An Evaluation of a Client-Server Real-Time Database System*

Özgür Ulusoy
Department of Computer Engineering and Information Science
Bilkent University
Bilkent, Ankara 06533, TURKEY

Abstract

A real-time database system (RTDBS) can be defined as a database system where transactions are associated with real-time constraints. In this paper, we investigate various performance issues in a RTDBS constructed on a client-server system architecture. In a client-server database management system the whole database is stored on the server disks, and copies of database items can be cached in the client memories. We provide a detailed simulation model of a client-server RTDBS, and present the initial results of a performance work that evaluates the effects of various workload parameters and design alternatives.

Index Terms - *Real-time database systems, client-server architecture, transaction scheduling, timing constraints.*

1 Introduction

Real-time database systems (RTDBSs) are designed to provide timely response to the transactions of data-intensive applications. Many properties from both real-time systems and database systems have been inherited by RTDBSs. Similar to a conventional real-time system, transactions processed in a RTDBS are associated with timing constraints, usually in the form of deadlines. Access requests of transactions to data or other system resources are scheduled on the basis of the timing constraints. What makes a RTDBS different from a real-time system is the requirement of preserving the logical consistency of data in addition to considering the timing constraints of transactions. The requirement to maintain data consistency is the essential feature of a conventional database system. However, the techniques used to preserve data consistency in database systems are all based on transaction blocking and transaction restart, which makes it virtually impossible to predict computation times and hence to provide schedules that guarantee deadlines in a RTDBS. As a result, it becomes necessary to extend traditional database management techniques with time-critical scheduling methods. While the basic scheduling goal in a conventional database system is to minimize the response time of transactions and to maximize throughput, a RTDBS scheduler primarily aims to maximize the number of transactions that satisfy their deadlines. A priority is assigned to each RTDBS transaction based on its timing constraint to be used in ordering resource and data access requests of transactions. An extensive exploration of the issues in concurrency control and other priority-cognizant scheduling concepts, such as buffer management, I/O scheduling, commitment, etc., is provided in [10].

The research on distributed RTDBSs has focused on development and evaluation of new time-cognizant scheduling techniques that can provide good performance in terms of the fraction of satisfied timing constraints (e.g., [6, 7, 8, 9]). However, all those performance works have assumed a database system completely distributed over individual data sites. In this paper, we study a RTDBS that executes on a client-server architecture. In a client-server database system, the whole database is stored on the disks connected to the servers. Data access requests received from the clients are handled by the servers. Part of the main memory of each client can be used to cache a small portion of the database. Caching the copies of database items provide faster access for the clients; however, it leads to a requirement of using some mechanisms to provide the consistency of multiple copies of the cached data. The mechanisms used for that purpose are called the "cache consistency algorithms". Among various cache consistency algorithms proposed for client-server database systems, the "Callback Locking" algorithm is the most popular one [4]. In this algorithm, a client requires to obtain a lock from the server before accessing a data item. If the requested lock conflicts with one or more locks currently held by various clients, the server sends a "callback" message to each of those clients. The lock can be granted to the requesting client when all the conflicting locks are released.

We describe a detailed simulation model designed for studying various performance issues in client-server RTDBSs. Performance of different priority-cognizant concurrency control protocols is studied under a range of workloads using the simulation model. The per-

*This work was supported by the Research Council of Turkey (TÜBİTAK) under grant number EEEAG-137.

formance results are compared against the results obtained by implementing the protocols on a completely distributed database system. We also investigate various performance characteristics of the client-server RTDBS under different system configurations and workloads.

The remainder of the paper is organized as follows. The next section summarizes the recent work in RTDBSs and client-server database systems. Section 3 describes our client-server RTDBS simulation model. The results of the performance evaluation experiments are provided in Section 4. The last section summarizes the conclusions of our work.

2 A Client-Server RTDBS Model

The client-server RTDBS model simulates a data-shipping page server; i.e., the unit of interaction between the server and the clients is a page, and the copies of the pages are transmitted to the clients to be processed by transactions. Clients generate transactions and request pages for the execution of the transactions. Their workload is derived from these transactions. Server's workload is generated by the requests coming from the clients.

