Implementing Crash Recovery in QuickStore: A Performance Study

Seth J. White[1]
Sun Microsystems
2550 Garcia Ave.
Mountain View, CA 94043-1100
seth.white@eng.sun.com

David J. DeWitt
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
dewitt@cs.wisc.edu

ABSTRACT

Implementing crash recovery in an Object-Oriented Database System (OODBMS) raises several challenging issues for performance that are not present in traditional DBMSs. These performance concerns result both from significant architectural differences between OODBMSs and traditional database systems and differences in OODBMS's target applications. This paper compares the performance of several alternative approaches to implementing crash recovery in an OODBMS based on a client-server architecture. The four basic recovery techniques examined in the paper are termed page differencing, sub-page differencing, whole-page logging, and redo-at-server. All of the recovery techniques were implemented in the context of QuickStore, a memory-mapped store built using the EXODUS Storage Manager, and their performance is compared using the OO7 database benchmark. The results of the performance study show that the techniques based on differencing generally provide superior performance to whole-page logging.

1. Introduction

This paper examines the performance of several alternative approaches to implementing crash recovery in QuickStore [White94], a memory-mapped store for persistent C++ that was built using the EXODUS Storage Manager (ESM) [Carey89]. Providing recovery services in a system such as QuickStore raises several challenging implementation issues, not only because it is a memory-mapped storage system, but also because of the kinds of applications that Object-Oriented Database Management Systems (OODBMSs) strive to support, i.e. CAx, GIS, OIS, etc. Furthermore, since QuickStore is implemented on top of the EXODUS Storage Manager (ESM), it is a client-server, page-shipping system [DeWitt90]. This raises additional performance concerns for recovery that are not present in database systems based on more traditional designs, i.e. centralized DBMSs or systems based on a query-shipping architecture.

The paper examines four basic recovery techniques that are termed page differencing, sub-page differencing, whole-page logging, and redo-at-server. All of the recovery techniques were implemented in the context of QuickStore/ESM so that an accurate comparison of their performance could be made. The performance study was carried out using the OO7 object-oriented database benchmark [Carey93]. The performance results illustrate the impact of different database sizes, update patterns, and available client memory on the relative performance of the various techniques. In addition, the number of clients accessing the database is varied in order to compare the scalability of the different recovery algorithms.

The remainder of the paper is organized as follows. Section 2 presents a detailed discussion of the factors (briefly mentioned above) that make recovery in QuickStore a challenging problem. Section 3 describes several alternative techniques for implementing recovery in QuickStore. Next, Section 4 describes the performance study that was carried out to compare the performance of the different recovery schemes. Section 5 presents the performance results. Section 6 discusses related work. Finally, Section 7 presents our conclusions and some proposals for future work.

2. Challenges for Recovery

One challenge faced by QuickStore recovery results from the way that persistent objects are accessed under its memory-mapped architecture. As described in [White94], QuickStore supports a comprehensive pointer swizzling strategy that allows application programs to manipulate persistent objects directly in the ESM client buffer pool by dereferencing normal virtual memory pointers. This strategy allows applications to update persistent objects at memory speeds with essentially no overhead, but it also makes detecting the portions of objects that have been updated more difficult than in systems that use traditional implementation techniques. For example, in OODBMSs that do not perform pointer swizzling or that implement pointer swizzling using traditional software-based techniques, a function that is part of the database runtime system is typically called to perform each update. This provides a hook that the system can use to record the fact that the update has occurred. We note that this approach requires special compiler support to insert function calls for updates into the application code in order to make updates transparent.

A second factor affecting the design of a recovery scheme for QuickStore is that QuickStore, like most OODBMSs, is designed to handle non-traditional database applications, e.g. CAD, geographic information systems (GIS), and office information systems (OIS). Applications of this type typically read objects into memory and then work on them intensively, repeatedly traversing relationships between objects and updating the objects as well. This behavior differs dramatically from that exhibited by relational database systems, which usually update an individual tuple just once during a particular update operation. For example, giving all of the employees in a company an annual raise requires only a single update to each employee tuple. Since relational database systems typically update each tuple only once, they generate a log record for recovery purposes for each individual update. However, such a strategy is not practical in an OODBMS where an object may be updated many times during a single method invocation. Here it is necessary to batch the effects of updates together in order to achieve good performance by attempting to generate a single log

[1] The research contained in this paper was conducted while the author was a graduate student at the University of Wisconsin-Madison.

This research is sponsored by the Advanced Research Project Agency, ARPA order number 018 (formerly 8230), and monitored by the U.S. Army Research Laboratory under contract DAAB07-91-C-Q518.

record that records the effects of several updates to an object.

A final consideration in the design of a recovery scheme for QuickStore arises from the fact that QuickStore is based on a client-server architecture in which updates are performed at the client workstations. This raises the issue of cache consistency between the clients and the server [Frank93] since both the client and the server buffer pools can contain cached copies of a page that has been updated. In addition, to increase availability, the stable copy of the transaction log is maintained by the server, so clients are required to ship log records describing updates over a network to the server before a transaction can commit. This differs from the traditional approach used in centralized database systems, where updates are performed at the server and log records are generated locally at the server as well.

3. Recovery Strategies

This section describes the four basic recovery schemes that were implemented and evaluated using QuickStore: page differencing, sub-page differencing, whole-page logging, and redo-at-server. We begin by describing the implementation of recovery in ESM since several of the techniques are built on top of or involve modifications to the underlying ESM recovery scheme. It should be noted that although the recovery algorithms are discussed in the context of QuickStore, they are not QuickStore specific and, in general, are applicable to any similar client-server OODBMS.

3.1. Recovery in ESM

The EXODUS Storage Manager is a client-server, page-shipping system [DeWitt90] in which both clients and servers manage their own local buffer pools. When a client needs to access an object on a page that is not currently cached in its local buffer pool, it sends a request (usually over a network) to the appropriate server asking for the page. If necessary, the server reads the page from secondary storage into main memory, sends a copy of the page to the client, and retains a copy of the page in its own buffer pool as well.

Objects are updated at clients and clients also generate log records that describe updates for recovery purposes. Log records for updates contain both redo and undo information for the associated update operation. For example, if a range of bytes within an object is updated, then the log record will contain the old and new values of that portion of the object. Log records are collected and sent from a client to the server a page-at-a-time. For simplicity, ESM enforces the rule that log records generated for a page are always sent back to the server before the page itself is sent. Thus, the server never has a page cached in its buffer pool for which it does not also have the log records describing the updates present on the page.

The ESM server manages a circular, append-only log on secondary storage and uses a STEAL/NO-FORCE buffer management policy [Haerd83]. Clients can cache pages in their local buffer pools across transaction boundaries. However, inter-transaction caching of locks at clients is not supported. Also, all dirty pages are sent back to the server at commit time in order to maintain cache consistency between the clients and the server and simplify recovery. The log records generated on behalf of a transaction must be written to the log by the server before the transaction commits, but the dirty pages themselves are not forced to disk. [Frank92] contains a more detailed description of the ESM recovery scheme.

3.2. The Page Differencing Approach

The ESM recovery mechanism handles the generation of log records for updates. However, if recovery were done in a straightforward way using the basic services provided by ESM, then each time an update is performed, a log record would be generated. This is a situation that we would like to avoid, if possible. Furthermore, there is no obvious way for a QuickStore application to detect the fact that an update has occurred since application programs are allowed to update objects by simply dereferencing standard virtual memory pointers.

3.2.1. Enabling Recovery for Page Differencing

The problems mentioned above can be addressed by employing a page differencing (PD) scheme to generate log records. This approach works as follows. QuickStore gives application programs access to persistent data by mapping a range of virtual memory (using the Unix mmap system call) in the address space of the application process to the location in the client buffer pool of a particular database page when it is cached in main memory [White94]. We refer to the range of virtual memory that has been mapped to a page as a *frame* of virtual memory. Virtual memory frames are contiguous and uniform in size (8 Kb). Read permission is enabled on a virtual frame that is mapped to a page in the buffer pool so that persistent objects located on the page can be accessed by the application process. Objects on the page are accessed by dereferencing standard virtual memory pointers into the virtual frame. However, write access is not automatically enabled on a virtual frame that has been mapped. In particular, write access is never enabled when the actions necessary to *enable* recovery for a particular page have not been taken. Thus, any attempt by an application program to update an object on a page for which recovery is not enabled will result in a page-fault, causing the QuickStore fault-handling routine to be invoked.

