An Efficient Memory Management Algorithm for Client–Server Database Management Systems

YASEMIN TURKAN AND ÖZGÜR ULUSOY*

*Department of Computer Engineering and Information Science, Bilkent University, Bilkent,
Ankara 06533, Turkey
Email: oulusoy@bilkent.edu.tr*

Due to the recent improvements in the price/performance characteristics of workstations and the networking capabilities, the client–server system architecture has become a target for database systems. A client–server database management system (DBMS) provides the management of a database that resides on a client–server system. Data-access requests of the clients are handled by the database servers. The whole database is stored on the disks that can be accessed by only the servers, and in order to reduce disk accesses, copies of database items can be cached in the global memory which comprises the memories of all the computers connected to the system. Designing efficient global memory management algorithms helps the transactions experience less disk input/output (I/O) during their execution. In this paper, we propose a global memory management algorithm for client–server DBMSs which aims to reduce disk I/O by increasing the portion of the database available in global memory. Performance of the algorithm is examined by a comparison with some previously proposed algorithms, using a simulation model designed for studying various performance issues in client–server DBMSs. Conditions for which the new algorithm provides significant improvements in the overall throughput of the system are identified.

Received July 26, 1995; revised June 24, 1996

1. INTRODUCTION

A client–server system is a distributed computing system that consists of one or more server processes, and one or more client processes that request services from the servers for the end-users. The machines on which the server and the client processes execute are interconnected by a network, and all the communication between these processes is provided by exchanging messages over the network. In a client–server database management system (DBMS), the whole database is stored on the disks that can be accessed by only 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 DBMSs (see [1, 2, 3, 4]), the ‘callback locking’ algorithm is the most popular one [5]. 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.

An important design issue in client–server DBMSs is the efficient management of the global memory which

consists of the memories of the servers and the clients. Designing efficient global memory management algorithms helps the transactions experience less disk input/output (I/O) during their execution. In a recent study, some techniques were proposed by Franklin *et al.* to provide efficient management of global memory in client–server DBMSs [6, 7]. These techniques are discussed in Section 3.2. Performance of global memory management algorithms employing the proposed techniques was studied using a simulation model, and it was shown that the algorithms can provide significant performance gains over an algorithm that does not involve any of these techniques.

In this paper, we present a global memory management algorithm that employs a new technique intended to provide further improvement in performance by enabling the system to keep a larger portion of the database in global memory, and thus reducing the number of disk I/Os. Considering a page to be the unit of interaction between the server and the clients, in the new technique (called *Forwarding-Dropped-Pages*), when a client has to drop a page from its memory which is the only copy in the global memory, the page is sent to the server if its memory has an available space. However, if the server’s memory is full, the page is sent to a remote client that has a free memory space. Therefore, client memories can also be exploited to keep more distinct pages in global memory.

Performance of the memory management algorithm that employs the new technique is studied using a detailed

simulation model of a client-server DBMS. The performance results are compared with the results obtained with a base algorithm that does not involve any global memory management techniques, and an algorithm that combines the techniques proposed by Franklin. The comparative results demonstrate that our algorithm can provide considerable performance improvements especially when the underlying network is fast, and the size of the server and client memories is not large enough to hold a significant portion of the database.

The next section summarizes recent work on global memory management and cache consistency in client-server DBMSs. In Section 3, the global memory management algorithms studied, including the proposed algorithm, are described. Section 4 presents our client-server DBMS simulation model. Section 5 provides a set of simulation experiments together with our findings. The last section summarizes the conclusions of our work.

2. RELATED WORK

Global memory management and cache consistency problems in client-server DBMSs have been addressed by a number of recent studies. Wang and Rowe distinguished between caching within a transaction (intra-transaction) and between transactions (inter-transaction) [3]. They evaluated the performance of five inter-transaction algorithms under different workloads and system configurations. Pu *et al.* provided an approach to offloading data access requests from overloaded data servers to clients that are idle or less busy [8]. Such clients are called *mutual servers*. They presented several memory management algorithms for efficient utilization of the global memory that comprises data servers' memories and mutual-servers' memories. Wilkinson and Neimat proposed two cache-consistency algorithms as extension to two-phase locking [4]. One of the algorithms, called *cache locking*, uses three additional lock modes that must be supported by the server. The second algorithm, called *notify locks*, requires that the server should send the owners of notify locks a notification message whenever an operation on a locked item is to be performed. Through a simulation work, it was shown that both algorithms perform better than two-phase locking without caching for various types of applications. Carey *et al.* examined the performance trade-offs that are raised by caching data at the clients of a client-server DBMS [1]. Later, that study was extended in several ways [2]. The first extension was examination of heuristics for algorithms that decide dynamically between propagating changes or invalidating remote copies of data items in order to maintain cache consistency. Another extension was an evaluation of different versions of callback locking. The authors conducted performance experiments in the light of more recent trends in processor and network speeds. Franklin *et al.* presented several techniques for efficiently exploiting the global memory in client-server DBMSs [6]. In a more recent paper,

