Site Selection for Real-Time Client Request Handling*

Vinay Kanitkar and Alex Delis
Department of Computer and Information Science
Polytechnic University
Brooklyn, NY 11201

Abstract

In a conventional client-server database system (CS-DBS), a transaction and its requisite data have to be colocated at a single site for the operation to proceed. This has traditionally been achieved by moving either the data or the transaction. Today, the availability of powerful workstations and high-bandwidth networking options has led users to expect real-time guarantees about the completion times of their tasks. So as to offer such guarantees in a CS-DBS, a transaction should be processed by any means that allows it to meet its deadline. In this paper, we explore the option of moving both transactions and data to the most promising sites for successful completion. We propose a load-sharing mechanism that oversees the shipment of data and transactions in order to increase the efficiency of a client-server cluster. Additionally, we make use of the concept of grouped locks to schedule the movement of data objects in the cluster in a more efficient manner. An experimental evaluation shows that the use of our load-sharing algorithm provides a considerable improvement in the real-time processing efficiency of a CS-DBS even in the presence of very high volumes of update transactions.

1 Introduction

Contemporary systems are expected to provide distributed data access and support real-time constraints on individual tasks. Such systems are found in many diverse application domains including financial environments, network management and process control systems as well as computer integrated manufacturing. In many such application areas, client-server database systems (CS-DBS) have utilized increasingly powerful processing capabilities and network bandwidths to successfully manage data and provide high transaction throughput. However, real-time transaction processing in a CS-DBS has not been examined in much detail, and presents an important new challenge. Transaction processing in a CS-DBS has generally been performed either by sending the transaction to the location of the data (transaction-shipping) [9, 15], or by moving the data to the location of the transaction (object-shipping) [6, 16]. Although both these techniques offer several advantages, individually they have limited flexibility for use in general real-time application environments. In this paper, we propose a new framework that ships the data or the transaction or both to the site that is most likely to execute the transaction within its deadline.

In real-time databases, a transaction completes successfully only if it finishes its execution within a pre-specified deadline. Deadlines are introduced to satisfy quality of service requirements or control the operation of physical systems. Therefore, the key measure of performance is the percentage of all transactions that complete within their deadlines rather than the average transaction response time or throughput. We have previously observed [13] that a client-server real-time database system (CS-RTDBS) can be more efficient than a centralized system in the presence of the following conditions: (i) if there is a reasonable amount of spatial and temporal locality in client data access patterns, and (ii) the percentage of data accesses that are updates is low. If the number of updates is large then centralized systems demonstrate better performance than their client-server counterparts. This is because a large volume of updates increases the overhead incurred in coordinating distributed concurrency and shipping objects among sites significantly. Thus, transactions at client sites are forced to block for long periods of time waiting for their required data objects and locks to become available.

In order to avoid such potential delays, we propose a load-sharing algorithm that can substantially improve real-time processing in a CS-RTDBS. Unlike the basic object-shipping client-server model [6, 16, 25, 26], we introduce an approach which performs either transaction-shipping or object-shipping or both in order to speed up processing of real-time transactions. This flexible approach offers greater

*This work was supported in part by the National Science Foundation under Grant NSF IIS-9733642 and the Center for Advanced Technology in Telecommunications, Brooklyn, NY.

opportunities for timely transaction completion. The site to which a transaction is shipped is decided by means of heuristics that combine the availability of data and the current processing load of that site. To test our hypothesis, we have developed detailed prototypes of the three configurations: centralized (CE-RTDBS), basic client-server (CS-RTDBS), and load-sharing client-server (LS-CS-RTDBS). The results of our experiments show that the load-sharing algorithm in the LS-CS-RTDBS allows it to demonstrate considerable performance gains over the CE-RTDBS and CS-RTDBS, even when the percentage of updates is heavy. We also confirm that, when update transactions are sparse, the basic CS-RTDBS can provide better performance than its centralized counterpart.

The rest of the paper is organized as follows. The next section discusses the used system models. In Section 3, we present the basic techniques of transaction-shipping, transaction decomposition, and object migration. Section 4 outlines our load-sharing algorithm and our experimental evaluation is described in Section 5. Section 6 describes related work and the last section contains our conclusions.

2 System Features and Real-Time Aspects

In a centralized architecture, the database server performs all the transaction processing. Clients are assumed to be simple terminals and act as user-interface points only. User transactions are initiated at the clients and are forwarded to the server for execution. Once they arrive at the server, the real-time scheduler assigns priorities to them and executes them in that order. All transactions are scheduled according to a single global schedule. The results of executing the transactions are communicated to the users through the clients.