The QuickStore runtime system maintains an in-memory table that contains an entry (called a page descriptor) for each virtual frame that has been associated with a page in the database. The in-memory table is implemented as a height balanced binary tree. When the fault-handling routine is invoked, it begins by searching the in-memory table for the page descriptor corresponding to the virtual memory address that caused the fault. By inspecting status information contained in the page descriptor entry, the fault-handler will detect that the access violation is due to a write attempt. If recovery is not already enabled on the page—it may be if paging in the buffer pool is taking place—then the fault-handler copies the page into an area in memory termed the *recovery buffer* and sets the page descriptor entry to point to the copy. The fault-handler also obtains an exclusive lock on the page from ESM, if needed, and enables write access on the virtual frame that caused the fault. At this point, all of the work needed to enable recovery on the faulted-on page is complete, so control is returned to the application program which can then proceed to update objects on the page directly in the client buffer pool.

Figure 1 shows the effect of the actions described above on the in-memory data structures maintained by QuickStore. In Figure 1 page *a*, which is cached in the client buffer pool, is shaded to show

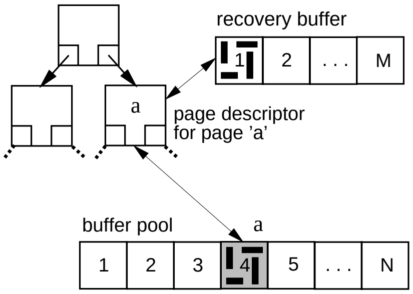

Figure 1. The page diffing approach.

that it has been updated. A copy containing the value of page *a* before recovery was enabled has been placed in the recovery buffer, and the page descriptor for page *a* has been set to point to the copy. As Figure 1 shows, the recovery buffer contains room for *M* pages, where *M* is fixed and $1 \leq M \leq N$ (where *N* is the number of pages in the ESM client buffer pool). Since *M* is fixed, the recovery buffer can become full, so it may be necessary to free up space periodically when an additional page is updated, by generating log records for a page that has already been copied into the recovery buffer. Space in the recovery buffer is managed using a simple FIFO replacement policy.

3.2.2. Generating Log Records for Pages

At transaction commit time, when paging in the buffer pool occurs, or when the recovery buffer becomes full, the old values of objects contained in the recovery buffer and their corresponding updated values in the buffer pool are compared (diffed) to determine if log records need to be generated. The actual algorithm used for generating log records is slightly more sophisticated than the simple approach of generating a single log record for each modified region of an object. This simple approach was rejected since it has the potential to generate a great deal of unnecessary log traffic. For example, consider an object in which the first and third words (1 word = 4 bytes) have been updated. The simple approach would generate two log records for the object. Since each ESM log record contains a header of approximately 50 bytes, the total space used in the log would be 116 bytes (50 bytes for each log header plus 4 bytes for each before and after image). On the other hand, if just one log record were generated, only 74 bytes would have been used (50 bytes, plus 12 bytes for each before and after image), providing a 36% savings in the amount of log space used.

The algorithm for generating log records uses diffing to identify consecutive modified regions in each object on a page; if only one region exists then a single log record is generated for the object. (Log records could, in principle, span objects but the current implementation of recovery in ESM does not allow this.) For example, Figure 2a shows an object that contains three modified regions, labeled *R1*, *R2*, and *R3*. The diffing algorithm starts from the beginning of the object, so initially it would identify the two modified regions *R1* and *R2*. It is easy to show, given the before/after-image format of log records in ESM, that if the distance *D1* between *R1* and *R2* satisfies the equation $2 * \text{size}(D1) > H$, where *H* is the size of a log record header, then generating separate log records for each region will generate the least amount of log traffic. If this were the case in the example, then the algorithm would generate a log record for *R1* before proceeding to consider additional regions. However, a log record for *R2* is not immediately generated, since it may be advantageous to combine *R2* with some region that has not yet been discovered. If, on the other hand, $2 * \text{size}(D1) <= H$, i.e. the distance between *R1* and *R2* is small, then the algorithm combines the two regions into a single region. Figure 2b illustrates the second case. Here, *R1* and *R2* have been combined into a single region *R1’*. Again, no log record is generated at this step, as the algorithm may decide to combine *R1’* with additional regions.

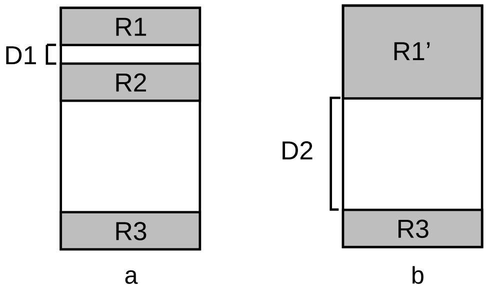

Figure 2. Combining modified regions in an object.

In either case mentioned above, the algorithm continues by identifying the next modified region in the object (if there is one) and repeats the previous check using the newly discovered region and either the combined region from the previous iteration or the region from the previous iteration for which no log record has yet been generated. In the example shown in Figure 2b, the algorithm would next examine regions *R1’* and *R3* to see if they should be logged separately or combined. Since the distance between *R1’* and *R3* is large, the algorithm will generate separate log records for *R1’* and *R3*. Finally, we note that the decision concerning whether or not to combine consecutive modified regions depends only on the distance between them and not on their size, so the order in which the regions are examined does not matter. Thus, the algorithm is guaranteed to generate the minimum amount of log traffic.

3.3. The Sub-Page Differencing Approach

The page differencing recovery scheme described in the previous section has some potential disadvantages. The most obvious disadvantage is that the CPU overhead for copying and diffing a whole page may be fairly high, especially when very few updates have actually been performed on the page. In addition, page diffing has the potential to waste space in the recovery buffer by copying a whole page when only a few objects on the page have been updated. This can increase the number of log records generated during a transaction, if the recovery buffer becomes full. However, the page-wise granularity of the page diffing scheme is necessary if applications are allowed to update objects via normal virtual memory pointers as virtual memory is page-based.

An alternative approach is to interpret update operations in software. One way that this can be accomplished, in general, is by compiling persistent applications using a special compiler that inserts additional code to handle update operations. This code can simply be a function call that replaces the usual pointer dereference at the points in the application program where updates occur. In our case, the function that is invoked is part of the QuickStore runtime system. This approach yields a system in which objects may be read at memory speed using standard virtual memory pointers, but in which update operations are more heavy-weight, requiring a function call and other software overhead. The hope when using such an approach is that the extra cost incurred on each update will be repaid through reduced recovery costs.

Figure 3 illustrates the in-memory data structures used by QuickStore to implement the sub-page differencing (SD) approach. Under the SD approach, each page is divided into a contiguous sequence of regions called blocks. Blocks are uniform in size (We experimented with block sizes ranging from 8 to 64 bytes.). As Figure 3 shows, the page descriptor for a page that has been updated holds a pointer to an array containing pointers to copies of blocks that have been modified. There is one entry in the array for each block on the page, and array entries for unmodified blocks are null. Blocks were used as the sub-page unit of copying and diffing instead of objects for two reasons. First, it is cheaper in terms of CPU cost to identify the block on a page that is being updated than it is the object, when an update occurs (see below). And second, objects within a page may be rather big (i.e. up to 8K-bytes in size). If this is the case, then the advantages of the sub-page diffing approach will be lost when updates are sparse.

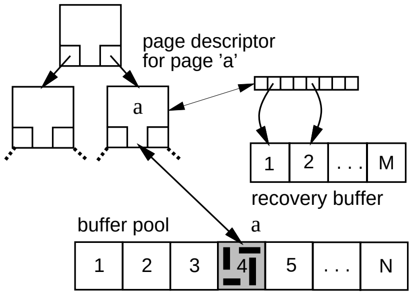

Figure 3. The Sub-page diffing approach.

3.3.1. Enabling Recovery for Sub-Page Differencing

Each time the QuickStore update function is called, it looks up the page descriptor of the appropriate page using the address of the memory location that is being updated (which is passed as a parameter). Once the page descriptor is found, a check is made to see if the block that is about to be updated has been copied. This check is relatively inexpensive since the address of the location in memory being updated can be used to index the array of block pointers contained in the page descriptor (after applying some simple logical operations). If a copy of the block has not yet been made, then a copy is placed in the recovery buffer. In addition, the status flags in the page descriptor are examined to see if an exclusive lock has been acquired for the page, and write access is enabled on the virtual frame mapped to the page (if it is not already). Finally, the update itself is performed. We note that write access could be enabled automatically on virtual frames when using the sub-page approach. We chose not to do this because not enabling write access allows the runtime system to catch erroneous writes to virtual frames that have not been updated, and because the extra cost of the approach we used is very low.