Franklin *et al.* proposed the use of client disks for caching database pages in an extended cache architecture [9].

Page servers are often criticized as being restrictive in terms of concurrency, as existing systems use pages as the locking unit as well. In [10], Carey *et al.* presented how to support object-level locking in a page server context. They described several approaches, including an adaptive granularity approach that uses page-level locking for most pages but switches to object-level locking when finer-grained sharing is demanded.

Acharya *et al.* presented a new technique called *broadcast disks* that proposes the use of repetitive broadcast as a way of augmenting the memory hierarchy of clients in an asymmetric communication environment [11]. This technique structures the broadcast in a way that provides improved performance for non-uniformly accessed data. In that work, it was assumed that clients access the broadcast in a demand-driven manner. In a more recent work, it was shown by the authors that demand-driven access does not fully exploit the dissemination-based nature of the broadcast, and client *prefetching* can be more effective with a broadcast disk [12]. Since pages continually flow past the clients, prefetching can be performed without placing additional load on shared resources.

Amsaleg *et al.* described an efficient server-based algorithm for garbage collection in a client-server DBMS [13]. Unlike previous algorithms, the proposed algorithm does not hold any locks on data and does not require callbacks to clients.

In [14], Franklin *et al.* investigated the performance trade-offs between two query execution strategies: data-shipping and query-shipping. They also described and evaluated a hybrid-shipping policy, which can execute queries at clients, servers or any combination of the two.

3. GLOBAL MEMORY MANAGEMENT ALGORITHMS

Global memory management in a client-server DBMS aims to minimize the cost of data access. In our client-server DBMS model, the global memory management algorithms described in the following subsections have been built on the Callback-All (CB-A) cache consistency algorithm [2].

In the CB-A algorithm, when a client wants to update a page, it needs to obtain a write lock from the server. If some other clients have conflicting locks on the same page, the server sends a callback message to those clients for the conflicting locks. When all the conflicting locks are successfully called back, the write lock is granted to the requesting client. However, the write lock is not released at the end of the transaction that has updated the page.

If a read lock request for a page arrives at the server and a write lock for the same page is currently being held

at another client, the server sends a downgrade request to that client. On receiving a downgrade request, the client downgrades its write lock (if it is not currently being used) to a read lock.

3.1. The base algorithm

The base algorithm works as follows. When a client needs to access a data page, it 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 is found in the memory, the server sends it to the client. Otherwise, the server has to transfer the page from its disk to the memory before sending it to the client. The base algorithm enables us to measure the performance benefits of the following global memory management algorithms.

3.2. Franklin's algorithm

The algorithm proposed by Franklin combines three global memory management techniques which are called *Forwarding*, *Hate-Hints* and *Sending-Dropped-Pages* [7].

The basic aim of the *Forwarding* technique is to minimize disk accesses by extending the global memory. When a client requests a page from the server, the server first searches the page in its cache. If the page does not exist in the server's cache, the page request message is forwarded to a client that has a local copy of the page. Upon receipt of a forwarded request, the remote client sends the copy of the page to the requesting client. Therefore, the forwarded request message prevents the disk I/O at the server.

The *Hate-Hints* technique is used to keep a larger portion of the database available in global memory. When the server sends a page to a client, the copy of the page in the server's memory is marked as hated. The page will probably be replaced soon, when an empty space in the server's memory is needed for a new page. Therefore, this technique reduces the replication of pages in the global memory.