The global memory hierarchy of the system consists of the client memory, the server memory, and the server disk (where the database resides). Clients obtain page locks from the server. All concurrency control and cache consistency maintenance is implemented at a page granularity. Database pages with corresponding locks are cached in clients' memories. Cache consistency is provided through the use of the callback locking scheme. To reduce disk accesses, we use the *forwarding* technique proposed by Franklin [3] which works as follows: When a transaction executing at a client needs to access a data page, the client first searches through its local cache. If the data page does not reside in the local cache, the client requests the page from the server. The server checks to see if the page is in its memory. If the page does not exist in the server's cache, the page request message is forwarded to a client (if any) that has a local copy of the page. The server keeps track of the information that where the copies of each data page reside in the system. Upon receipt of a forwarded request, the client sends the copy of the page to the requesting client. Therefore, the forwarded request message prevents the disk I/O at the server. It was shown in [3] that the forwarding technique can provide significant performance gains considering that in today's systems, disk access delay, rather than network delay, is the performance bottleneck.

Each client in the system contains a transaction generator, a client manager, a scheduler, a memory manager, and a resource manager. The transaction generator is responsible for generating the transaction workload for each client. The arrivals at a client are assumed to be independent of the arrivals at the other clients. Each transaction in the system is distinguished by a globally unique transaction id. The id of a transaction is made up of two parts: a transaction number which is unique at the originating client of the transaction and the id of the originating client which is unique in the system. Each transaction is assigned a real-time constraint in the form of a deadline. The transaction deadlines are ***soft***; i.e., each transaction is executed to completion even if it misses its deadline. Data pages to be accessed by each transaction are also determined by the transaction generator.

The client manager at the originating client of a transaction assigns a real-time priority to the transaction based on the ***earliest deadline first*** priority assignment policy; i.e., a transaction with an earlier deadline has higher priority than a transaction with a later deadline. The client manager processes the page access requests of the transactions. If a data page needed by a transaction does not reside in the local cache, the client manager requests the page from the server. At the commit time of a transaction, the client manager generates a commit message to the server along the list of pages updated by the transaction. Updates of a committed transaction are stored in the server's disk.

The scheduler of each client provides the cache consistency and concurrency control at that site. The buffer manager is responsible for the management of the client's memory. It makes use of the Least Recently Used (LRU) algorithm for the replacement of pages in the memory. Finally, the resource manager provides CPU service for processing data pages and communication messages.

The server has the same components as the clients except that it does not contain a transaction generator since we assume that the transaction workload of the system is generated at the clients. The server manager processes and responds to the messages received from the clients. The scheduler processes the page access requests coming from the clients, and responds to those requests on the basis of the concurrency control protocol executed. The responsibilities of the buffer manager are similar to those of the buffer manager at each client. The resource manager of the server provides I/O service as well as CPU service. Both CPU and IO queues are organized on the basis of real-time priorities, and preemptive-resume priority scheduling is used by the CPU.

A network manager is also included in the model to simulate a local area network connecting the clients and the server. Reliability and recovery issues are not addressed in this paper. We assume a reliable system, in which no site failures or communication network failures occur.

For the detection of deadlocks a wait-for graph is maintained at the server. Periodic detection of deadlocks is the server's responsibility. A deadlock is recovered from by selecting the lowest priority transaction in the deadlock cycle as a victim to be aborted.

The list of parameters described in Table 1 was used in specifying the configuration and workload of the client-server RTDBS. The parameters were basicly adapted from the client-server database management system model used in Franklin's experiments [3].