3.3.2. Generating Log Records for Sub-Pages

Like the page diffing approach, the SD approach generates log records at transaction commit time, when a modified page is paged out by the buffer manager, or when the recovery buffer becomes full. Log records can be generated by diffing the original values of the blocks contained in the recovery buffer with the modified versions of the same blocks located in the buffer pool, using the diffing scheme described in Section 3.2. In addition, one could avoid the expense of diffing altogether by simply logging entire blocks. We experimented with both techniques, and refer to the sub-page approach without diffing as sub-page logging (SL) in the upcoming discussion on performance.

3.4. The Whole-Page Logging Approach

This section describes the third recovery algorithm included in the study. This algorithm is termed whole-page logging (WPL) since entire modified pages are written to the log instead of log records for updated regions of objects. We note that WPL is also the basic approach used in ObjectStore [Lamb91], a commercial OODBMS product. The advantages of WPL are that it avoids the client CPU cost that is incurred by the two diffing schemes for copying and diffing. WPL also avoids the memory overhead at clients for storing the original values of pages/blocks. This can potentially improve performance by allowing WPL to allocate more memory to the client buffer pool and, thereby, generate fewer log records. Finally, WPL allows applications to update objects at memory speeds by dereferencing normal virtual memory pointers, so the cost of actually performing updates is low.

The disadvantage of whole-page logging is that it logs entire after-images of dirty pages at the server. This means that all of the pages dirtied by a transaction must be forced to the log at the server before the transaction commits. We note, however, that the cost of shipping dirty pages back to the server does not add any additional costs in ESM since its recovery algorithm always sends updated pages back to the server when a transaction commits (see Section 3.1). The WPL scheme does not rely on the support provided by ESM for recovery as the diffing schemes do. Thus, WPL differs from the previous schemes in the actions taken at both clients and the server to support recovery.

3.4.1. Actions Performed at Clients

The whole-page logging algorithm works as follows at the clients. When an application first attempts to update a page at a client, a page-fault is signaled, as usual. The QuickStore page-fault handling routine marks the copy of the page cached in the buffer pool as dirty, in addition to requesting an exclusive lock if necessary, and enables write access on the virtual memory frame that is mapped to the page. Control is then returned to the application program. Dirty pages are shipped back to the server when the transaction commits, or possibly sooner if paging in the client buffer pool occurs. Note that no log records are generated for updates at the clients under this approach; only dirty pages are shipped back to the server.

3.4.2. Actions Performed at the Server

When the server receives a dirty page, it appends the page to the log and caches a copy of the page in its own buffer pool. The server does not allow the original copy of the page on disk to be overwritten with the new copy of the page until after the transaction that updated the page commits. If paging in the server buffer pool causes a dirty page to be replaced during a transaction, then the page is read from the log if it is reaccessed during the same transaction. When a transaction reaches its commit point, the original values of any updated pages are still located on disk in their permanent locations, and the updated values of the pages have been flushed to the log together with a commit log record for the transaction. This makes it possible to abort a transaction at any time before the commit point is reached by simply ignoring, from then on, any of its updated values of pages located in the log or cached in memory. No undo processing for updates is required.

A page updated by a committed transaction, must be maintained in the log until one of two things happens. The first is that the page is read from the log and used to overwrite its permanent location on disk. The log space for the page can then be reused since the copy of the page contained in the log is no longer needed for recovery. The reason for this is fairly obvious. For example, suppose that transaction *T* updates page *P* and then commits, i.e. *P* is forced to the log. If a crash resulting in the loss of the server's volatile memory occurs any time after *T* commits, but before *P* overwrites its permanent location on disk, then the value of *P* stored in the log must be available in order for the system to correctly restart. If *P* has safely overwritten its permanent location on disk, however, then that copy of *P* can be used following a restart. Space for a page in the log can also be reused if a subsequent transaction updates the page and commits, thereby forcing a new copy of the page, say *C2,* to the log, before the initial copy of the page *C1* in the log overwrites the permanent location of the page. In effect, *C1* is not needed at this point, since following a crash *C2* will be used. Note, however, that both *C1* and *C2* must be maintained in the log until the transaction that wrote *C2* commits.

The server maintains an in-memory table, called the WPL table, to keep track of pages contained in the log that are needed for recovery purposes. Each table entry contains the page id (PID) of the page, which identifies its permanent location on disk. Entries also contain the log sequence number (LSN) of the log record generated for the page. The LSN identifies the physical location of the page in the log. Additional fields stored in each entry include the transaction id (TID) of the transaction that last dirtied the page and some additional status information. When a page is initially written to the log, the status information records the fact that the transaction that dirtied the page has not yet committed. Finally, table entries contain a pointer that refers to the entry for a previously logged copy (if any) of the same page if that copy is still needed for recovery.

The server also maintains a list, for each active transaction, of the pages that have been logged for that particular transaction. When a transaction commits the WPL table entry for each page on this list is updated to show that the transaction that modified the page has committed. In order to reclaim log space there is a background thread that asynchronously reads pages modified by committed transactions from the log. (As an optimization, pages modified by a transaction that are still cached in the server buffer pool at commit time are simply marked as having been read.) Once a page has been read from the log, the server is free to flush the page to disk at any time. Once this happens, the entry for the page in the WPL table is removed.

3.4.3. Recovering from a Crash

In order to be able to recover from an unexpected crash, the server periodically takes checkpoints. At checkpoint time, the WPL table is written to the log. In order to recover from an unexpected failure, the server must be able to reconstruct the WPL table so that it contains entries for all of the pages that have been updated by committed transactions, but not yet written to their permanent disk locations. Once this is done, the server can resume normal operation.

Restart after a crash requires a single pass through the log that begins at the end of the log and proceeds backward to the most recent checkpoint record. At the start of the pass, a list that records committed transactions, termed the committed transactions list (CTL), is initialized to empty. During the pass a transaction is added to the CTL when a commit record for the transaction is encountered during the backward scan. When a log record for a modified page is encountered, an entry for the page is inserted into the WPL table if the transaction that updated the page is in the committed transaction list. Otherwise, the modify record is ignored since the associated transaction did not commit before the crash.

When the checkpoint record is reached, the committed transaction list contains an entry for each transaction that committed after the checkpoint was taken. The contents of the checkpoint record itself are then examined. Entries in the checkpoint record that pertain to members of the CTL, or which are marked as pertaining to transactions that committed before the checkpoint was taken, are added to the newly constructed WPL table. At this point the WPL table has been fully reconstructed and normal processing can resume.

3.5. The Redo-at-Server Approach

The final recovery algorithm that we examined is termed redo-at-server (REDO). This algorithm is a modification of the ARIES-based recovery algorithm used by ESM in which clients send log records, but not dirty pages, back to the server. (Recall that the usual EXODUS recovery algorithm sends both log records and dirty pages back to the server, although only log records are forced to disk before commit.) Under REDO, when a log record is received at the server the redo information in the log record is used to update the server's copy of the page. The disadvantage of REDO is that the server may have to read the page from secondary storage in order to apply the log record.

In addition to the obvious advantage of not having to ship dirty pages from clients to the server, REDO is also appealing from an implementation standpoint. It simplifies the implementation of a storage manager by providing cache consistency between clients and the server [Frank93]. REDO is currently being used in the initial version of SHORE [Carey94], a persistent object system being developed at Wisconsin. Since REDO only involves changes at the storage manager level, it can be used in combination with any of the recovery schemes mentioned previously that make use of the recovery services provided by EXODUS. Here, we will study its use in conjunction with the PD scheme.

3.6. Implementation Discussion

Each of the four recovery schemes described in this section has been implemented in the context of QuickStore/ESM. The two diffing schemes did not require any changes to the base recovery services provided by ESM. Instead of modifying the gnu C++ compiler—the gnu compiler was used to compile the QuickStore application code—to insert function calls for updates to support the sub-page diffing approach, the necessary function calls were inserted by hand at the application level to save time. In implementing whole-page logging we made use of the existing ESM recovery code whenever possible. For example, ESM already supported whole-page logging for newly created pages. The changes made to the ESM client to support WPL were therefore fairly minor. The changes made to the server were more substantial, and involved tasks such as maintenance of the WPL table (Section 3.4) and rereading pages from the log. Implementing redo-at-server was also relatively easy. The ESM server already supported redo as part of the ARIES-based recovery scheme that it uses, and it was not hard to get the server to apply each log record to the appropriate page as log records were received by inserting a function call in the appropriate spot in the server code.