In object-shipping CS architectures, transactions are executed at client sites. If an object referenced by a transaction is not cached locally (either main memory or disk) then it is fetched from the server. The locally available disk/memory buffers and CPU of the client are used to carry out the necessary processing. The server performs only low-level database functionalities (I/Os, buffering and management of concurrency) on the behalf of requesting clients. The set of objects cached at a client is treated as a local dataspace and is stored in the client's short and long-term memory. All future requests for the cached data can now be satisfied at the client itself. In this paper, we support the client framework used in [16, 25, 26] where clients cache the locks for objects as well. Since several clients can cache the same database objects, the server maintains a global lock table to serialize updates to cached data.

There are two kinds of locks, Shared (SL) and Exclusive (EL) [16]. A client transaction can update a cached database object only if the client has an EL on that object. The locking scheme is a variant of strict *two phase locking* (2PL) for distributed environments. Unserialized accesses to the database are prevented by ensuring that no two clients are able to acquire conflicting locks on an object simultaneously. If a client's lock request on an object conflicts with locks presently granted to other clients then the server calls back all such locks. Once these locks have been released, the server grants the lock to the requesting client and sends the object over. We have modified the lock callback scheme in order to allow greater latitude for data sharing. Now, when the server requests a client to give up an EL on a database object, it also specifies the type of lock the requesting client desires. If the other client has requested a SL then the client that holds the EL returns the object to the server but only *downgrades* its own lock to a SL. Now, transactions at both clients can continue to access the object in shared mode.

Each client in the system has its own scheduler to prioritize local transactions. Clients also have their own local lock managers to ensure that concurrent transactions at the client access the data in a serialized manner. Transactions in both the centralized and client-server models are scheduled according to the Earliest Deadline First (ED) policy. In this scheduling policy, the transaction with the earliest deadline is assigned the highest priority. We assume that the system has no knowledge of task execution times so we are unable to use the Least Slack scheduling discipline to assign priorities to transactions [1]. Since the ED policy does not use the estimated processing time of jobs it can possibly schedule jobs that have already missed their deadline. To avoid this inefficiency, we use an additional criterion that tasks that have missed their deadlines are not processed at all. It is in this CS framework that we have employed our load-sharing algorithm in order to improve its real-time processing efficiency. The techniques used in our load-sharing algorithm are described in the following section.

3 Techniques for Improved Real-Time Processing

This section describes the techniques that we have used in our load-sharing algorithm, namely, transaction-shipping [9], transaction decomposition [19], object migration, and object request scheduling.

3.1 Transaction-Shipping

In object-shipping client-server environments different, and possibly overlapping, sets of database objects are cached by clients. At times, clients may not be able to commence transaction execution because other sites have locked solicited objects in a conflicting manner. In these cases, it may be beneficial for a client to ship the transaction to the site that has locked the object(s) in question.

Such transaction-shipping requires up-to-date knowledge of database object locations and load at each client. This knowledge can be maintained at the server since it stores the global lock tables for all database objects and is in frequent contact with all clients. There are two cases when it is feasible to make use of a transaction-shipping mechanism: (i) If a significant percentage of a transaction's required data is already cached at another site, and (ii) If the client where a transaction has been launched is heavily loaded.

Transaction-shipping can offer several advantages in distributed real-time systems, namely: (a) the client can initiate processing of transactions that it is possible to execute locally, (b) a shipped transaction will have at least as much chance of successful completion at that site as at its originating site, and (c) network traffic can be reduced by shipping transactions instead of database objects.

3.2 Transaction Decomposition

Transaction decomposition refers to the disassembly of multiple object requests from a client transaction and the quest to individually fulfill independent object requests. Transaction decomposition consists of three phases: request disassembly, materialization, and answer synthesis. For example, if a transaction needs objects already cached in four different locations then, the requesting client can communicate its data and processing needs to these four clients. This communication could be facilitated by the server which is aware of the locations and the lock status of cached objects. Once the four clients retrieve the necessary objects in parallel (materialization phase), they can forward the results to the requesting client for further processing and final transaction handling (answer synthesis). In order to make transaction decomposition easier to accomplish, an *a priori* classification of transactions based on data access/update patterns may be established.