The *Sending-Dropped-Pages* technique is also used to increase the portion of the database in the global memory. With this technique, the server's memory is exploited to prevent a page to be completely dropped out of global memory. If a client is going to drop a page from its memory to place a new page, it informs the server by piggybacking this information to the page request message. If the page is the only copy in the global memory, the server tells the client to send the dropped page, by piggybacking this information to the requested page. Otherwise, the page is just dropped as it has at least one more copy that resides in the global memory.

3.3. Our algorithm

The global memory algorithm we propose intends to extend and improve Franklin's algorithm. Our algorithm

makes use of an additional memory management technique which we call *Forwarding-Dropped-Pages*. In this technique, when a client is going to drop a page that is the only copy in the global memory, the server tries to keep it in its local memory. However, if its memory is full or does not contain any hated page, the server tries to find a remote client that has a free memory space to receive the dropped page. Therefore, client memories can also be exploited to keep more distinct pages in global memory.

The introduced technique enables the system to keep a larger portion of the database in global memory. It is expected to improve the performance of client-server DBMSs, especially when the cost of disk I/O is high compared with the cost of transmitting and processing messages. In the remainder of this section, we describe a global memory management algorithm that makes use of this technique in addition to those introduced by Franklin.

When an executing transaction needs to access a page, it must first obtain a read or a write lock according to the type of the access. If the required lock is currently being held by the client executing the transaction, then the transaction continues its execution. Otherwise, the client requests the lock from the server. After obtaining the lock, the transaction can access the page if it is currently being stored in the local memory. If not, the client has to request the page from the server. Before sending the page request, the client checks its memory to see if it has to drop a page in order to receive a new one. If so, it piggybacks this information to the page request message that is going to be sent to the server.

On receiving a page request message, the server searches its memory. If the page is there, the server directly sends the page to the client and marks that page as hated. If a copy of the page does not exist in the server's memory, but resides in the global memory (in one of the clients' caches), the server forwards the page request message to a client that has a local copy of the page. In the case that no copy of the page resides in the global memory, the server reads the page from the disk into its memory and sends it to the client.

If the page request message has a piggybacked information stating that the client has to drop a page out of its memory in order to receive a new one, the server checks whether that page is the only copy in the global memory. If so, the page should not be dropped from the global memory. In Franklin's algorithm, the server requests the client to send the page directly to itself. If the server's memory is full, the Least Recently Used (LRU) page is replaced with the dropped page. However, in our algorithm, the server tells the client to send the page to itself if its memory has an empty space or a hated page to receive the dropped page. Otherwise, the server searches for a client that can take that page. If there exists any remote client that can store the page in its cache, the server requests the client to send the dropped page to that remote client. A client which has in its

memory either an empty space or a copy of a multiple copy page can accept the dropped page. If there exists more than one such clients, one of them is chosen randomly. If no client has a place to store the dropped page in its cache, the server accepts that page into its memory replacing the page with the LRU value.

4. CLIENT-SERVER DBMS SIMULATION MODEL

A simulation model has been developed to study various performance issues in client-server DBMSs [15]. The simulation model is based on the client-server architecture described in [7]. The simulated client-server system comprises a single server and a number of clients that are all interconnected through a local network. The client-server DBMS 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. The server's workload is generated by the requests coming from the clients. The physical structure of the client-server DBMS model is shown in Figure 1.

Each client in the system contains a transaction generator, a client manager, a memory manager and a resource manager. The transaction generator is responsible for generating the transaction workload for each client. Data pages to be accessed by each transaction are also determined by the transaction generator. The client manager processes the page access requests of the transactions. If the client does not have the appropriate lock on the required page, the client manager requests that lock from the server. While waiting for a lock from the

server, an abort message can arrive for the transaction due to either a deadlock or invalidation of the required page. In that case, the client manager aborts and then restarts the transaction. 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 client manager is also responsible for responding to the messages received from the server and the other clients. The buffer manager is responsible for the management of the client's memory. It makes use of the LRU algorithm for the replacement of pages in the memory. Finally, the resource manager provides CPU service.

The server's components are a server manager, a scheduler, a memory manager and a resource manager. The server 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 at the server keeps track of the information where the copies of each data page reside in the system. It processes the lock requests of the clients in a First-Come, First-Served (FCFS) basis and responds to those requests on the basis of the CB-A locking protocol described in Section 3. After granting the appropriate lock to a client, the server sends the page to the client. 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.