4. Performance Experiments

This section describes the performance study that was conducted to compare the performance of the recovery algorithms described in the previous section. The OO7 object-oriented database benchmark was used as a basis for carrying out the study [Carey93].

4.1. OO7 Benchmark Database

The structure of the OO7 database benchmark is discussed in detail in [Carey93], but we describe it briefly here for completeness. The OO7 database is intended to be suggestive of many different CAD/CAM/CASE applications. A key component of the database is a set of *composite parts*. Each composite part is intended to suggest a design primitive such as a register cell in a VLSI CAD application. Associated with each composite part is a *document* object that models a small amount of documentation associated with the composite part. Each composite part also has an associated graph of *atomic parts*. Intuitively, the atomic parts within a composite part are the units out of which the composite part is constructed. One atomic part in each composite part's graph is designated as the "root part". Each atomic part is connected via a bi-directional association to three other atomic parts (This can be varied.). The connections between atomic parts are implemented by interposing a connection object between each pair of connected atomic parts.

Additional structure is imposed on the set of composite parts by a structure called the "assembly hierarchy". Each assembly is either made up of composite parts (in which case it is a *base assembly*) or it is made up of other assembly objects (in which case it is a *complex assembly*). The first level of the assembly hierarchy consists of *base assembly* objects. Each base assembly has a bi-directional association with three composite parts which are chosen at random from the set of all composite parts. Higher levels in the assembly hierarchy are made up of *complex assemblies*. Each complex assembly has a bi-directional association with three subassemblies, which can either be base assemblies (if the complex assembly is at level two in the assembly hierarchy) or other complex assemblies (if the complex assembly is higher in the hierarchy). Each assembly hierarchy is called a *module*. Modules are intended to model the largest subunits of the database application. Each module also has an associated *Manual* object, which is a larger version of a document.

We included two different database sizes in the study, termed small and big. Table 1 shows the OO7 parameters used to construct the two databases. We note that the parameters used here do not correspond exactly to the "standard" OO7 database specification of [Carey93]. As indicated in Table 1, a module in the small database here is the same size as a module in the small database of [Carey93]; however, modules in the big database differ from the small database in that they contain 2,000 composite parts instead of 500, and there are eight levels in the assembly hierarchy in the big database versus seven in the small database. Both the small and big database here contain five modules, as the number of clients that access the database will be varied from one to five. During a given experiment, each module will be accessed by a single client, so a module represents private data to the client that uses it. We decided not let the clients share data in order to avoid locking conflicts and deadlocks both of which can have a major effect on performance. Removing these effects simplified the experiments and allowed us to concentrate on the differences in performance that were due to the recovery mechanisms being studied.

Parameter	**Small**	**Big**
NumAtomicPerComp	20	20
NumConnPerAtomic	3	3
DocumentSize (bytes)	2000	2000
Manual Size (bytes)	100K	100K
NumCompPerModule	500	2000
NumAssmPerAssm	3	3
NumAssmLevels	7	8
NumCompPerAssm	3	3
NumModules	5	5

Table 1. OO7 Benchmark database parameters.

Table 2 lists the total size of the databases and the size of a module within each database. The size of a module in the small database is 6.6 Mb which is small enough that an entire module can be cached in main memory at a client (12 Mb). In addition, the total size of the small database (33 Mb) is small enough that it fits into main memory of the server (36 Mb). Thus, the experiments performed using the small database test the performance of the recovery algorithms when the entire database can be cached in main memory. The size of a module in the big database, however, is larger than the main memory available at any single client. In addition, when more than a single client is used the amount of data accessed is also bigger than the memory available at the server. Experiments performed using the big database test the relative performance of the algorithms when a significant amount of paging is taking place in the system.

	Small	**Big**
module	6.6	24.3
total	33.0	121.5

Table 2. Database sizes (in megabytes)

4.2. OO7 Benchmark Operations

This section describes the OO7 benchmark operations used in the study. Since the goal of the study is to examine recovery performance we only include the OO7 tests that perform updates. In addition, some of the OO7 update tests stress index updates, these tests are also not included in the study as they did not shed any additional light on the performance of the recovery algorithms.

The experiments were performed using the T2A, T2B, and T2C OO7 traversal operations. The T2 traversals perform a depth-first traversal of the assembly hierarchy. As each base assembly is visited, each of its composite parts is visited and a depth first search on the graph of atomic parts is performed. Each T2 traversal increments the (x,y) attributes contained in atomic parts as follows and returns a count of the number of updates performed[2]:

T2A—Update the root atomic part of each composite part.
T2B—Update all atomic parts of each composite part.
T2C—Update all atomic parts four times.

During each experiment, the traversals were run repeatedly at each client, so that the steady state performance of the system could be observed. Each traversal was run as a separate transaction. The client and server buffer pools were not flushed between transactions, so data was cached in memory across transaction boundaries.

4.3. Software Versions

We experimented with several QuickStore recovery software versions. Table 3 shows the names used to identify the different versions in the performance section. Each name generally consists of two parts. The first part identifies the scheme used for generating log records (PD, SD, or SL), and the second specifies the underlying recovery strategy that was used (ESM or REDO). In addition, the size of the recovery buffer given to the diffing schemes is sometimes appended to the name of these systems. For example, PD-REDO-4 denotes a system using page diffing with a 4 Mb recovery buffer in combination with redo-at-server recovery. In the case of whole-page logging the name has only one part (WPL). We note that the sub-page diffing (SD) versions shown in the performance section use a block size of 64 bytes. We experimented with other block sizes, but their performance was similar.

Name	**Description**
PD-ESM	page diffing, ESM recovery
SD-ESM	sub-page diffing, ESM recovery
SL-ESM	sub-page logging(no diffing), ESM recovery
PD-REDO	page diffing, REDO recovery
WPL	whole page logging

Table 3. Example software versions.

4.4. Hardware Used

As a test vehicle we used six Sun workstations on an isolated Ethernet. A Sun IPX workstation configured with 48 megabytes of memory, two 424 megabyte disk drives (model Sun0424) and one 1.3 gigabyte disk drive (model Sun1.3G) was used as the server. One of the Sun 0424s was used to hold system software and swap space. The Sun 1.3G drive was used by ESM to hold the database, and the second Sun 0424 drive was used to hold the ESM transaction log. The data and recovery disks were configured as raw

[2] [Carey93] specifies that the (x, y) attributes should be swapped. We increment them instead so that multiple updates of the same object change the object's value. This guarantees that the diffing schemes always generate a log record for each modified object.

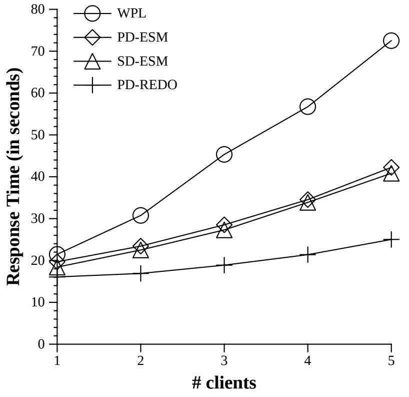

Figure 4. T2A, small database.

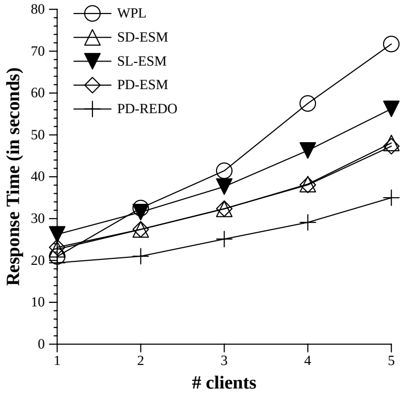

Figure 6. T2B, small database.

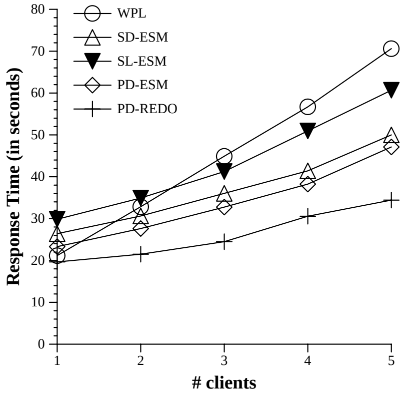

Figure 8. T2C, small database.