In general, decomposition may assist in the off-loading of a client site in two major ways: (i) when a client requests objects that have been cached in numerous sites (a situation reminiscent of data fragmentation in distributed databases), and (ii) when a set of client transactions can be considered all together and the client database manager can take advantage of the common object requests that these transactions demonstrate [19, 20]. The advantages of transaction decomposition are similar to those of transaction-shipping with the added benefit that each of the subtasks could be processed in parallel and may take considerably shorter time to process. A disadvantage of such parallel subtask processing is that the failure of any subtask to meet the transaction deadline implies the failure of the entire transaction.

3.3 Object Request Scheduling

In a non real-time environment, the order in which client object requests are served is often First-Come First-Serve. However, in a real-time system, object requests can be prioritized according to the deadlines of the requesting transactions. Object requests by clients can convey the deadline information of the requesting transactions. Requests by transactions with earlier deadlines are satisfied before others. If a client transaction has already missed its deadline then the server can unilaterally decide not to ship the object to that client. In fact, if reasonably accurate estimates of transaction execution times were available, the server could decide whether to satisfy certain object requests at all viz. requests by transactions that are expected to miss their deadlines.

3.4 Object Migration Among Clients

The movement of database objects within a cluster can play a significant role in improving the efficiency of the system. To this end, we group the granting and release of locks requests by multiple clients on the same object [2]. Assuming that a transaction accesses n data objects, the first phase of Standard 2PL will involve n requests from the client to the server and n replies from the server. In the second phase, the client sends n messages to release the locks on the objects and return all modified objects to the server. Therefore, the total number of messages sent is $3n$. In a CS-DBS that allows inter-transaction caching, the client may retain the objects and locks until the server calls them back. If the client does not return some objects voluntarily, the server may have to call each of them back individually. Therefore, the total number of messages required during this interaction could be as high as $4n$. An example interaction using 2PL is shown in Figure 1. which indicates that moving an object from Client A to Client B (through the server) requires 7 messages.

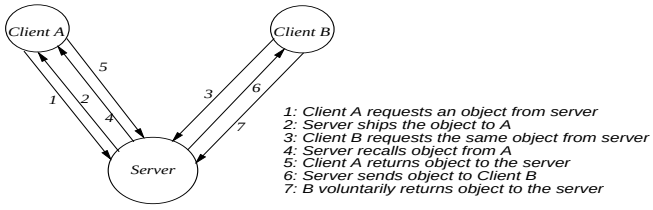

Figure 1. The 2PL Protocol

In the request grouping technique, the object server collects all the lock requests for each database object for a specified time interval (*collection window*) in an ordered list (*forward list*) [2]. At the end of the collection window, the lock is granted to the first transaction in the forward list and the object is shipped to the respective client along with the

forward list. When this transaction commits, the client ships the object to the next client in the forward list. Finally, after the last transaction on the forward list completes, the object is returned to the server. Therefore, in this scheme, the lock release of the previous client is combined with the lock grant of the next client. For n requests on a database object within a collection window, the lock grouping protocol will require only $2n + 1$ messages. An example interaction that requires only 5 messages is shown in Figure 2. In a real-time environment, the forward list can be prioritized according to the deadlines of the requesting transactions. The deadline information is stored in the forward list and is used to ignore transactions that have missed their deadlines. Appropriate information can also be placed in the forward list to indicate parallel read-only access to data.

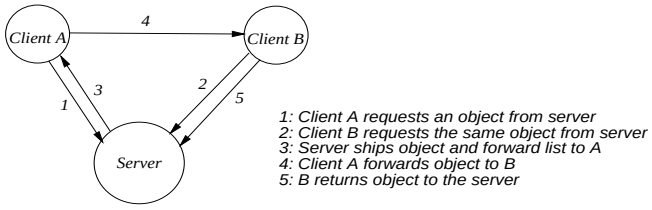

Figure 2. The Lock Grouping Protocol

4 Load Sharing

In this section, we describe a load sharing algorithm that makes use of the techniques described so far. In the context of real-time systems, the primary objective is to minimize the number of transaction that miss their deadlines. A major advantage of CS-DBSs is that the clients and the server are in frequent contact. Therefore, information about the current processing load at clients can be conveyed to the server piggybacked on object requests and releases. This allows the server to maintain up-to-date information about the load on the clients without incurring an additional messaging overhead. There are two reasons to ship a transaction to another site: (i) if a transaction is expected to miss its deadline at the client where it is initiated. Since we do not have accurate transaction execution times, we use the observed transaction times as a heuristic (H_1) to assist in making this decision, and (ii) we have observed that most of the transaction blocking time is spent in waiting for clients that have conflicting locks to release them. Therefore, executing a transaction at a site which has to acquire the least number of conflicting locks is the most efficient available option (H_2). In the description of both heuristics, consider a transaction T that has been initiated at Client A.

