitivity of real-time performance results to the locality of data references. In order to model different levels of relation locality, we varied the values of parameters *relation_access* and *page_access_per_relation*. We used three pairs of values for these parameters. Besides the default values of 3 for *relation_access* and 5 for *page_access_per_relation*, we also used values 5 and 3, respectively, to model a relatively low relation locality, and values 1 and 15, respectively, to model a relatively high relation locality. For all three pairs of values, the average number of pages accessed by each transaction remains the same. Therefore, increasing relation locality enables transactions to process the same number of pages with fewer relation accesses.

Figure 8 displays the results obtained with different levels of relation locality. In this experiment, the mean interarrival time value was fixed at 2 mseconds. Not surprisingly, increasing the relation locality results in an improvement in the performance of protocol PRED, as it is executed at a relation granularity. On the other hand, the performance of other protocols, which are executed at a page granularity, is not affected by the locality, and the performance remains almost constant under different locality levels. An important observation in this experiment is that although the performance of protocol PRED degrades with a relatively low locality as a result of the

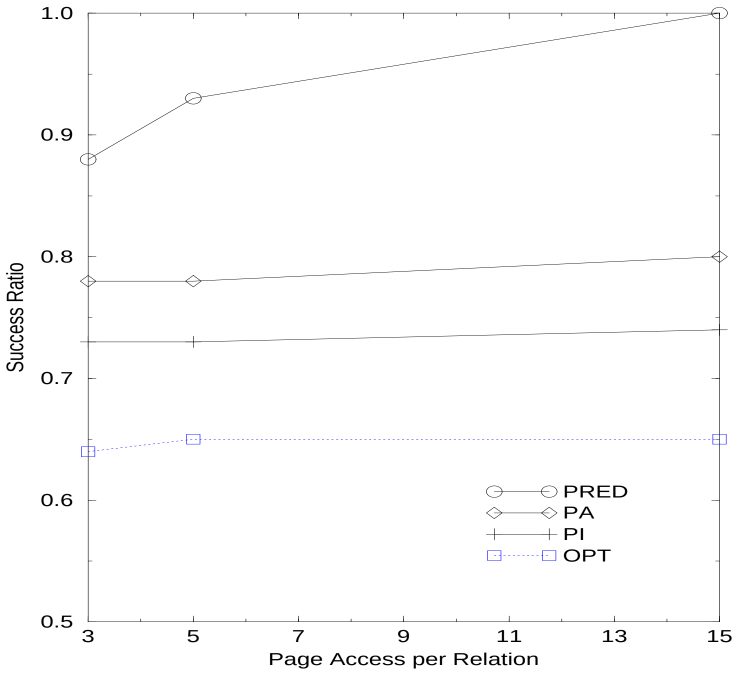

Fig. 8: Real-Time Performance of the Protocols Under Different Levels of Relation Locality.

reduced concurrency (because each transaction locks a larger number of relations), it is still much better than the performance of the other protocols. This result is again due to the differences in implementation overheads that we discussed before.

4.4. The Case Where Entire Database does not Fit in Main Memory

Some critics can argue that the entire database might not always fit in main memory. This is a reasonable argument for some applications that deal with video, image, and voice data. However, as discussed in [5] and [14], memory-resident database management techniques and disk-resident database management techniques can be used together in the same system. For such very large applications, the "hot spots" data model can be adapted: the hot region of the database where most of the references are made can be stored in main memory, while the cold region is kept on disk. [5] provides examples of applications where this partitioning of data arises naturally.

In this experiment, we assume that the hot region of the database which is stored in main memory, contains half of the relations where 80% of the data accesses are directed. The rest of the database is the cold region and stored on a disk. The I/O queue used for accessing the cold region is organized on the basis of the transactions' real-time priorities. Access time to the disk for each data page is uniformly distributed within the range *min_disk_time* through *max_disk_time*. In this experiment, the values used for these two parameters are 10 mseconds and 30 mseconds, respectively. The CPU spends *instr_disk_access* instructions for each I/O operation. The value of this parameter is set to 5000 instructions. If any data in the cold region is updated by a transaction, the updated data is written back to disk at the commit time of the transaction.

Figure 9 displays the performance results obtained using the extended simulation model. Comparing the results against those provided in Figure 1, it can be seen that the performance of each protocol with a disk-resident database system is at a lower level. This result can be attributed to the following two factors: the increased data contention over the hot region, and the delay experienced due to I/O contention. The performance of protocols PA, PI, and OPT is closer, in this case, to that of protocol PRED. The CPU overhead for performing disk I/O takes up a significant portion of the CPU time used by transactions, and this reduces the effective CPU utilization for all protocols including ours.