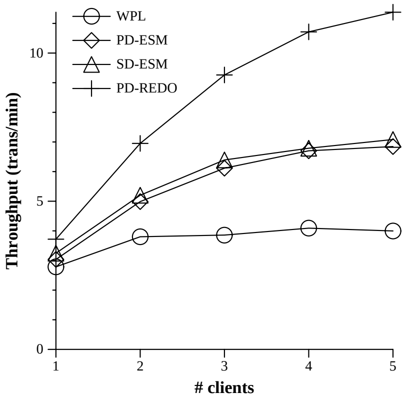

Figure 5. T2A, small database.

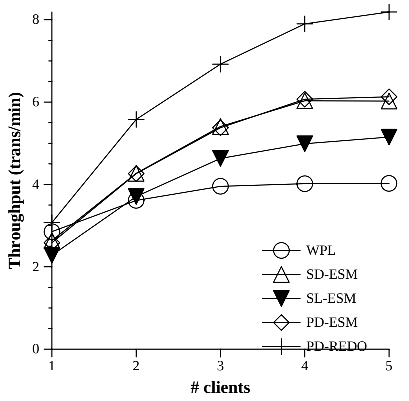

Figure 7. T2B, small database.

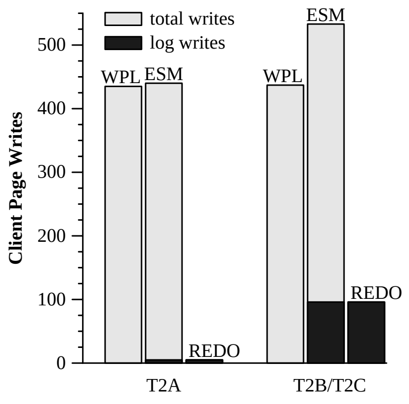

Figure 9. Client Writes, small database.

disks. For the clients we used five Sun Sparc ELC workstations (about 20MIPS) configured with 24 megabytes of memory and one 207 megabyte disk drive (model Sun0207) each. The disk drives were used to hold system software and as swap devices.

5. Performance Results

This section presents the performance results. We first present and analyze the results obtained using the small database and then turn to the results obtained using the big database.

5.1. Unconstrained Cache Results

This section presents results for experiments using the small database. All systems were given 12 megabytes of memory at each client to use for caching persistent data. For the systems that do diffing, 8Mb was allocated for the client buffer pool and 4Mb for the recovery buffer. This allocation of memory allowed all of the persistent data (modified and unmodified) accessed by a client to be cached completely in the client's main memory. Since the small database was used, all of the data accessed by the clients could be cached in the server buffer pool as well.

Figures 4 and 5 show the response time and throughput versus number of active clients for the T2A traversal[3] for several software versions. PD-REDO (page diffing using redo recovery) has the best performance overall, while WPL (whole-page logging) has the worst. WPL is 22% slower than PD-REDO when one client is used, but its performance relative to the other systems steadily worsens as the number of clients increases. At five clients, WPL is 2.4 times slower than PD-REDO. WPL has slow performance in this experiment because T2A does sparse updates, causing WPL to write significantly more pages to the log than the other systems.

Figure 9 shows the total number of pages (data and log) and the number of log record pages shipped from each client to the server on average during a transaction. The results in Figure 9 are labeled according to the underlying recovery scheme used, since that determined the number of pages sent for each system, e.g. PD-ESM and SD-ESM had the same write performance since when no paging occurs at the clients they always generate the same number of log records and dirty pages. The main difference between PD-ESM and SD-ESM in this case is in the amount of data copied into the recovery buffer and diffed per transaction. The number of pages written to the log by the server was very close to the total number of pages shipped for WPL, and it was also close to the number of log pages shipped for the other systems. Figure 9 shows that WPL writes 435 pages back to the server on average during T2A, while PD-REDO writes just 5. Thus, the diffing scheme used by PD-REDO is very effective at reducing the amount work required at the server for recovery in this case.

The performance of PD-ESM and SD-ESM lies between that of the other two systems in Figure 4. Surprisingly, the overall response time of SD-ESM is only slightly faster than PD-ESM (6.5% at 1 client, 3.3% at 5 clients), as the savings in CPU cost provided by SD-ESM were only a small part of overall response time in this experiment. The absolute difference in response time between

[3] T2A: update root atomic part of each composite part.

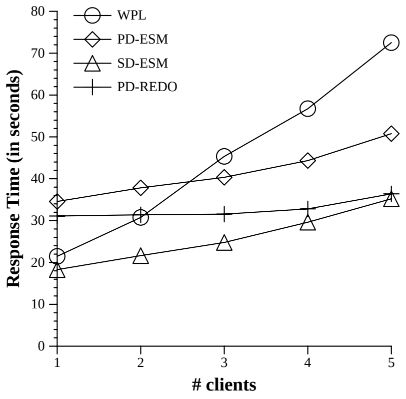

Figure 10. T2A, small, constrained cache.

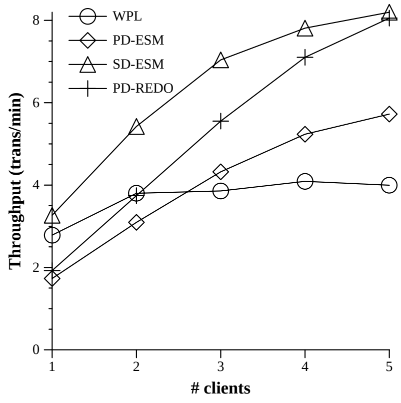

Figure 11. T2A, small, constrained cache.

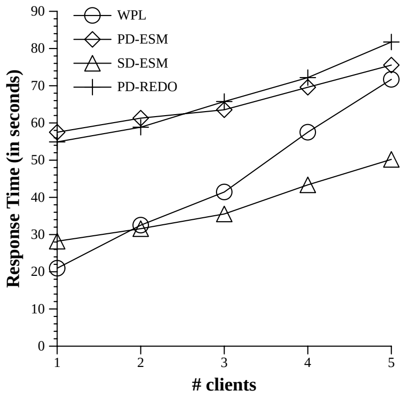

Figure 12. T2B, small, constrained cache.

SD-ESM and PD-ESM did not change significantly as the number of clients varied, and was roughly 1.3 seconds. This amounted to a savings of 3 milliseconds for SD-ESM for each page that was updated. The difference in CPU usage between PD-ESM and SD-ESM in Figure 4 was approximately 8% throughout, which was also quite low. We believe the small difference in CPU usage was caused partly by the CPU overhead at clients for shipping dirty pages back to the server. The response time for SL-ESM is not shown in Figure 4 since it was basically the same as SD-ESM. This was because the number of additional log pages generated by SL-ESM relative to SD-ESM was very small during this experiment.

The throughput results shown in Figure 5 mirror the response time results in Figure 4. Figure 5 shows that while the throughput increases with the number of clients for the systems that use diffing, WPL becomes saturated when more than two clients are used. The increase in throughput for PD-ESM is 56%, and for PD-REDO it is 67% as the number of clients increases from 1 to 5.

We turn next to Figures 6 and 7, which show the results of the T2B[5] traversal. Comparing Figure 6 with Figure 4 shows that the difference in performance between the systems is smaller during T2B, as T2B performs significantly more updates per page than T2A—slowing the performance of the diffing schemes. PD-REDO again has the best overall multi-user performance, however, the difference in performance between PD-REDO and WPL ranges from just 5% to 41% as the number of clients increases since PD-REDO must write a significant number of log records to disk per transaction (Figure 9). WPL is faster than the remaining systems when a single client is used, but its performance degrades more swiftly than the other systems since writing log records at the server is more of a bottleneck for WPL.

Interestingly, the performance of PD-ESM and SD-ESM is nearly identical during T2B due to the fact that the client CPU usage of the two systems was the same. The performance of SD-ESM is a bit worse relative to PD-ESM during T2B because T2B updates more objects on each page that is updated, causing SD-ESM to do more copy and diffing work. In addition, since more updates are performed, the cost of actually doing the updates is more of a factor for SD-ESM, since each update incurs the cost of a function call and other CPU overhead, as described in Section 3. The performance difference between SL-ESM and SD-ESM is approximately 14% in all cases in Figure 6, showing that it is indeed worthwhile here to diff the 64 byte blocks copied by the sub-page diffing scheme. Lastly, Figure 7 shows that the transaction throughput begins to level off after 4 clients for most of the systems, with WPL showing no increase in throughput beyond 3 clients.