H_1 : If Client A has n transactions before transaction T in its priority queue then transaction T has a *reasonable chance* of successfully completing at Client A if $(CurrentTime + n * ATL_A) \leq T_{deadline}$, where ATL_A is the average execution time for all completed transactions at Client A.

H_2 : The time spent by a transaction waiting for its requisite data to become available is a key factor responsible for the failure of real-time transactions [13]. Therefore, for transaction T, we consider Client A to be a better processing site than Client B if T has to wait for fewer conflicting locks to be released if it is processed at A.

Transaction decomposition can be incorporated into the load sharing algorithm in an elegant manner. If a transaction is decomposable then it can be executed as a set of independent tasks. Each of these tasks is processed as a separate operation by the load sharing algorithm. The object migration scheme works orthogonally to the load sharing algorithm. However, when a client requests an object's location, the server refers to the object's forward list and reports the last client in the list as the object's location.

In the description of the load-sharing protocol, we use granting of locks and the shipping of objects interchangeably. The load-sharing algorithm works as follows:

- Transaction T is initiated at a client.
- **IF** T can be accommodated in the local processing queue with a *reasonable chance* (H_1) of meeting its deadline **THEN**
  - the client looks in its local cache (memory/disk) for the objects/locks requested by the transaction. For all objects that are not available locally the client sends requests to the server.
  - **IF** the server can grant all the objects/locks requested by transaction T **THEN** these objects are locked appropriately on behalf of that client and shipped to it. The transaction is executed by the client locally. **ELSE**
    - the server does not ship any objects over but instead sends the locations for the requested objects that have been locked in conflicting modes by other clients.
    - **IF** another client is in a better position to complete this transaction (H_2) **THEN** the transaction is shipped to that client **ELSE** the client sends a message to the server indicating that the transaction will be processed locally and asks that the requisite objects be shipped over as soon as possible.

  ELSE
  - the client queries the server for information about the location of the required objects and the processing loads on the other clients.
  - Once this information is received, the most suitable client (H_2) is picked and the transaction is shipped to that client. Requests for objects required by that transaction are sent to the server on behalf of that client.
- **IF** necessary, the results of executing the transaction are communicated to the originating client.

From the above pseudocode, it can be seen that the final decision about transaction-shipping is made by the clients. Off-loading the load sharing effort from the server is important because in situations of high load, the server may lag in serving object requests, maintaining load tables and forward lists.

5 Prototype Experiments

In order to evaluate the effectiveness of the load-sharing algorithm we have developed prototypes of the centralized real-time database (CE-RTDBS), client-server real-time database system (CS-RTDBS), and the load-sharing CS-RTDBS (LS-CS-RTDBS) using Solaris socket and thread libraries. In this section, we describe the experimental setup and discuss the results of our prototype experiments. The main questions that our investigation attempts to answer are: (i) how does the performance of the client-server database model compare with that of the centralized model in a real-time environment?, and (ii) do the load sharing and object migration techniques used in the LS-CS-RTDBS allow a better level of performance than the CS-RTDBS?

5.1 Methodology

The test environment for our experiments was a system consisting of 5 Sun ULTRAs residing on a 10-Mbps Ethernet LAN. The database server executed by itself on one machine and the clients were uniformly divided on the remaining workstations. The Paged-File (PF) layer, from the MiniRel system [23], is used to create and manage a database containing 10,000 objects. The PF layer implements a file page buffer manager, and allows the storage and retrieval of uniquely numbered fixed-sized pages from its memory buffers and disk file. For convenience, we assume that the size of the database objects and that of the PF layer pages used to store the database are the same, i.e., 2 KB. Table 1 states the values of the parameters used in our prototype experiments.