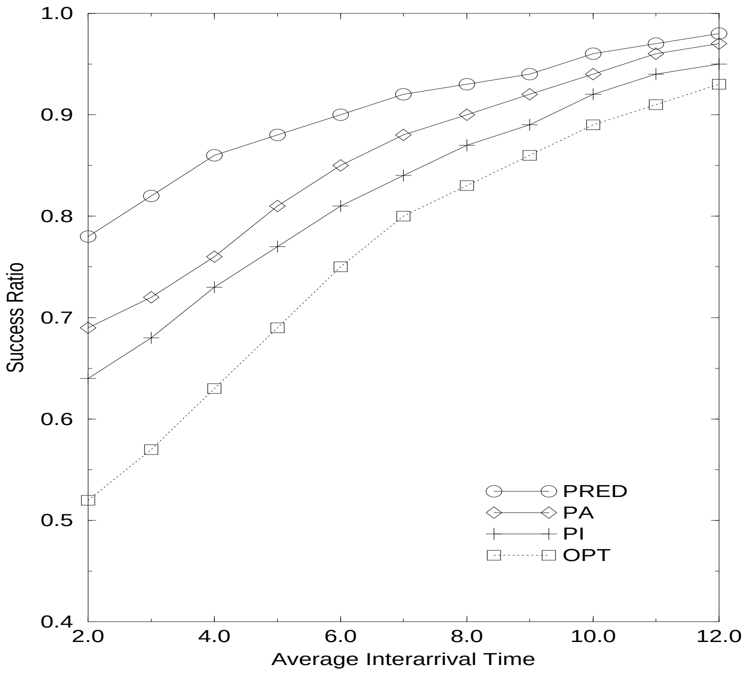

Fig. 9: Real-Time Performance of the Protocols When a Portion of the Database is Disk-Resident.

4.5. A Variant of the Predeclaration Protocol

In protocol PRED, when a transaction gets all the data resources it requires (i.e., when it becomes ready to execute), it holds those resources until it executes. A possible variant of this protocol is to take the resources already allocated to a ready transaction if those resources are requested by a higher-priority transaction. In this variant, it is guaranteed that a high priority transaction always executes before a lower priority transaction that needs the same resources. Assuming that transaction priorities are distinct, the protocol is modified as follows:

For each transaction T submitted to the system,
OFF-LINE:
Parse transaction T, identify the relations to be accessed, and construct *read_set[T]* and *write_set[T]*.
ON-LINE:
Set *conflicting_transactions[T]* to empty.
For each transaction T' that is executing or (is in the *ready* or *wait-queue* with priority(T') > priority(T)),
If ($read\_set[T] \cap write\_set[T']$) $\neq \emptyset$ OR ($write\_set[T] \cap (read\_set[T'] \cup write\_set[T'])$) $\neq \emptyset$
insert T' into *conflicting_transactions[T]*
EndFor.
If *conflicting_transactions[T]* is empty
insert T into *ready-queue*
else,
insert T into *wait-queue*
EndIf.
For each transaction T' in the *ready* or *wait-queue* with priority(T') < priority(T),
If ($read\_set[T'] \cap write\_set[T]$) $\neq \emptyset$ OR ($write\_set[T'] \cap (read\_set[T] \cup write\_set[T])$) $\neq \emptyset$
insert T into *conflicting_transactions[T']*
If T' is a ready transaction and *conflicting_transactions[T']* becomes nonempty,
transfer T' from the *ready-queue* to the *wait-queue*
EndIf.
EndFor.

We compared the performance of two variants of the protocol in terms of *success_ratio* through a set of experiments. We observed in one of those experiments that the new variant of protocol PRED can outperform the original one under the condition that transactions are executed with small slack times. When we evaluated all the protocols with varying values of parameter *slack_rate* (average rate of slack time of a transaction to its processing time), we also observed that the performance of the original PRED is comparable to that of protocol PA for small values of *slack_rate*. The performance benefit of PRED over PA is outweighed by the fact that with protocol PA a high priority transaction (with closer deadline) is never blocked by a lower priority one. Blocking a high priority transaction is not a problem if the slack time of the transaction is large enough to wait for the completion of the conflicting transaction. Otherwise, the transaction misses its deadline. The new variant of PRED does not have that problem with small slack times as it guarantees that a high priority transaction is always executed before a lower priority transaction that has some conflicting access requests. Performance results of these protocols are presented in Figure 10. The mean interarrival time (*iat*) value used in this experiment was 5 mseconds. The performance of each of protocols PI and OPT was at a lower level compared to that of other protocols for each value of *slack_rate* employed in this experiment.