Figure 8 shows the response time results for the T2C[5] traversal. The response time for PD-ESM, PD-REDO, and WPL did not change significantly relative to T2B because these systems allow applications to update objects by dereferencing normal virtual memory pointers. The performance of both SD-ESM and SL-ESM was 3.5 seconds slower, independent of the number of clients, due to the higher cost of performing updates in these systems—the overhead for performing the updates themselves is significant during T2C since a total of 1,049,760 additional updates are performed per transaction relative to T2B. Thus, the performance of PD-ESM is between 12% (1 client) and 6% (5 clients) faster than the performance of SD-ESM in Figure 8. The throughput results for T2C are not shown since they were also similar to those for T2B and can be deduced from the response time results of Figure 8. In addition, the number of pages (total pages and log record pages) sent from the client to the server during T2C was the same as during T2B (Figure 9).

5.2. Constrained Cache Results

This section presents results obtained by running the various systems with a restricted amount of memory at each client. As in the previous section, the small database is used, but each client is given only 8 megabytes of memory to use for caching persistent data here. For the systems that do diffing, 7.5Mb was allocated for the client buffer pool and .5Mb for the recovery buffer. This allocation of memory results in a client buffer pool that is large enough to avoid paging. So, for example, the performance of WPL here is the same as in the previous section. However, now there is insufficient space in the recovery buffer to hold all of the data required by the diffing schemes until commit time.

Figures 10 and 11 show the response time and throughput for traversal T2A in the constrained case. SD-ESM has the best performance in Figure 10. SD-ESM is 31% faster than PD-ESM and 40% faster than WPL at five clients. PD-ESM is slower than SD-ESM in this case because it experiences more contention in the recovery buffer. This causes PD-ESM to generate 4 times as many pages of log records on average per transaction as did SD-ESM (see Figure 14). WPL has competitive performance when the number of clients is low, but it has the worst performance when three or more clients are used. The performance of WPL degrades faster than the performance of the other systems, as before,

[4] T2B: update all atomic parts of each composite part.

[5] T2C: update all atomic parts of each composite part four times.

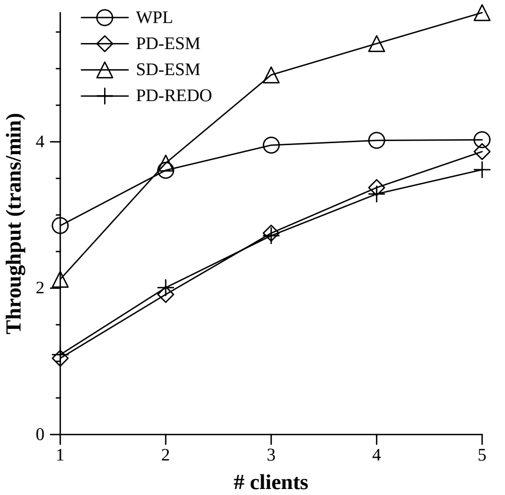

Figure 13. T2B, small, constrained cache.

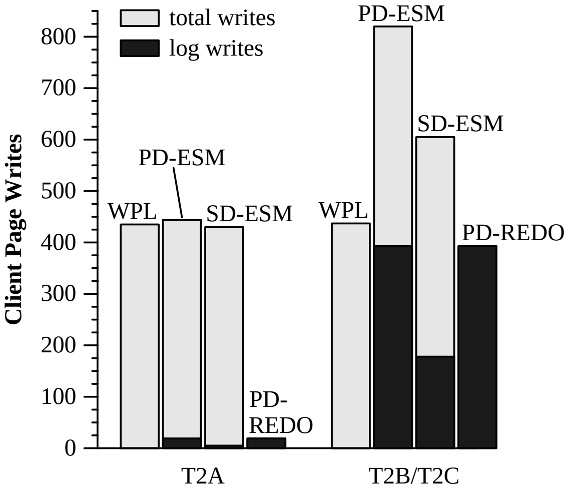

Figure 14. Client Writes, small, constrained cache.

because server performance is more of a limiting factor for WPL. On the other hand, the diffing schemes benefit from their ability to perform more work at the clients, which allows them to scale better.

The performance of PD-REDO appears to be approaching that of SD-ESM as the number of clients increases. PD-REDO scales better than the other systems here because it does not suffer from the overhead of shipping dirty pages back to the server. PD-REDO thus has the second best performance when two or more clients are used and is only 3% slower than SD-ESM at five clients. The throughput results shown in Figure 11 show that transaction throughput increases for all of the systems, except WPL (which becomes saturated), as the number of clients increases. SD-ESM and PD-REDO show the biggest increases in throughput. The throughput of SD-ESM increases by 2.5 times when the number of clients is increased, while the throughput for PD-REDO increases by a factor of 4.2.

We now turn to Figures 12 and 13, which show the response time and throughput, respectively, for traversal T2B. WPL has the best performance at one client, where it is 27% faster than SD-ESM (which is the next best performer). Beyond two clients, however, SD-ESM has the best performance, and at five clients, SD-ESM is 30% faster than WPL. SD-ESM scales better than WPL because it performs most of its recovery work at the clients, while WPL relies more heavily on the server. The page diffing systems, PD-ESM and PD-REDO, have the worst performance in Figure 12. The reason for this can be seen in Figure 14 which shows the number of page writes per transaction for the systems. In Figure 14, PD-ESM produces 2.2 times as many log pages per transaction as does SD-ESM during T2B. In addition, the number of log pages produced by PD-ESM is approaching the number of pages written to the log per transaction by WPL due to the high level of contention in the recovery buffer for PD-ESM.

PD-REDO is slightly faster than PD-ESM when the number of clients is low, but for more than three clients, PD-ESM has better performance. PD-REDO does not scale as well as PD-ESM in Figure 11 because of the CPU overhead of applying log records at the server for PD-REDO. The results for T2C are not shown for this experiment because they were similar to the results for T2B. The only difference in the times for T2C was that SD-ESM was consistently slower by a few seconds relative to its times for T2B.

5.3. Big Database Results

This section contains the results of the experiments that were run using the big database. In these experiments, all of the systems were given 12Mb of memory at each client to use for caching persistent data. For the systems that do diffing, two alternative strategies for partitioning the memory between the client buffer pool and the recovery buffer were explored. Some of the systems were given 8Mb of memory to use as the client buffer pool, and the remaining 4Mb was used for the recovery buffer. Others were allocated 11.5Mb for the buffer pool and just .5Mb for the recovery buffer.

Figure 15 shows the average response time for each system when performing traversal T2A. Surprisingly, WPL has the fastest response time when the number of clients is less than four; however, when four or more clients are used, PD-ESM-1/2 has the best performance. PD-ESM-1/2 is 17% faster than WPL for five clients. Overall, WPL has fast performance in this experiment because it is able to devote all of available memory at the clients to the buffer pool; thereby, decreasing the amount of paging in the system and lessening the burden placed on the server. However, the server log disk becomes a bottleneck for WPL as the number of clients increases and eventually PD-ESM-1/2 performs better. As in previous experiments, the diffing scheme (PD-ESM-1/2) appears to scale better since it makes use of the clients' aggregate processing power to lessen the burden placed on the server's log disk and CPU.

Comparing the performance of PD-ESM-1/2 and PD-ESM-4 in Figure 15 shows the importance of choosing a good division of memory between the client buffer pool and the recovery buffer. PD-ESM-4 has better performance than PD-ESM-1/2 when one client is used, but for multiple clients, when paging between the client and the server becomes relatively more expensive, PD-ESM-1/2 is always faster. Indeed, as Figure 16 shows, the systems that were given smaller client buffers pool begin to thrash significantly when more than two clients are used, while the throughput for PD-ESM-1/2 continues to increase as the number of clients increases (although the increase is very small for more than two clients).

Interestingly, there is little difference in performance between PD-ESM-4 and SD-ESM-4, in Figure 15. This is because there is a significant amount of paging going on in the client buffer pool in Figure 15 (paging in the buffer pool is exactly the same in both systems), and each time

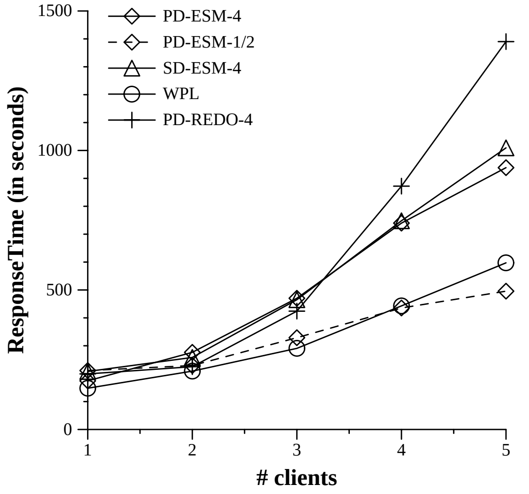

Figure 15. T2A, big database.