A network manager is also included in the model to simulate a local area network connecting the clients and the server. The network is modelled as a FCFS server with a fixed service rate.

For the detection of deadlocks, a wait-for graph is

FIGURE 1 Client-server DBMS structure.

maintained at the server. A deadlock is recovered from by aborting one of the transactions in the deadlock cycle. An aborted transaction is immediately restarted at its originating client and it accesses the same data pages as before.

The parameters described in Table 1 were used to specify the configuration and workload of the client-server DBMS. The parameters were adapted from the model used in Franklin's experiments [7]. CPU queues at the clients and the server, and the I/O queue at the server, are organized on a FCFS basis. 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, lock-request, lock-grant messages, for example, 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, for example. When a transaction execution is completed at a client, a new transaction is generated after a think time which is specified by parameter *ThinkTime*. 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, and this page is updated with a probability of *HotWriteProb*. If the page

is chosen from the cold region, it will be updated with a probability of *ColdWriteProb*.

5. SIMULATION EXPERIMENTS

The simulation program was written in CSIM [16], which is a process-oriented simulation language based on the C programming language. Each run of the following experiments was continued until 10,000 transactions were processed in the system. The 'independent replication' method was used to validate the results by running each configuration 10 times with different random number seeds and using the averages of the replica means as final estimates. Ninety percent confidence intervals were obtained for the results with the assumption of independent observations [17]. The width of the confidence interval of each statistical data point is < 3% of the point estimate. In displayed graphs, only the mean values of the performance results are plotted.

Simulation experiments were driven by the parameter values presented in Table 1. The parameter values were chosen so as to be comparable with the related simulation studies such as [7]. *NetworkBandwidth* values specified (i.e. 8 Mbits/s and 80 Mbits/s) roughly correspond to Ethernet and FDDI networks, respectively [2]. All the clients are assumed identical and operating under the same parameter values.

The basic performance metric used is *throughput*; i.e. the number of committed transactions per second. The other important performance metrics that helped us to

TABLE 1. Client-server DBMS model parameters and default values.

Configuration parameters		
<i>DatabaseSize</i>	Size of the database in pages	1250 pages
<i>PageSize</i>	Page size in bytes	4096 bytes
<i>ServerMemSize</i>	Number of pages that can be held in the server's memory	10–100% of <i>DatabaseSize</i>
<i>ServerMIPS</i>	Instruction rate of CPU at the server	100 MIPS
<i>ClientMemSize</i>	Number of pages that can be cached in a client's memory	5–50% of <i>DatabaseSize</i>
<i>ClientMIPS</i>	Instruction rate of CPU at each client	50 MIPS
<i>MinDiskTime</i>	Minimum disk access time	10 ms
<i>MaxDiskTime</i>	Maximum disk access time	30 ms
<i>NetworkBandwidth</i>	Network Bandwidth	8 or 80 Mbits/s
<i>FixedMsgInst</i>	Fixed number of instructions to process a message	20,000 instructions
<i>PerByteMsgInst</i>	Additional number of instructions per message byte	10,000 instructions/4 kbyte
<i>ControlMsgSize</i>	Control message size in bytes	256 byte
<i>DeadlockInterval</i>	Deadlock detection frequency	1 s
<i>DiskOverheadInst</i>	CPU overhead for performing disk I/O	5000 instructions
<i>SystemOverhead</i>	Number of instructions for certain system operations	300 instructions
Workload parameters		
<i>NoOfClients</i>	Number of clients	1 to 25
<i>ThinkTime</i>	Think time between client transactions	0
<i>TransactionSize</i>	Number of pages accessed per transaction	20 pages
<i>ReadPageInst</i>	Number of CPU instructions per page read	30,000 instructions
<i>WritePageInst</i>	Number of CPU instructions per page write	60,000 instructions
<i>HotBounds</i>	Page bounds in the hot region	for client n , p to $p + 49$ where, $p = 50(n-1) + 1$
<i>ColdBounds</i>	Page bounds in the cold region	Rest of the database
<i>HotAccessProb</i>	Probability of access to a page in the hot region	0.8
<i>HotWriteProb</i>	Probability of writing to a page in the hot region	0.2
<i>ColdWriteProb</i>	Probability of writing to a page in the cold region	0.2

understand the behaviour of the algorithms are the portion of the database available in global memory, average number of disk I/Os per committed transaction, average number of messages transmitted per committed transaction, and average message volume per committed transaction.