Access time to the server disk is uniformly distributed within the range *MinDiskTime* through *MaxDiskTime*. The server CPU spends *DiskOverheadInst* instructions for each I/O operation. A control message is a non-data message like commit, abort,

CONFIGURATION PARAMETERS		
NoOfClients	Number of clients	1 to 25
DatabaseSize	Size of the database in pages	1250 pages
PageSize	Page size in bytes	4,096 bytes
ServerMemSize	Number of pages that can be held in the server's memory	25% of the *DatabaseSize*
ServerMIPS	Instruction rate of CPU at the server	100 MIPS
ClientMemSize	Number of pages that can be cached in each client's memory	10% of the *DatabaseSize*
ClientMIPS	Instruction rate of CPU at each client	50 MIPS
MinDiskTime	Minimum disk access time	10 milliseconds
MaxDiskTime	Maximum disk access time	30 milliseconds
NetworkBandwidth	Network Bandwidth	8 or 80 Mbits/sec
FixedMsgInst	Fixed number of instructions to process a message	20,000 instructions
PerByteMsgInst	Additional number of instructions per message byte	10,000 inst. per 4Kb
ControlMsgSize	Control message size in bytes	256 bytes
DeadlockInterval	Deadlock detection frequency	1 second
DiskOverheadInst	CPU overhead for performing disk I/O	5000 instructions
SystemOverhead	Number of instructions for certain system operations	300 instructions
TRANSACTION PARAMETERS		
ThinkTime	Think time between client transactions	0
TransactionSize	Number of pages accessed per transaction	20 pages
StartTransInst	Number of instructions to initialize a transaction	30,000 instructions
EndTransInst	Number of instructions to terminate a transaction	40,000 instructions
ReadPageInst	Number of CPU instructions per page read	30,000 instructions
WritePageInst	Number of CPU instructions per page write	60,000 instructions
HotBounds	Page bounds in the hot region	for client n, p to p+49 where, p=50(n-1)+1
ColdBounds	Page bounds in the cold region	rest of the database
HotAccessProb	Probability of access to a page in the hot region	0.8
WriteProb	Probability of writing to a page	0.2
SlackRate	Average rate of slack time of a transaction to its processing time	5

Table 1: Client-Server RTDBS Model Parameters

lock-request, lock-grant messages etc., and the size of such messages is specified by *ControlMsgSize*. The parameter *SystemOverhead* is used to simulate the CPU overhead of various operations performed. These operations include locking/unlocking, conflict check, locating a data page, etc.

The workload model specifies a "hot region" in the database for each client, where most of the references are made. The rest of the database is specified as the "cold region". With a probability of *HotAccessProb*, a page to access is chosen from the hot region of the database.

The slack time of a RTDB transaction specifies the maximum length of time the transaction can be delayed and still satisfy its deadline. In our system, the transaction generator at each client chooses the slack time of a transaction randomly from an exponential distribution with a mean of *SlackRate* times the estimated minimum processing time of the transaction. Although the transaction generator uses the estimation of transaction processing times in assigning deadlines, we assume that the system itself lacks the knowledge of processing time information.

3 Performance Experiments

The simulation program was written in CSIM [5], which is a process-oriented simulation language based on the C programming language. The simulation results were evaluated as averages over 10 independent runs. Each run was continued until 10,000 transactions were processed in the system. 90% confidence intervals were obtained for the performance results. The width of the confidence interval of each statistical data point is less than 4% of the point estimate. In displayed graphs, only the mean values of the results are plotted.

The values of configuration and transaction parameters common to all simulation experiments are presented in Table 1. The parameter values were chosen so as to be comparable to the related simulation studies such as [3]. *NetworkBandwidth* values specified (i.e., 8 Mbps and 80 Mbps) roughly correspond to Ethernet and FDDI networks, respectively [2]. All the clients are assumed identical and operating under the same parameter values.

The performance metric used in the evaluations is *success_ratio*; i.e., the fraction of transactions that satisfy their deadlines. The workload model used in our experiments simulates an environment in which there exist some amount of data contention among the clients and locality of data accesses at each client. Such a workload is expected to be common in client-server RTDBSs. In this workload, each client has a 50 page hot region where their 80% of the page accesses are directed. The hot region of each client is distinct from the hot regions of the other clients; however, the sharing of pages is provided by overlapping the hot region of each client with the cold regions of the other clients.