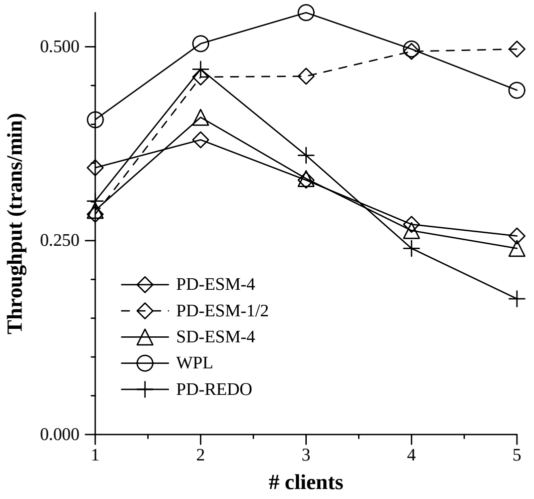

Figure 16. T2A, big database.

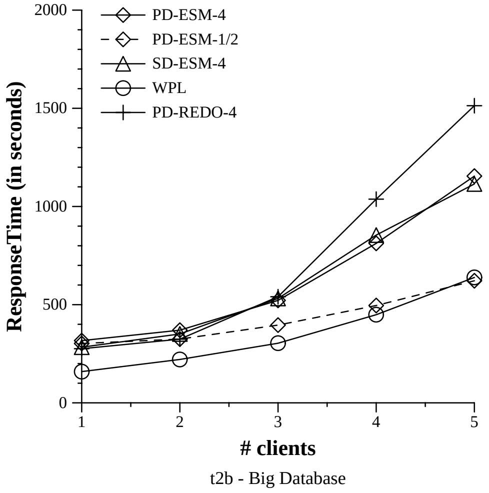

Figure 17. T2B, big database.

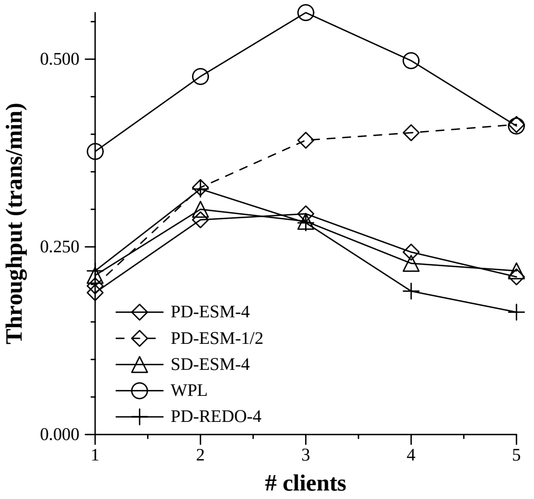

Figure 18. T2B, big database.

a modified page is replaced in the client buffer pool the same number of log records are generated by both PD-ESM-4 and SD-ESM-4. Thus, sub-page diffing does not save nearly as much in terms of the number of log records generated in Figure 15 as it did in the small, constrained experiment (Figure 10). In fact, SD-ESM-4 only generates 10% fewer log records than does PD-ESM-4 in Figure 15, as opposed to generating 75% fewer log records in Figure 10. Finally, we note that although PD-REDO-4 is competitive with PD-ESM-4 when fewer than three clients are used; its performance grades more quickly than the other systems when the number of clients increases beyond three. This is because a larger number of pages must be reread from disk at the server by PD-REDO-4 as the number of clients increases, so that log records describing updates to the pages can be applied.

Comparing the results shown in Figure 15 to those of Figure 17, we see that the relative performance of the systems during T2B (dense updates) and T2A (sparse updates) is similar for the big database. WPL does somewhat better relative to PD-ESM-1/2 during T2B, as T2B updates a much larger number of objects per page, thus causing PD-ESM-1/2 to generate more log records. However, PD-ESM-1/2 scales better than the other systems (including WPL) and the performance of PD-ESM-1/2 appears to surpass that of WPL when five or more clients are used. The performance of SD-ESM-4 and PD-ESM-4 is almost identical in Figure 17 (as it was in Figure 15). Again, this is because SD-ESM-4 generates almost as many log records as PD-ESM-4, and because the savings in client CPU cost provided by SD-ESM-4 for performing less diffing and copying work does not provide any noticeable benefit in terms of performance. The scalability of the REDO algorithm is again the worst overall in Figure 17 as it was in the sparse update case (Figure 15). When five clients are used PD-REDO-4 is 25% slower than PD-ESM-4

Figure 18 shows the throughput for the dense traversal run on the big database. Not surprisingly, the throughput results for the dense traversal are similar to those for the sparse traversal (Figure 16). Figure 18 highlights the fact that the systems that were given a larger client buffer pool (WPL, PD-ESM-1/2) scale better than the systems that were given smaller buffer pools (SD-ESM-4, PD-ESM-4, PD-REDO-4). This is because avoiding paging between the client and the server when using the big database is more important for achieving good performance than avoiding the generation of additional log records as is done by PD-ESM-1/2 and WPL. Even though WPL was given a large (12Mg) client buffer pool it begins to thrash for more than three clients, as the server becomes more of a performance bottleneck.

6. Related Work

This section discusses related work that has been done on the design and implementation of recovery algorithms for client-server database systems. We also compare the study presented here with other studies that have dealt with the issue of recovery performance.

[Frank92] describes the design and implementation of crash recovery in the EXODUS Storage Manager (ESM). In addition, [Frank92] discusses the issues that must be addressed when implementing recovery in a client-server environment. The recovery algorithm described in [Frank92] is based on ARIES [Mohan92], and supports write-ahead-logging and a STEAL/NO FORCE buffer management policy at the server. However, the ESM currently requires that all dirty pages be shipped from a client to the server before a transaction commits. This could be termed a force-to-server-at-commit policy. We note here that while ESM addresses the recovery issues raised by a client server architecture, it does not address the issues discussed in Section 2, i.e. issues that are specific to object-oriented systems and memory-mapped stores.

More recently, [Mohan94] has presented an algorithm termed ARIES/CSA which also extends the basic ARIES recovery algorithm to the client-server environment. ARIES/CSA differs from ESM in that it supports fine-granularity locking which ESM currently does not. ARIES/CSA also supports unconditional undo. Another difference between the two approaches is that clients in ARIES/CSA take their own checkpoints in addition to checkpoints taken by the server, while in ESM checkpoints are only performed at the server. In principle the recovery techniques described in Section 3 that use ESM could also be used in conjunction with ARIES/CSA. However, it is not clear how features such as fine-granularity locking that are supported by ARIES/CSA would be used by a memory-mapped storage system such as QuickStore since the memory-mapped approach is inherently page-based.

A related study on recovery performance appears in [Hoski93]. The study presented in [Hoski93] differs from the study presented here, in that it is only concerned with alternative methods for detecting and recording the occurrence of updates. We consider these issues as well, and also examine different strategies for processing log records in a client-server environment, i.e. ARIES-based schemes, whole-page logging, and redo-at-server. An interesting note about the study presented here is that our results differ substantially from the results presented in [Hoski93]. Some of the variation in the results is undoubtedly due to architectural differences between the systems examined in the two studies. For example, in [Hoski93] the transaction log is located at the client machine instead of at the server as in ESM. The advantage of locating the log at the server as ESM does is that it increases system availability, since if a client crashes, the server can continue processing the requests of other clients. Placing the log at the server impacts performance because log records must be sent from clients to the server (usually over a network) before they are written to disk. Thus, differences in alternative techniques for generating log records will be smaller in a system in which the log is located at the server than in a system where log records are written to disk at the client.

Other reasons for differences in the results have to do with implementation details. For example, both [Hoski93] and our study examine the tradeoffs involved in detecting the occurrence of updates in software versus using virtual memory hardware support. However, [Hoski93] uses a *copy* architecture in which objects are copied one-at-a-time as they are accessed from the client buffer pool into a separate area in memory called the *object cache*. The hardware-based recovery scheme used in [Hoski93] requires that the virtual memory page in the object cache that will hold an object be unprotected and then reprotected each time an object is copied into the page—producing a substantial amount of overhead (up to 100%), even for read-only transactions. Under the approach used in QuickStore (in-place access, page-at-a-time swizzling) a page's protection is only manipulated once, when the first object on the page is updated. Thus, our results show that the hardware-based detection scheme does much better than do the results presented in [Hoski93]. In particular, the scheme used in QuickStore does not impact the performance of read-only transactions.