The workload model used in our experiments simulates an environment in which there exists some amount of data contention among the clients and locality of data accesses at each client. 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. Experiments with some other types of workloads concerning different levels of locality and data contention were also conducted. The results of those experiments are briefly discussed at the end of this section.

Before discussing the throughput results, we first present the results obtained in terms of the other performance metrics. Figure 2 shows the portion of the database available in global memory as a function of the number of clients. These results have been obtained by setting *ServerMemSize* to the 50% of the database size, *ClientMemSize* to the 5% of the database size and *NetworkBandwidth* to 8 Mbits/s. The highest portion of the database that can be available in the global memory is shown in the graph with dotted lines. The values obtained with the memory management algorithms deviate from the ideal condition because of the replicated

data that reside in the global memory. An increase in the number of clients leads to an increased global memory size, and thus the amount of the database that can be kept in the global memory increases with all three algorithms. However, since the level of data replication obtained with each memory management algorithm is different, the values produced with the algorithms in terms of the portion of the database available in global memory are also different. The values obtained with the base algorithm are much smaller than those obtained with the other two algorithms. This obvious result is due to the fact that none of the memory management techniques employed by the other two algorithms to reduce data replication is used by the base algorithm.

Franklin's and our algorithms are very close to the ideal case and they approach the 100% ratio as the number of the clients is increased beyond 15 clients. With Franklin's algorithm, due to the *Hate-Hints* technique, when a page is sent to a client, the server marks the page as LRU. This means that it is the first page to be dropped out of the server's memory when the memory becomes full. As a result, the replicated copy will no longer be in the global memory. Another technique used in this algorithm to increase the chance of storing more 'distinct' pages in global memory is *Sending-Dropped-Pages*. This technique ensures that when a client has to drop a page which is the only copy in the global memory, it is stored in the server's memory. Our new algorithm provides slightly better results than Franklin's algorithm. The basic reason for enabling the system to store a larger fraction of the database in memory is the use of the *Forwarding-Dropped-Pages* technique in the new algorithm. This technique provides that the client memories can also be exploited for the dropped pages if they are the only copies in the system.

Recall that the major aim of the global memory management algorithms is to reduce disk I/O. Disk access delay has a significant impact on the performance of the system. If a page requested from the server is not in the global memory, the server reads the page from its disk. As the portion of the database in global memory increases, the number of disk I/Os due to page requests is decreased. Figure 3 displays the average number of disk I/Os needed for each transaction to be completed. It can be seen from the figure that the new technique is quite effective in reducing the amount of disk I/O.

In all of the algorithms, clients send the same number of page requests to the server. The base algorithm has the least number of the messages exchanged for each committed transaction, as it does not forward page request messages to clients and also does not send dropped pages to the server or the other clients. Our algorithm involves a little bit more messages per commit than Franklin's algorithm. The *Forwarding-Dropped-Pages* technique leads to an increase in the number of messages. There are two different types of messages exchanged in the system. One is the control message that is small in size (page request, lock request, forwarding messages, for

FIGURE 2. Portion of the database available in global memory (50% server memory, 5% client memory, slow network).

FIGURE 3. Total number of disk I/Os per commit (50% server memory, 5% client memory, slow network).

example) and the other is the data message that is large in size (i.e. data page message). Total number of message bytes transmitted for each transaction determines the message volume per transaction. The extra messages involved with the *Forwarding-Dropped-Pages* technique are just control messages, so that they do not cause much increase in the message volume. Figures 4 and 5 present the average number and the volume of messages exchanged per transaction for all three algorithms. With ours and Franklin's algorithm, although the number of messages increases with increasing number of clients, a decrease in the message volume is observed. As more clients are added to the system, the probability that a page dropped at a client has other copies at the remote sites becomes higher. As a result, a smaller number of dropped pages are sent to the other sites and the total message volume decreases.