Parameter	*Value*
Database Size	10,000 objects
Centralized RTDBS Server Main Memory Capacity	5,000 objects
CS-RTDBS Server Main Memory Capacity	1,000 objects
Client Disk Cache Size	500 objects
Client Memory Cache Size	500 objects
Average Transaction Inter-Arrival Time (Poisson Distribution)	10 sec.
Average Transaction Length (Exponential Distribution)	10 sec.
Average Transaction Deadline (Exponential Distribution)	20 sec.
Percentage of Updates	1%, 5%, 20%
Average Number of Objects Accessed by Each Transaction	10

Table 1. Experimental Parameters

The prototypes for all three systems have been developed in C. Communication between the clients and the server is done using TCP sockets. In all implementations, the server is designed as a concurrent connection-oriented program. Once a connection is created between a client and the server, it is maintained for the duration of the experiment. This is done so that the relatively high overhead of establishing a socket connection between the clients and the server is incurred only once. The server executes one thread per client which handles all future interaction with it. In the LS-CS-RTDBS, we ensure that client-to-client communication is not always routed via the server by using a specialized directory server. The only function of this directory server is to forward data or messages to their intended recipients.

The transaction stream at each client is a mixture of updates and queries. Ten percent of all submitted transactions were decomposable. A transaction is constructed by using calls to PF layer functions to load and update database objects from the UNIX disk file. Objects (pages) are read into the buffer space allocated to the transaction either from the disk or PF memory buffer. Updated objects are marked as dirty and are automatically written back to the disk file by the PF buffer manager when the page is replaced in the buffer. The "processing" performed by each transaction is the calculation of products of random numbers until the prescribed transaction execution time has elapsed. The CE-RTDBS server has been designed to be able to process as many as one hundred transactions simultaneously depending upon the availability of memory buffer space and access to database objects. This is done by executing each transaction as a separate thread.

Lock managers are used to ensure that conflicting accesses to the object database are not allowed. In the centralized system, the lock manager ensures that transactions cannot obtain conflicting locks on database objects. In the CS-RTDBS, the global lock manager arbitrates between lock requests from clients. **Wait-for** graphs are used to detect deadlocks. When an object request is received by the server, it is added to the request queue only if it does not cause a deadlock cycle in the wait-for graph. Since each program is multi-threaded, we synchronize accesses to the database, lock table, and other shared variables by using of Solaris mutex primitives. In our database access pattern, which we call *Localized-RW*, 75% of each client's accesses were made to a particular portion of the database according to the Uniform distribution while the other 25% of the accesses were to the remainder of the database according to the Zipf distribution [8, 12].

5.2 Results

We have evaluated the CE-RTDBS, CS-RTDBS, and the LS-CS-RTDBS with the *Localized-RW* database access pattern and a varying percentage of updates. The percentage of transactions that completed within their deadlines in the three configurations for 1% updates is shown in Figure 3. We first compare the performance of the CE-RTDBS and the CS-RTDBS. For a small number of clients, the centralized

system performs better than the CS-RTDBS. The faster processing ability of the centralized system and the low contention for database objects allow the CE-RTDBS to exhibit a very high degree of concurrent transaction execution. However, as the number of clients increases, the performance of the CE-RTDBS system deteriorates rapidly. For more than 40 clients, the centralized system does not perform as well as the CS-RTDBS. The locality in every client's data accesses and the low percentage of updates means that a high percentage of client data object requests can be satisfied locally (Table 2). The server is also able to satisfy most object requests within a very short time (Table 3). An important factor that causes client transactions (in the CS-RTDBS) to fail is the delay in obtaining objects/locks that are held by other clients. The LS-CS-RTDBS is able to reduce this blocking delay in many cases by using the load-sharing algorithm and the object migration policies, thus, resulting in a significantly higher percentage of successful transactions. The number of requests that have been pushed along with the objects in forward lists is significant, and a corresponding reduction can be seen in the number of messages passed for object recalls and returns (Table 4).

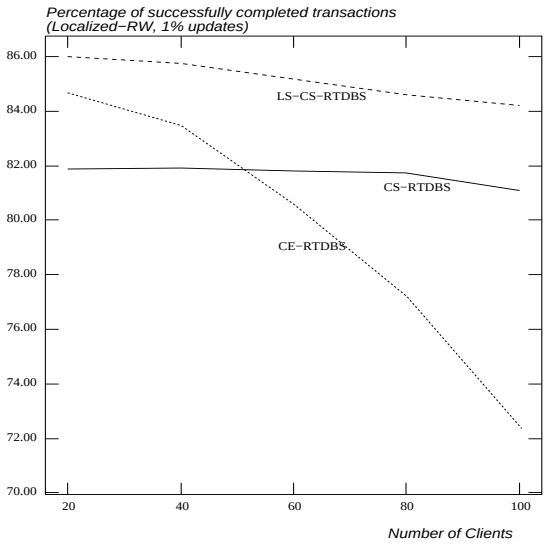

Figure 3. Percentage of Transactions Completed Within Their Deadlines (1% updates)