[Hoski93] also examines several schemes (termed *card marking*) that are similar to the sub-page approach examined here. The results presented in [Hoski93] show that the size of the sub-page region used for recovery has a great impact on performance. The difference in performance reported in [Hoski93] is due to the fact that using a small sub-page unit for recovery can reduce diffing costs when updates are sparse. The results presented in this study, on the other hand, show that the size of the sub-page blocks is important, not because of savings in CPU costs when the block size is small, but because smaller block sizes can result in the generation of fewer log records when space in the recovery buffer is tight. [Hoski93] does not consider this case. We believe the differences in the results are due to the fact that we use an in-place scheme while [Hoski93] uses a copy approach, and to the cost for supporting concurrency control (the system examined in [Hoski93] is single user system that do not support concurrency). The study presented here also differs from [Hoski93] in that we examine the scalability of the various recovery schemes as the number of clients accessing the database increases. We also examine the performance of the different recovery techniques when a large database is used and a significant amount of paging (object replacement) is taking place in the system. [Hoski93] does not consider these issues. We also note the performance study contained in [White92] also contains some results concerning recovery performance. However, the systems compared in [White92] (ObjectStore and E/EXODUS) were not built upon the same underlying storage manager, so it is not possible to accurately measure the performance differences due to recovery in [White92].

Although several OODBMSs are commercially available, very little has been published concerning the recovery algorithms they use. The O_2 system [Deux91] also uses an ARIES-based approach to support recovery. O_2 differs from ESM and ARIES/CSA in that it uses shadowing to avoid undo. The most popular commercial OODBMS is ObjectStore [Lamb91] which like, QuickStore, uses a memory-mapping scheme to give application programs access to persistent data. ObjectStore currently uses whole-page logging to support recovery. The basic idea of this approach is that dirty pages are shipped from a client to the server and written to the log before a transaction is allowed to commit. This differs from the ARIES-based schemes mentioned above which only require that the log records generated by updates be written to disk at commit time. [Wilso92] describes the Texas storage manager which also uses a memory-mapping scheme. [Wilso92] was the first to propose the use of differencing to detect updates of persistent data. Finally, we note that the whole-page logging approach to recovery was first described in [Elhar84] which presents the design of the *database cache*.

7. Conclusions

This paper has presented an in-depth comparison of the performance of several different approaches to implementing recovery in QuickStore, a memory-mapped storage manager based on a client-server, page-shipping architecture. Each of the recovery algorithms was designed to meet the unique performance requirements faced by a system like QuickStore. The results of the performance study show that using diffing to generate log records for updates at clients is generally superior in terms of performance to whole page logging. The diffing approach is better because it takes advantage of the aggregate CPU power available at the clients to lessen the overall burden placed on the server to support recovery.

This provides much better scalability, as it prevents the server from becoming a performance bottleneck as quickly when the number of clients accessing the database increases.

The study also compared the performance of two different underlying recovery schemes upon which the diffing algorithms were based. These included the recovery algorithm used by the EXODUS Storage Manager and a simplified scheme termed redo-at-server (REDO). While the REDO approach provided significant performance benefits in some cases—when using a small database, while producing only a moderate number of log records per transaction—it failed to perform well when the database was bigger than the server buffer pool, and when the volume of log records sent to the server per transaction was high. The results presented in the study show that REDO can suffer from both disk and CPU bottlenecks at the server. System builders will have to decide whether the simplifications in system design and coding are worth the poor scalability of REDO in certain situations.

Finally, the study compared the performance of page-based and sub-page diffing techniques. Surprisingly, the sub-page diffing techniques provided very little advantage in terms of performance over the page-based approach. This is apparently due to the fact that diffing is a relatively inexpensive operation compared to the other costs involved in the system, such as network and disk access costs. The sub-page diffing approach did pay off in one situation, i.e. when the amount of memory that could be devoted to recovery was very low. System designers will have to decide whether this situation is likely to arise often enough in practice to justify using sub-page diffing. In addition, it was shown that sub-page diffing can have worse performance than page-diffing when updates are performed repeatedly. This fact must be weighed against the mild advantages of the technique when deciding on an implementation strategy. Finally, the results showed that diffing is even worthwhile when a sub-page granularity is used for recovery. Systems that used a sub-page granularity, but which did not use diffing always had comparable or worse performance than the systems that used diffing.

In the future, we would like to explore improvements to the recovery schemes based on differencing to see if the diffing approach can be enhanced to better adapt to dynamic workload changes. One class of techniques that we would like to explore involves dynamically varying the amount of memory allocated to the buffer pool and the recovery buffer of a client during and across transactions.

References

[Carey89] M. Carey et al., "Storage Management for Objects in EXODUS," in *Object-Oriented Concepts, Databases, and Applications*, W. Kim and F. Lochovsky, eds., Addison-Wesley, 1989.

[Carey93] M. Carey, D. DeWitt, J. Naughton, "The OO7 Benchmark", *Proc. ACM SIGMOD Conf.*, Washington, DC, May 1993.

[Carey94] M. Carey, D. DeWitt, M. Franklin, N. Hall, M. McAuliffe, J. Naughton, D. Schuh, M. Solomon, C. Tan, O. Tsatalos, S. White, M. Zwilling, "Shoring Up Persistent Applications", *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May 1994.

[DeWitt90] D. DeWitt, P. Futtersack, D. Maier, F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proc. 16th VLDB Conf.*, Brisbane, Australia, August, 1990.

[Deux91] O. Deux et al., "The O2 System", *Comm. of the ACM*, Vol. 34, No. 10, October 1991

[Elhar84] K. Elhardt, R. Bayer, "A Database Cache for High Performance and Fast Restart in Database Systems", *ACM Trans. on Database Sys.*, 9(4):503-525, December 1984.

[Frank92] M. Franklin et al., "Crash Recovery in Client-Server EXODUS", *Proc. ACM SIGMOD Conf.*, San Diego, California, 1992.

[Frank93] M. Franklin, "Caching and Memory Management in Client-Server Database Systems", Ph.D. thesis, University of Wisconsin-Madison, tech. report #1168, July 1993.

[Haerd83] T. Haerder, A. Reuter, "Principles of Transaction Oriented Database Recovery - A Taxonomy", *Computing Surveys*, Vol. 6, No. 1, February 1988.

[Hoski93] A. Hosking, E. Brown, J. Moss, "Update Logging in Persistent Programming Languages: A Comparative Performance Evaluation", *Proc. 19th VLDB Conf.*, Dublin, Ireland, 1993.

[Lamb91] C. Lamb, G. Landis, J. Orenstein, D. Weinreb, "The ObjectStore Database System", *Comm. of the ACM*, Vol. 34, No. 10, October 1991

[Mohan92] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, P. Schwartz, "ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging", *ACM Trans. on Database Sys.*, Vol. 17, No. 1, March 1992.

[Mohan94] C. Mohan, I. Narang, "ARIES/CSA: A Method for Database Recovery in Client-Server Architectures", *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, 1994.

[Moss92] J. Eliot B. Moss, "Working with Persistent Objects: To Swizzle or Not to Swizzle", *IEEE Trans. on Software Eng.*, 18(8):657-673, August 1992.

[Rich93] J. Richardson, M. Carey, and D. Schuh, "The Design of the E Programming Language", *ACM Trans. on Prog. Lang. and Sys.*, Vol. 15, No. 3, July 1993.

[Schuh90] D. Schuh, M. Carey, and D. Dewitt, Persistence in E Revisited---Implementation Experiences, in *Implementing Persistent Object Bases Principles and Practice, Proc. 4th Int'l. Workshop on Pers. Obj. Sys.*, Martha's Vineyard, MA, Sept. 1990.

[White92] S. White and D. DeWitt, "A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies", in *Proc. 18th VLDB Conf.*, Vancouver, British Columbia, August 1992.

[White94] S. White, D. DeWitt, "QuickStore: A High Performance Mapped Object Store", *Proc. ACM SIGMOD Conf.*, Minneapolis, MN, May 1994.

[Wilso92] P. Wilson, S. Kakkad, "Pointer Swizzling at Page Fault Time: Efficiently and Compatibly Supporting Huge Address Spaces on Standard Hardware", *Proc. Int'l. Workshop on Obj. Orientation in Operating Sys., Paris, France, Sept. 1992.*

Acknowledgements

We would like to thank all of the people at Wisconsin who helped to build the EXODUS Storage Manager. Special thanks go to Mike Zwilling who patiently answered the many questions we had concerning EXODUS while working on this paper. C. K Tan provided some valuable advice on how to implement the redo-at-server algorithm.