The throughput results of the algorithms have been obtained for different values of *ClientMemSize*, *ServerMemSize* and *NetworkBandwidth*. The ranges of *ClientMemSize* and *ServerMemSize* experimented are [5% and 50% of *DatabaseSize*] and [10%, 100% of *DatabaseSize*], respectively. In order to keep the discussion brief, we present only the results obtained with two extreme values for each of these two parameters. Figures 6 and 7 display the throughput results obtained by setting *ClientMemSize* values to 5 and 50% of *DatabaseSize*, respectively. *ServerMemSize* is fixed at 50% of *DatabaseSize* and a slow network is used (i.e. *NetworkBandwidth* is 8 Mbits/s). As expected, the base algorithm exhibits the worst performance as it requires the highest

FIGURE 4. Number of messages sent per commit (50% server memory, 5% client memory, slow network).

amount of disk I/O. With a slow network and a small client memory, the new algorithm performs better than Franklin's algorithm (Figure 6). However, when the client memory size becomes large, the performances of

FIGURE 5. Message volume per commit (50% server memory, 5% client memory, slow network).

FIGURE 6. Throughput (50% server memory, 5% client memory, slow network).

these two algorithms become comparable to each other (Figure 7). The *Forwarding-Dropped-Pages* technique employed by the new algorithm does not help much under such a condition as the global memory is large

FIGURE 7. Throughput (50% server memory, 50% client memory, slow network).

FIGURE 8. Throughput (50% server memory, 5% client memory, fast network).

enough to keep the replicated copies. The performance benefit provided by that technique due to less disk I/O is outweighed by the overhead of issuing larger number of messages under a slow network.

Figures 8 and 9 illustrate the throughput results

FIGURE 9. Throughput (50% server memory, 50% client memory, fast network).

obtained with a fast network (i.e. *NetworkBandwidth* is 80 Mbits/s). The base algorithm is again out-performed by the other two algorithms. Also, in this case, the new algorithm provides better performance than Franklin's algorithm for both the small and the large client memory sizes. When the network is fast, the message delay does not seriously affect the performance and the disk I/O becomes the basic factor that determines the performance of the system. As discussed before, fewer disk I/Os are issued when the new algorithm is used. When *ClientMemSize* is increased, the difference between the performance of Franklin's algorithm and the new algorithm becomes less. This is because the global memory management techniques that differentiate the algorithms play a less significant role in determining the performance since storing replicated copies of data pages in global memory does not lead to much disk I/O when the size of the global memory is large.

The performance impact of the server memory size is presented for two extreme cases where the server memory size is 10 and 100% of the database size. 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 Figures 10 and 11. The new algorithm provides the best performance when the server memory size is small. With a small memory size at the server, if Franklin's algorithm is used, the dropped pages received by the server can cause page replacements in the server's memory. More disk I/O is experienced due to the requirement to reload replaced pages at a later time.

When the server memory size was set to be equal to the

FIGURE 10. Throughput (10% server memory, 10% client memory, slow network).

FIGURE 11. Throughput (100% server memory, 10% client memory, slow network).

database size, it was observed that the base algorithm performs better than the other algorithms. The server has all the database in its memory, and thus the memory management algorithms used to increase the portion of the database in global memory are not required. The implementation overhead of the memory management techniques, which are redundant under such a condition, leads to poor performance for both Franklin's and the new algorithms. The performances of these two algorithms are indistinguishable under the extreme condition that the server memory size is equal to the database size.

When a fast network is employed, similar to the results obtained with the slow network, the new algorithm provides the best performance when the server memory size is small (see Figure 12). And again, both the new algorithm and Franklin's algorithm lead to worse performance than the base algorithm under the extreme case that the server memory size is equal to the database size (see Figure 13); however, in this case, the performances of both algorithms are closer to that of the base algorithm, compared with the results obtained with the slow network. This observation can be explained by the fact that the overhead of redundant messages involved in the new algorithm and Franklin's algorithm becomes less with high speed networks.

5.1. Summary of the results of other experiments

The size of the database in our client-server DBMS model was set to 1250 pages in all experiments described in the preceding section. The purpose of assuming a

FIGURE 12. Throughput (10% server memory, 10% client memory, fast network).

small database size was to obtain a high level of data conflicts to be able to include the effects of scheduling decisions of the CB-A cache consistency algorithm. What is important in evaluating the relative effectiveness of the global memory management algorithms, on the

FIGURE 13. Throughput (100% server memory, 10% client memory, fast network).

FIGURE 14. Throughput (10% server memory, 10% client memory, fast network) with a database size of 25,000 pages.

other hand, is the ratio of server's and clients' memory sizes to the database size, rather than the actual database size.