3.1 Evaluation of Some Concurrency Control Protocols

In [8], we evaluated the performance of a number of RTDBS concurrency control protocols in a distributed database system environment. The same experiment is repeated here to see how the results obtained are affected when the protocols are implemented in a client-server database system. In this section, we first provide a brief description of the protocols selected for evaluation, and then discuss the performance results obtained using our client-server RTDBS model. We do not intend to provide a detailed evaluation of concurrency control protocols proposed for RTDBSs; rather, we try to observe how the selected protocols behave on a client-server architecture.

The Base Protocol: The base protocol is the pure implementation of the two-phase locking (2PL) algorithm. Real-time priorities of the transactions are not considered by the server in processing their page access requests. 2PL serves as the base protocol to enable us to measure the performance benefits of the priority-based protocols.

Priority Inheritance Protocol (PI): The priority inheritance method, proposed in [6], 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.

Priority Abort Protocol (PA): 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.

Figure 1 displays the *success_ratio* results for the concurrency control protocols as a function of the number of clients. The results have been obtained by setting *NetworkBandwidth* to 8 Mbits/sec. An increase in the number of clients leads to an increased global memory size (which is composed of the server's and the clients' memories), and thus the amount of the database that can be kept in the global memory increases. Due to the forwarding technique used in our model, an increased size of global memory results

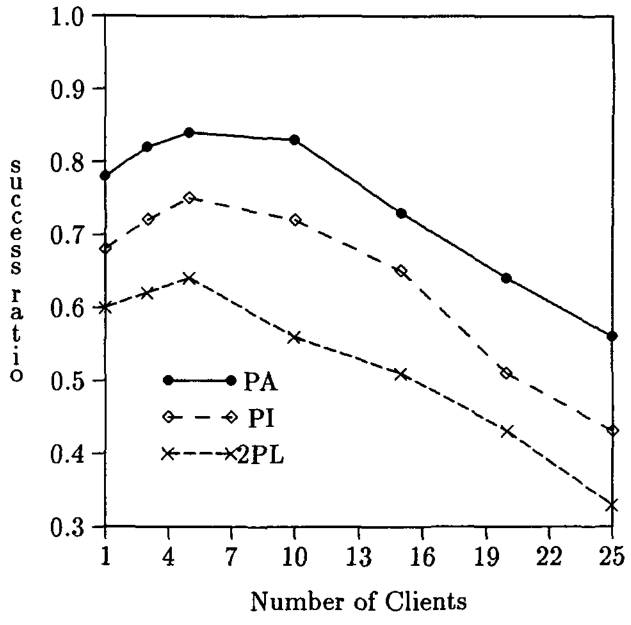

Figure 1: Performance of the concurrency control protocols.

in less amount of I/O at the server, and thus the performance becomes better for all three protocols. However, the improvement in the performance is possible upto a certain number of clients, and beyond that point, an increase in the number of clients results in a performance degradation. The reason for that result is the fact that increasing the number of clients also leads to an increase in data contention among transactions. And, after a certain number of clients, the performance advantage gained due to the larger global memory is outweighed by the overhead of data contention.

If we compare the results obtained by the concurrency control protocols, we can see that the base protocol 2PL exhibits the worst performance under varying numbers of clients. This result is not surprising, as 2PL does not include real-time priorities of transactions in data access scheduling decisions. Protocols PI and PA both provide a considerable improvement in real-time performance over 2PL. Between PI and PA, the performance of PA is consistently better than that of PI for all experimented values of *NoOfClients*. Remember that PA never blocks higher priority transactions, but instead aborts low priority transactions when necessary. PA also eliminates the possibility and cost of deadlocks. It can be concluded that aborting a low priority transaction is preferable in RTDBSs to blocking a high priority one, even though aborts lead to a waste of resources.