Number of Clients	*CS-RTDBS*			*LS-CS-RTDBS*		
	1%	*5%*	*20%*	*1%*	*5%*	*20%*
20	87.08	84.63	79.74	89.63	87.11	84.31
60	85.54	78.18	74.64	88.63	84.11	81.71
100	82.63	75.52	62.29	86.55	82.21	66.90

Table 2. Average Cache Hit Rates in the CS-RTDBS and LS-CS-RTDBS

Number of Clients	*CS-RTDBS*		*LS-CS-RTDBS*	
	Shared Locks	*Exclusive Locks*	*Shared Locks*	*Exclusive Locks*
20	0.024	0.487	0.027	0.433
60	0.063	0.538	0.052	0.509
100	0.069	0.850	0.058	0.628

Table 3. Average Object Response Times (in seconds) for 1% updates

	CS-RTDBS	LS-CS-RTDBS
Object Request Messages (client to server)	109,911	104,314
Objects Sent (server to client)	108,273	94,596
Object Requests Satisfied Using Forward Lists (client to client)	-	9,718
Objects Recall Messages (server to client)	45,130	41,071
Objects Returned (client to server)	45,136	41,020

Table 4. Number of Messages Passed in the CS-RTDBSs (100 Clients, 1% updates)

The performance of the three models for 5% updates is shown in Figure 4. As the curves show, the increased contention resulting from a higher percentage of updates affects the performance of all three systems adversely. In the centralized system, the effect of this increased contention is in the degree of transaction concurrency that can be achieved but, the CS-RTDBS is affected the most by the increased locking conflicts. Lock conflicts between clients cause data objects to be returned to the server and re-fetched much more frequently. This results in long blocking periods for client transactions. Particularly, when an object in a client's frequently accessed region has to be returned to the server, the delay in re-fetching it can affect many transactions. The LS-CS-RTDBS can avoid such delays in a large number of cases. Rather than wait for the data be sent to the transaction's client, the transaction itself is shipped to the client which has presently locked the data. This flexibility in the execution of transactions allows the LS-CS-RTDBS to outperform the CS-RTDBS and centralized system (once the number of clients is greater than 20).

Figure 5 shows the performance of the three models for 20% updates. The overall performance trends match those seen in the two earlier scenarios. The CS-RTDBS and the LS-CS-RTDBS demonstrate a very small deterioration in performance as the load increases whereas the CE-RTDBS's efficiency degrades much more rapidly. Although the CE-RTDBS does better than the LS-CS-RTDBS initially, its rapid performance degradation makes it worse when the number of clients becomes larger than 40. The CS-RTDBS is affected severely by the very high number of update transactions. A large percentage of object requests cannot be granted by the server immediately. Similarly, an increased number of objects have to be given up and re-fetched from

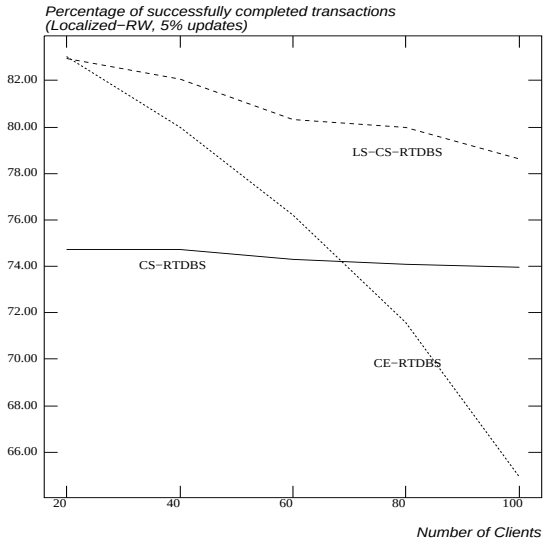

Figure 4. Percentage of Transactions Completed Within Their Deadlines (5% updates)