For a more complete study of the algorithms' performance for different levels of data conflict, the algorithms were evaluated under varying values of the database size. As database size got larger, less data contention occurred due to fewer data access conflicts, and the throughput results of all the algorithms were improved. However, as expected, the relative performance of the algorithms was not affected. As an example, Figure 14 illustrates the performance results of the algorithms with a database size of 25,000 pages which is a much larger value compared with that used in the experiments presented in the preceding section. In this experiment, each of the client memory size and the server memory size was fixed at 10% of the database size, and a fast network (with *NetworkBandwidth* = 80 Mbits/s) was employed. Comparing the results with those presented in Figure 12 (which were obtained with the same parameter values except the database size of 1250 pages), one can see the similar trends in the relative performance of the algorithms.

Other experiments were conducted to evaluate the impact of various types of workloads that involve different levels of locality of access at each client. The performance behaviour of the algorithms under those workloads were not much different from the results presented in the previous section. One considerable observation was that as the level of locality decreases, the performance improvement of Franklin's and the new algorithms

over the base algorithm becomes more pronounced. This is because, as the level of client–client and/or client–server interaction increases, these algorithms get the chance to apply their memory management techniques more frequently.

6. CONCLUSIONS

A typical client–server system interconnects a number of computers on which client processes execute and request services from server processes for the end-users. In a client–server DBMS, the whole database is stored on the server disks. Global memory of a client–server DBMS comprises the servers' memories and the clients' memories that can be used to cache copies of database items. Efficient management of global memory is necessary to prevent excessive disk access activity, since disk I/O is the most significant factor determining the system performance.

A new global memory management technique, called *Forwarding-Dropped-Pages*, has been introduced in this paper. The *Forwarding-Dropped-Pages* technique is intended to increase the portion of the database available in global memory, and thus to reduce the amount of disk I/O. In this technique, when a client is going to drop a page which is the only copy in the global memory, the page is sent to the server if it has an available space in its local memory. If the server's memory is full, a remote client is found (if any) that has a free memory space to receive the dropped page. As a result, the single copy of the page is kept in global memory.

The performance of the memory management algorithm that employs this new technique has been evaluated using a detailed simulation model of a client–server DBMS. Performance results of the algorithm have been compared with the results obtained with a base algorithm that does not involve any global memory management techniques, and an algorithm that was proposed by Franklin. Both our new algorithm and Franklin's algorithm have out-performed the base algorithm under most of the conditions tested. For the performance of Franklin's algorithm, our results obtained in terms of the improvement in throughput over the base algorithm seem to be in accordance with the results provided in [6, 7].

When the number of data pages that can be cached in the memories of the clients or the server is small, the use of an efficient global memory management algorithm becomes more crucial for system performance. Our algorithm has exhibited the best performance for such conditions, since it is more effective than other algorithms in increasing the portion of the database in global memory. Small-sized memories require to be managed efficiently to be able to store as many distinct pages as possible in order to reduce the large number of disk I/Os. If, on the other hand, the memory size of the server is large enough to hold most of the database pages, it is redundant to apply specific memory management techniques. Both the new algorithm and Franklin's algorithm

have performed a slightly worse than the base algorithm for the extreme case where the server buffer size has been set to be equal to the database size. The most significant performance gains of the new algorithm have been provided when the underlying network of the system was fast. This is because, with a fast network, the disk access delay rather than the message delay, becomes the dominant factor determining the system performance, and the new algorithm is effective in reducing the number of disk I/Os. This is an encouraging result, because it is expected that in the near future high-bandwidth networks such as FDDI will be dominantly used in client–server systems [18]. With a slow network, the performance benefit of the new algorithm has not reached the level achieved with a fast network. However, the comparative performance results have not been affected to a large degree because the extra messages involved with the algorithm are just the control messages, so that they do not cause much increase in the message volume.