In [8], we evaluated the performance of the concurrency control protocols in a RTDBS where the database is completely distributed over the sites. In that work, each transaction was modeled as a master process that executes at the originating site of the transaction and a number of cohort processes that ex-

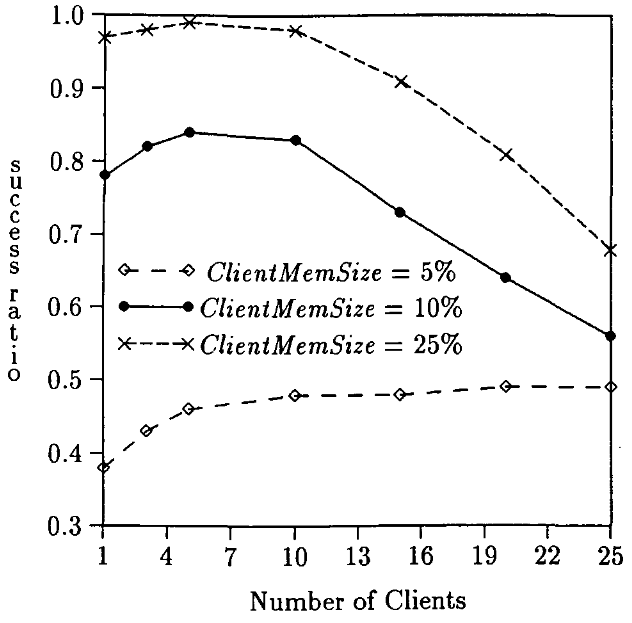

Figure 2: Performance with different client memory sizes (*NetworkBandwidth* = 8 Mbps).

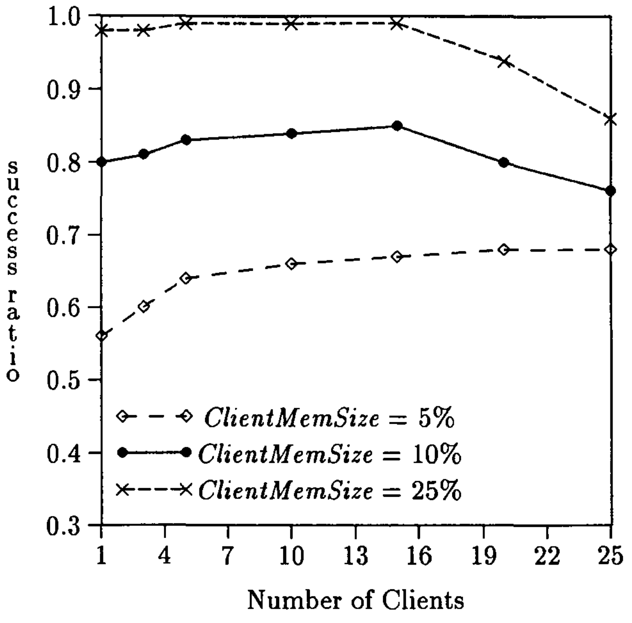

Figure 3: Performance with different client memory sizes (*NetworkBandwidth* = 80 Mbps).

ecute at various sites where the the copies of required data items reside. The results obtained in that work for the comparative performance of the concurrency control protocols also showed that protocol PA can provide better performance than PI. However, the improvement in that case was quite marginal (much less than the improvement observed with the client-server RTDBS model). We think that this result is due to the fact that the overhead of transaction aborts experienced with PA becomes less if the protocol is implemented on a client-server architecture, since the abort of a transaction is handled completely at the originating client of the transaction. The other clients are not involved in that process, and thus the communication overhead between the clients is avoided.

3.2 Impact of Memory Size and Network Speed

In this section, the performance results obtained in terms of the *success_ratio* are presented for different values of *ClientMemSize* (i.e., number of pages that can be cached in each client's memory), *ServerMemSize* (i.e., number of pages that can be held in the server's memory), and *NetworkBandwidth.* The ranges of *ClientMemSize* and *ServerMemSize* experimented are [5%, 25% of *DatabaseSize*] and [10%, 100% of *DatabaseSize*], respectively. The concurrency control protocol PA is employed in these experiments.