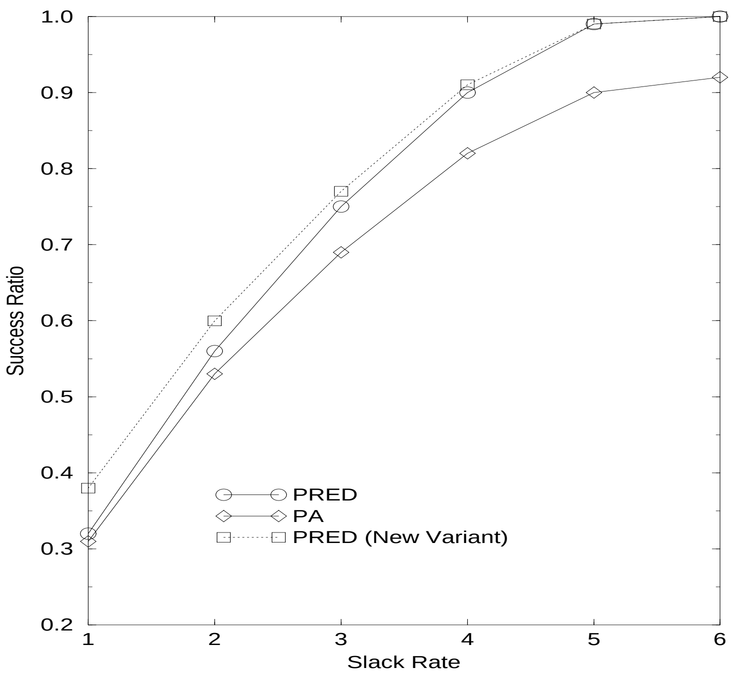

Fig. 10: Performance Results as a Function of the Average Rate of Slack Time of a Transaction to Its Processing Time.

5. CONCLUSIONS

In this paper, we have presented a new concurrency control protocol for main-memory real-time database systems (RTDBSs). Unlike other protocols proposed recently for RTDBSs, our scheme is based on predeclaration of data resources and implemented at a relation granularity. The development of the protocol was motivated by the following facts:

- With dynamic acquisition of data resources, the predictability of transaction execution is reduced due to the possibility of blocking, deadlock and rollback.
- It is simple to determine by purely syntactic means during transaction compilation the data access set of a transaction at the relation level.

- Large locking granularities are more efficient in main-memory databases.

In the proposed protocol, for each new transaction submitted to the system, the set of relations to be accessed by the transaction are determined. Then, a conflict check is performed between the relations to be accessed by the new transaction and the relations in the access list of already scheduled transactions. If no conflict is detected, the transaction is scheduled by preacquiring the necessary relations. Otherwise, it is blocked until the conflicting locks on its relations are released. We compared the performance of the protocol against some other RTDBS concurrency control protocols through a detailed simulation. The performance metric used in the simulation experiments was the fraction of transactions that satisfy their timing constraints. This metric was imposed by the lack of predictability of the other compared algorithms. The results obtained from the experiments can be summarized as follows:

- Our protocol provides much better performance compared to the other protocols, as it wastes less CPU time for handling data conflicts and lock management.
- Increasing the relation locality improves the performance of our protocol. Even with very low localities, it still performs much better than the other protocols.
- The protocol is clearly preferable to other protocols with disk-resident databases as well.
- We also proposed a variant of the protocol in which a high-priority transaction is allowed to preempt the relations already allocated to lower-priority transactions in the ready queue. It was shown through experiments that this variant of the protocol can provide better performance if the transactions executed are characterized by small slack times.

We have presented our protocol in the presence of soft transaction deadlines. However, we believe that the potential of the protocol lies not only in the fact that under equal circumstances it is more efficient than other concurrency control protocols proposed for RTDBSs, but the fact that it is possible to give harder guarantees as we do not use dynamic resource allocation. Two basic properties of the protocol as discussed above are:

- using predeclaration, and
- using a main-memory approach.