As a final remark, we would like to state that although the proposed algorithm has been studied on a 'one-server, multiple-clients' DBMS model, it can easily be adapted to a 'multiple-server, multiple-clients' architecture as well. The modifications required in the algorithm for that purpose would be as follows. When a client requires a page that does not reside in its local memory, the client requests the page from the server that stores that page on its disk. A global data dictionary can be assumed to exist and can be referred to in order to find out the server responsible for keeping the original copy of each page. If the client has to drop a page from its memory in order to receive the new one, the server checks to see if the page to be dropped is the only copy in the global memory. If so, the server tries to keep that page in its memory. However, if its memory is full or does not contain any hated page, the server tries to find another server or a client that has a free memory space to receive the dropped page. If no client or server has a place to store the dropped page in its memory, the page is sent to the server that is responsible for keeping the original copy of that page. The page is then placed into the memory of that server replacing the page with the LRU value.

REFERENCES

- [1] Carey, M. J., Franklin, M. J., Livny, M. and Shekita, E. (1991) Data Caching Trade-offs in Client–Server DBMS Architectures. *Proceedings of ACM SIGMOD Conference*, Denver, CO.
- [2] Franklin, M. J. and Carey, M. J. (1992) Client–Server Caching Revisited. *Proceedings of the International Workshop on Distributed Object Management*, Edmonton, Canada.
- [3] Wang, Y. and Rowe, L. A. (1991) Cache Consistency and Concurrency Control in a Client/Server DBMS. *Proceedings of ACM SIGMOD Conference*, Denver, CO.
- [4] Wilkinson, K. and Neimat, M. (1990) Maintaining Consistency of Client-Cached Data. *Proceedings of the International Conference on Very Large Data Bases*, Brisbane, Australia.

- [5] Howard, J. H. *et al.* (1988) Scale and Performance in a Distributed File System. *ACM Trans. Comput. Syst.*, **6**, 51–81.
- [6] Franklin, M. J., Carey, M. J. and Livny, M. (1992) Global Memory Management in Client–Server DBMS Architectures. *Proceedings of the International Conference on Very Large Data Bases*, Vancouver, Canada.
- [7] Franklin, M. J. (1993) *Caching and Memory Management in Client–Server Database Systems*. Ph.D. Thesis, Computer Sciences Department, University of Wisconsin-Madison, WI.
- [8] Pu, C., Florissi, D., Soares, P., Wu, K. and Yu, P. S. (1993) Performance Comparison of Dynamic Policies for Remote Caching. *Concurrency: Practice and Experience*, **5**, 239–256.
- [9] Franklin, M. J., Carey, M. J. and Livny, M. (1993) Local Disk Caching for Client–Server Database Systems. *Proceedings of the International Conference on Very Large Data Bases*, Dublin, Ireland.
- [10] Carey, M. J., Franklin, M. J. and Zaharioudakis, M. (1994) Fine-Grained Sharing in a Page Server OODBMS. *Proceedings of ACM SIGMOD Conference*, Minneapolis, MN.
- [11] Acharya, S., Alonso, R., Franklin, M. J. and Zdonik, S. (1995) Broadcast Disks: Data Management for Asymmetric Communication Environments. *Proceedings of ACM SIGMOD Conference*, San Jose, CA.
- [12] Acharya, S., Franklin, M. J. and Zdonik, S. (1996) Prefetching from a Broadcast Disk. *Proceedings of the International Conference on Data Engineering*, New Orleans, LA.
- [13] Amsaleg, L., Franklin, M. J. and Gruber, O. (1995) Efficient Incremental Garbage Collection for Client–Server Object Database Systems. *Proceedings of the International Conference on Very Large Data Bases*, Zurich, Switzerland.
- [14] Franklin, M. J., Johsson, B. T. and Kossmann, D. (1996) Performance Trade-offs for Client–Server Query Processing. *Proceedings of ACM SIGMOD Conference*, Montreal, Canada.
- [15] Çetintemel, U., Çınar, A., Tunali, G. D. and Ulusoy, Ö. (1994) A Simulation Model for Client–Server Database Management Systems. *Proceedings of the International Symposium on Computer and Information Sciences*, Antalya, Turkey.
- [16] Schwetman, H. (1986) CSIM: A C-Based, Process-Oriented Simulation Language. *Proceedings of the Winter Simulation Conference*.
- [17] Ferrari, D. (1978) *Computer Systems Performance Evaluation*. Prentice-Hall.
- [18] DeWitt, D. J., Fattersack, P., Maier, D. and Velez, F. (1990) *A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems*. Computer Sciences Technical Report No. 936, University of Wisconsin-Madison, WI.