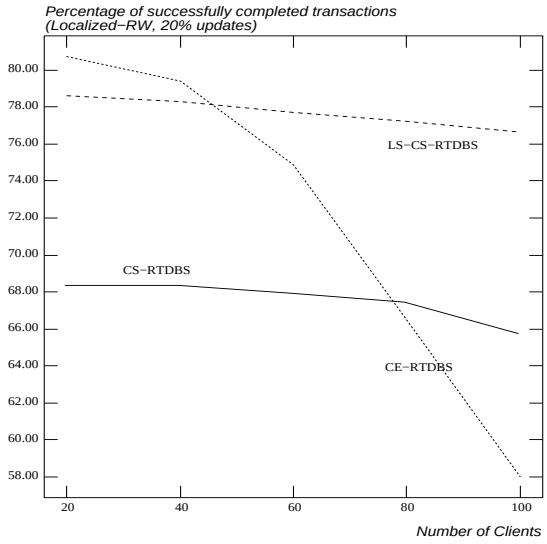

Figure 5. Percentage of Transactions Completed Within Their Deadlines (20% updates)

the server. These delays are instrumental in causing the CS-RTDBS to perform worse than the CE-RTDBS (for up to 80 clients). It is only the locality in each client's data accesses that causes the CS-RTDBS to do better than the CE-RTDBS when the number of clients is 100. The LS-CS-RTDBS completes almost 10% more transactions successfully than the CS-RTDBS. In a real-time environment, this is a very sizeable improvement.

6 Related Work

Many techniques have been suggested for load sharing in shared memory multi-processor machines [5, 7, 17, 18]. Since the data is visible to all processors, the load sharing exclusively concerns CPU loads. Real-time systems have also exploited available information about the submitted tasks so as to guarantee their completion. This information includes the task's deadline, its expected processing time and I/O requirements, and the priority assigned to it. Scheduling techniques have been proposed to make use of this information in order to maximize the number of transactions that complete within their deadlines [1, 10]. Optimization of assignment of tasks in distributed soft real-time environments has been studied in [14]. The Imprecise Computation Server [11], advocates the use of approximate but usable results so as to maintain an acceptable level of system performance.

Techniques for load sharing have also been proposed in distributed database systems. The application of an economic model to adaptive query processing and dynamic load-balancing in a non real-time context was proposed in [22]. Load-balancing heuristics for use in a distributed real-time system are described in [21]. An algorithm that migrates tasks from one node of a distributed real-time system to another if the latter offers a higher probability of successful task completion is described in [4].

The primary distinction between the works described above and our load-sharing algorithm is that we examine the problem of scheduling real-time tasks in a CS-DBS environment. In this setting, the availability of database objects/locks is essential for a transaction to proceed. Our load-sharing algorithm also uses information about the location of data and available client resources in order to minimize the percentage of missed deadlines.

7 Conclusions

The use of client-server systems for database computing is pervasive. In this paper, we have utilized a set of techniques to enhance the real-time processing capabilities of a client-server database. These techniques are: transaction-shipping and decomposition as well as object migration scheduling. Exploiting the above techniques, we have designed a load-sharing algorithm that transfers processing tasks to clients that can provide immediate access to the required data and may be less loaded. To evaluate our load sharing algorithm, we developed operational prototypes of the centralized (CE-RTDBS), basic client-server (CS-RTDBS), and load-sharing client-server (LS-CS-RTDBS) real-time databases. From the experimental results, our key conclusions are: (i) The client-server systems provide a very consistent level of performance as compared to the centralized real-time database system. As the number of clients

increase, the performance of the CE-RTDBS degrades very rapidly while those of the CS-RTDBS and LS-CS-RTDBS show very little deterioration. (ii) The use of load-sharing and object migration policies in the LS-CS-RTDBS significantly improve its efficiency over that of the CS-RTDBS. In fact, the LS-CS-RTDBS completes 10% more transactions than the CS-RTDBS under the Localized-RW access pattern with 20% updates. This is a considerable improvement in a real-time environment. (iii) An increase in the percentage of updates affects the client-server systems more than the centralized one. However, the LS-CS-RTBDS is able to avoid unnecessary data movement and offer a much better level of performance than the CS-RTDBS.

This paper evaluates the processing efficiency of CS-RTDBSs with pessimistic data locking. In the future, we intend to study the use of optimistic concurrency control and speculative transaction processing techniques to evaluate their impact on real-time system performance [24, 3].

Acknowledgements: We would like to thank Krithi Ramamritham and Joel Wein for their helpful suggestions in the preparation of this paper as well as the referees for their comments.

References

[1] R. Abbott and H. Garcia-Molina. Scheduling Real-Time Transactions: A Performance Evaluation. *ACM Transactions on Database Systems*, 17(3), 1992.

[2] S. Banerjee and P. K. Chrysanthis. Data Sharing and Recovery in Gigabit-Networked Databases. In *Proceedings of the Fourth International Conference on Computer Communications and Networks*, September 1995.