Figure 2 presents the performance results obtained with different values of *ClientMemSize.* For a small client memory size (i.e., 5% of *DatabaseSize*), it is possible to improve the performance with each additional client connected to the system. As the global memory size of the system increases, more data pages can be cached, and more page access requests can be satisfied without disk access. However, for larger client memory sizes experimented (i.e., 10% and 25% of *DatabaseSize*), increasing the number of clients beyond a certain point leads to a degradation in the performance. For those values of *ClientMemSize*, the peak values obtained for *success_ratio* corresponds to the points at which most of the data pages are cached in the global memory. Increasing the global memory by adding more clients beyond that point does not lead to much decrease in disk I/O; rather, the increasing contention among the larger number of concurrent transactions becomes the determining factor for the performance.

Besides disk I/O and data contention, another factor that has a significant impact on the performance is the network speed. The results obtained with a fast network (i.e., *NetworkBandwidth* = 80 Mbps) are displayed in Figure 3. Since the message delay experienced by the transactions becomes much less, a considerable performance improvement is observed for all values of *ClientMemSize* and *NoOfClients.* Also, compared to the results obtained with the slow network, the peak performance value with any *ClientMemSize* is obtained for larger number of clients and the drop after the peak value is not that steep. As expected, employing a fast network reduces the adverse effects of high transaction loads on the performance.

The performance impact of the server memory size is presented for three different values of *ServerMemSize.* The client memory size in these experiments is fixed at 10% of the database size. The results obtained using the slow network are shown in Figure 4. As the size of the server memory increases, the server hit ratio also increases; i.e., a larger number of page access requests of the clients can be satisfied by the server without requiring disk I/O or forwarding the requests

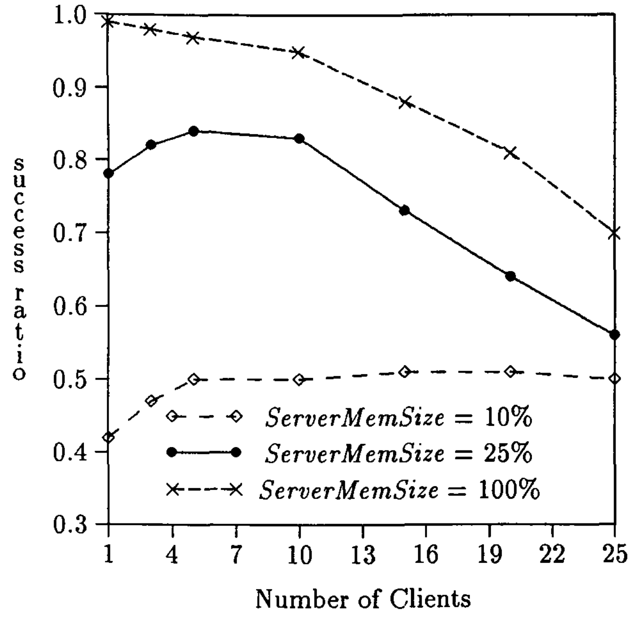

Figure 4: Performance with different server memory sizes (*NetworkBandwidth* = 8 Mbps).

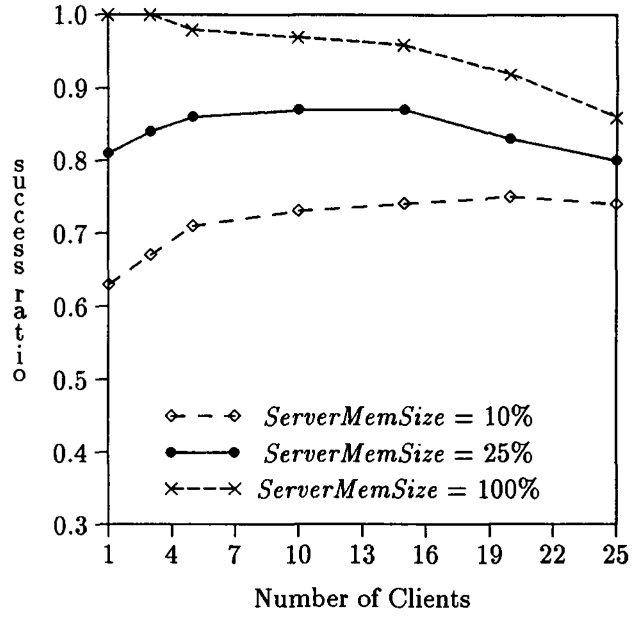

Figure 5: Performance with different server memory sizes (*NetworkBandwidth* = 80 Mbps).

of a client to the other clients. Remember that, if the server has a copy of the requested page in its memory, it directly sends the page to the requesting client. If, however, its memory does not contain that page but any of the clients has a cached copy of the page, then the page access request is forwarded to that client.