Our claim is that hard guarantees can be provided by the protocol if the following additional properties are supported:

- setting upper bounds for the relation sizes, and
- using predictable indexing.

Acknowledgements — Özgür Ulusoy was supported by the Research Council of Turkey (TÜBİTAK) grant no. EEEAG-246.

REFERENCES

[1] R. Abbott and H. Garcia-Molina. Scheduling real-time transactions: a performance evaluation. *ACM Transactions on Database Systems*, **17**(3): 513–560 (1992).

[2] A. Bestavros and S. Braudakis. Value-cognizant speculative concurrency control for real-time databases. *Information Systems*, **21**(1):75–101 (1996).

[3] S.R. Biyabani, J.A. Stankovic, and K. Ramamritham. The integration of deadline and criticalness in hard real-time scheduling. In *Proceedings of the 9th Real-Time Systems Symposium*, pp.152–160 (1988).

[4] L.C. DiPippo and V.F. Wolfe. Object-based semantic real-time concurrency control. In *Proceedings of the 14th Real-Time Systems Symposium*, pp.87–96 (1993).

[5] H. Garcia-Molina and K. Salem. Main-memory database systems. *IEEE Transactions on Knowledge and Data Engineering*, **4**(6):509–516 (1992).

[6] J.R. Haritsa, M.J. Carey, and M. Livny. Dynamic real-time optimistic concurrency control. In *Proceedings of the 11th Real-Time Systems Symposium*, pp.94–103 (1990).

[7] J.R. Haritsa, M.J. Carey, and M. Livny. *Value-Based Scheduling in Real-Time Database Systems.* Technical Report 1204, Department of Computer Science, University of Wisconsin-Madison (1991).

[8] D. Hong, S. Chakravarthy, and T. Johnson. Locking based concurrency control for integrated real-time database systems. In *Proceedings of the First Workshop on Real-Time Databases: Issues and Applications* (1996).

[9] J. Huang, J.A. Stankovic, D. Towsley, and K. Ramamritham. Experimental evaluation of real-time transaction processing. In *Proceedings of the 10th Real-Time Systems Symposium*, pp.144–153 (1989).

[10] J. Huang, J.A. Stankovic, K. Ramamritham, and D. Towsley. On using priority inheritance in real-time databases. In *Proceedings of the 12th Real-Time Systems Symposium*, pp.210–221 (1991).

[11] B. Kao and H. Garcia-Molina. An overview of real-time database systems. In *Advances in Real-Time Systems*, S.H. Son, editor, Prentice-Hall, Englewood Cliffs, New Jersey (1995).

[12] K.Y. Lam and S.L. Hung. Concurrency control for time-constrained transactions in distributed database systems. *The Computer Journal*, **38**(9):704–716 (1995).

[13] T.J. Lehman and M.J. Carey. A recovery algorithm for a high-performance memory-resident database system. In *Proceedings of the ACM SIGMOD Conference*, pp.104–117 (1987).

[14] T.J. Lehman, E.J. Shekita, and L. Cabrera. An evaluation of starburst's memory resident storage component. *IEEE Transactions on Knowledge and Data Engineering*, **4**(6):555–566 (1992).

[15] P.E. O'Neil, K. Ramamritham, and C. Pu. A Two-phase approach to predictably scheduling real-time transactions. In *Performance of Concurrency Control Algorithms in Centralized Database Systems*, V. Kumar, editor, Prentice-Hall (1996).

[16] H. Schwetman. CSIM: a C-based, process-oriented simulation language. In *Proceedings of the Winter Simulation Conference*, pp.387–396 (1986).

[17] L. Sha, R. Rajkumar, S.H. Son, and C.H. Chang. A real-time locking protocol. *IEEE Transactions on Computers*, **40**(7):793–800 (1991).

[18] S.H. Son, S. Park, and Y. Lin. An integrated real-time locking protocol. In *Proceedings of the International Conference on Data Engineering*, pp.527–534 (1992).

[19] A. van Tilborg and G. Koob, editors. *Foundations of Real-Time Computing, Scheduling, and Resource Management.* Kluwer Academic Publishers (1991).

[20] Ö. Ulusoy and G.G. Belford. Real-time transaction scheduling in database systems. *Information Systems*, **18**(8):559–580 (1993).

[21] Ö. Ulusoy. Research issues in real-time database systems. *Information Sciences*, **87**(1–3):123–151 (1995).