[3] A. Bestavros and S. Braoudakis. Timeliness via Speculation for Real–Time Databases. In *Proceedings of the IEEE Real–Time Systems Symposium*, December 1994.

[4] A. Bestavros and D. Spartiotis. Probabilistic Job Scheduling for Distributed Real-Time Applications. In *Proceedings of the 1st IEEE Workshop on Real-Time Applications*, May 1993.

[5] A. Burchard, J. Liebeherr, Y. Oh, and S. Son. Assigning Real–Time Tasks to Homogeneous Multiprocessor Systems. *IEEE Transactions on Computers*, 44(12), December 1995.

[6] M. Carey, M. Franklin, M. Livny, and E. Shekita. Data Caching Tradeoffs in Client-Server DBMS Architecture. In *Proceedings of the ACM SIGMOD Conference*, May 1991.

[7] Y. Chow and W. Kohler. Models for Dynamic Load Balancing in a Heterogenous Multiple Processor System. *IEEE Transactions on Computers*, 28(5), May 1979.

[8] A. Dan, P. S. Yu, and J. Chung. Characterization of Database Access Skew in a Transaction Processing Environment. In *Proceedings of the SIGMETRICS and PERFORMANCE '92 International Conference on Measurement and Modeling of Computer Systems*, June 1992.

[9] A. Delis and N. Roussopoulos. Performance Comparison of Three Modern DBMS Architectures. *IEEE Transactions on Software Engineering*, 19(2), February 1993.

[10] W. Feng and J.-S. Liu. Algorithms for Scheduling Real-Time Tasks with Input Error and End-To-End Deadlines. *IEEE Transactions on Software Engineering*, 23(2), 1997.

[11] D. Hull, W. Feng, and J. Liu. Enhancing the Performance and Dependability of Real-Time Systems. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium*, pages 174–182, April 1995.

[12] Y. Ioannidis. Universality of Serial Histograms. In *Proceedings of the 19th VLDB Conference*, Dublin, Ireland, 1993.

[13] V. Kanitkar and A. Delis. A Case for Real-Time Client-Server Databases. In *Proceedings of the 2nd International Workshop on Real-Time Databases*, September 1997.

[14] B. Kao and H. Garcia-Molina. Subtask Deadline Assignment for Complex Distributed Soft Real-Time Tasks. In *Proceedings of the 14th IEEE ICDCS*, June 1994.

[15] A. Keller and J. Basu. A Predicate-Based Caching Scheme for Client-Server Database Architectures. *The VLDB Journal*, 5(1), 1996.

[16] E. Panagos, A. Biliris, H. V. Jagadish, and R. Rastogi. Client-Based Logging for High Performance Distributed Architectures. In *Proceedings of the 12th International Conference on Data Engineering*, Feb-March 1996.

[17] K. Ramamritham, J. Stankovic, and P. Shiah. Efficient Scheduling Algorithms for Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990.

[18] L. Rudolph, M. Slivkin-Allalouf, and E. Upfal. A Simple Load Balancing Scheme for Task Allocation in Parallel Machines. In *ACM Symposium on Parallel Algorithms and Architectures*, 1991.

[19] T. Sellis. Multiple-query optimization. *ACM Transactions on Database Systems*, 13(1), March 1988.

[20] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez. Transaction Chopping: Algorithms and Performance Studies. *ACM Transactions on Database Systems*, 20(3), September 1995.

[21] J. Stankovic, K. Ramamritham, and S. Cheng. Evaluation of a flexible task scheduling algorithm for distributed hard real-time systems. *IEEE Transactions on Computers*, 34(12), December 1985.

[22] M. Stonebraker, P. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin, and A. Yu. Mariposa: A Wide-Area Distributed Database System. *The VLDB Journal*, January 1996.

[23] The MiniRel Development Team. The MiniRel Relational DBMS. University of Wisconsin, Madison, 1989.

[24] A. Thomasian. Distributed optimistic concurrency control methods for high-performance transaction processing. *Transactions on Knowledge and Data Engineering*, 10(1), January/February 1998.

[25] Y. Wang and L. Rowe. Cache Consistency and Concurrency Control in a Client/Server DBMS Architecture. In *Proceedings of the ACM SIGMOD Conference*, May 1991.

[26] K. Wilkinson and M. Neimat. Maintaining Consistency of Client-Cached Data. In *Proceedings of the 16th VLDB Conference*, August 1990.