For the extreme case where the server memory size is equal to the database size, all the page access requests are satisfied from the server memory. Disk I/O is required only to store the updates in the database. Having larger global memory size by adding more clients to the system does not have an effect on the amount of disk I/O. Thus, increasing the number of clients just leads to an increase in the contention among the transactions, and the performance becomes worse. For smaller server memory sizes (e.g., *ServerMemSize* = 25% of *DatabaseSize*), it is possible to improve the performance by adding more clients if the number of clients is small. For such cases, an increase in the global memory size (as a result of increasing the number of clients) prevents some of disk I/Os. As can be seen from the figure, the system benefits the most from having more clients when the server memory size small (i.e., 10% of the database size in our experiments).

Whan the underlying network is fast, the comparative performance results obtained with different server memory sizes do not change much (see Figure 5). However, similar to the results obtained by varying *ClientMemSize*, the steep decrease in the performance with large numbers of clients is prevented by employing a fast network.

4 Conclusions

In this paper, we have investigated various performance issues in a client-server RTDBS. We have provided a detailed simulation model of a client-server RTDBS, and presented the initial results of a performance work that evaluates the effects of various workload parameters and design alternatives. The performance metric used in the evaluations is the fraction of transactions that satisfy their timing constraints.

In the first part of the evaluations, we have studied performance of different priority-cognizant concurrency control protocols. Comparing the results to those obtained by implementing the protocols on a completely distributed database system, we have observed that the protocols that resolve data conflicts by aborting low priority transactions are more appropriate for client-server RTDBSs. The overhead of transaction aborts is less with the client-server architecture, since the abort of a transaction is handled completely at the originating client of the transaction. Our other evaluations have investigated the impact of various configuration parameters on the performance of the client-server RTDBS. Those evaluations have helped us identify the ranges of workloads for which each setting of the parameters provides improvements in the performance.

References

[1] R. Abbott, H. Garcia-Molina, 'Scheduling Real-Time Transactions: A Performance Evaluation', *ACM Transactions on Database Systems*, vol.17, no.3, pp.513-560, 1992.

[2] M.J. Franklin, M.J. Carey, *Client-Server Caching Revisited*, Computer Sciences Technical Report

no. 1089, University of Wisconsin-Madison, 1992.

[3] M.J. Franklin, *Caching and Memory Management in Client-Server Database Systems*, Computer Sciences Technical Report no. 1168, University of Wisconsin-Madison, 1993.

[4] J. H. Howard et al., 'Scale and Performance in a Distributed File System', *ACM Transactions on Computer Systems*, vol.6, no.1, pp.51-81, 1988.

[5] H.Schwetman, 'CSIM: A C-Based, Process-Oriented Simulation Language', *Proceedings of the Winter Simulation Conference*, pp.387-396, 1986.

[6] L.Sha, R.Rajkumar, S.H.Son, C.H.Chang, 'A Real-Time Locking Protocol', *IEEE Transactions on Computers*, vol.40, no.7, pp.793-800, 1991.

[7] S.H.Son, C.H.Chang, 'Performance Evaluation of Real-Time Locking Protocols Using a Distributed Software Prototyping Environment', *International Conference on Distributed Computing Systems*, pp.124-131, 1990.

[8] Ö. Ulusoy, G.G. Belford, 'Real-Time Lock Based Concurrency Control in a Distributed Database System', *International Conference on Distributed Computing Systems*, pp.136-143, 1992.

[9] Ö. Ulusoy, 'Processing Real-Time Transactions in a Replicated Database System', *Distributed and Parallel Databases*, vol.2, no.4, pp.405-436, 1994.

[10] Ö. Ulusoy, 'Research Issues in Real-Time Database Systems', to appear in *Information Sciences*, 1995.