# Quickstore: A High Performance Mapped Object Store

Seth J. White
David J. DeWitt

Technical Report #1241

September 1994

# QuickStore: A High Performance Mapped Object Store

Seth J. White
Computer Sciences Department
University of Wisconsin
Madison, WI 53706
white@cs.wisc.edu

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

**ABSTRACT**

This paper presents, QuickStore, a memory-mapped storage system for persistent C++ built on top of the EXODUS Storage Manager. QuickStore provides fast access to in-memory objects by allowing application programs to access objects via normal virtual memory pointers. The paper also presents the results of a detailed performance study using the OO7 benchmark. The study compares the performance of QuickStore with the latest implementation of the E programming language. These systems (QuickStore and E) exemplify the two basic approaches (hardware and software) that have been used to implement persistence in object-oriented database systems. In addition, both systems use the same underlying storage manager and compiler allowing us to make a truly apples-to-apples comparison of the hardware and software techniques.

## 1. Introduction

This paper presents, QuickStore, a memory-mapped storage system for persistent C++ built on top of the EXODUS Storage Manager (ESM) [Carey89]. QuickStore uses standard virtual memory hardware to trigger the transfer of persistent data from secondary storage into main memory [Wilso90]. The advantage of this approach is that access to in-memory persistent objects is just as efficient as access to transient objects, i.e. application programs access objects by dereferencing normal virtual memory pointers, with no overhead for software residency checks as in [Moss92, Schuh90, White92].

QuickStore is implemented as a C++ class library that can be linked with an application, requiring no special compiler support. Instead, QuickStore uses a modified version of the GNU debugger (gdb) to obtain information describing the physical layout of persistent objects. QuickStore uses the information provided by gdb to automatically maintain database schemas. The memory-mapped architecture of QuickStore supports "persistence orthogonal to type", so that both transient and persistent objects can be manipulated using the same compiled code. Because QuickStore uses ESM to store persistent data on disk, it features a client-server architecture with full support for transactions (concurrency control and recovery), indices, and large objects. QuickStore places no additional limits on the size of a database, and the amount of data that can be accessed in the context of any single transaction is limited only by the size of virtual memory.

The paper also presents the results of a detailed performance study, in which we use the OO7 benchmark [Carey93] to compare the performance of QuickStore with the latest implementation of E [Rich93], a persistent programming language developed at Wisconsin that is also based on C++. The comparison between QuickStore and E is interesting because each of the systems takes a radically different approach toward implementing persistence. QuickStore employs a hardware faulting scheme that relies on virtual memory support (as mentioned above), while E uses an interpretive approach that is implemented in software.

These systems (QuickStore and E) exemplify the two basic approaches (hardware and software) that have been used to implement persistence in object-oriented database systems. Moreover, both QuickStore and E use the same underlying storage manager (ESM) and compiler. This allows us to make a truly apples-to-apples comparison of the hardware and software swizzling schemes, something which has not been done previously.

The remainder of the paper is organized as follows. Section 2 discusses related work on hardware and software based pointer swizzling schemes and points out how the performance results presented in this paper differ from previous studies. Section 3 describes the design of QuickStore. Section 4 presents our experimental methodology and Section 5 presents the results of the performance study. Section 6 contains some conclusions and proposals for future work.

A version of this paper appeared in Proceedings of the 1994 ACM-SIGMOD Conference on the Management of Data, Minneapolis, MN, May 1994. 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.

## 2. Related Work

A detailed proposal advocating the use of virtual memory techniques to trigger the transfer of persistent objects from disk to main memory, first appeared in [Wilso90]. The basic approach described in [Wilso90] is termed "pointer swizzling at page fault time" since under this scheme all pointers on a page are converted from their disk format to normal virtual memory pointers (i.e. swizzled) by a page-fault handling routine before an application is given access to a newly resident page. In addition, pages of virtual memory are allocated for non-resident pages one step ahead of their actual use and access protected, so that references to these pages will cause a page-fault to be signaled. The technique described in [Wilso90] allows programs to access persistent objects by dereferencing standard virtual memory pointers, eliminating the need for software residency checks.

The basic ideas presented in [Wilso90] were, at the same time, independently used by the designers of ObjectStore [Objec90, Lamb91], a commercial OODBMS product from Object Design, Inc. The implementation of ObjectStore, outlined briefly in [Objec90], differs in some interesting ways from the scheme described in [Wilso90]; most notably in the way that pointer swizzling is implemented, and in how pointers are represented on disk.

Under the approach outlined in [Objec90], pointers between persistent objects are stored on disk as virtual memory pointers instead of being stored in a different disk format as in [Wilso90]. In other words, pointer fields in objects simply contain the value that they last were assigned when the page was resident in main memory in ObjectStore. When a page containing persistent objects is first referenced by an application program, ObjectStore attempts to assign the page to the same virtual address as when the page was last memory resident. If all of the pages accessed by an application can be assigned to their previous locations in memory, then the pointers contained on the pages can retain their previous values, and need not be "swizzled", i.e. changed to reflect some new assignment of pages to memory locations, as part of the faulting process. If any page cannot be assigned to its previous address (because of a conflict with another page), then pointers that reference objects on the page will need to be altered (i.e swizzled) to reflect the new location of the page.

This scheme requires that the system maintain some additional information describing the previous assignment of disk pages to virtual memory addresses. The hope is that processing this information will be less expensive on average, than swizzling the pointers on pages that are faulted into memory by the application program. We note here that QuickStore is similar to ObjectStore in that QuickStore also stores pointers on disk as virtual memory pointers. Section 3 contains a detailed discussion of the implementation of QuickStore.

The Texas [Singh92] and Cricket [Shek90] storage systems also use virtual memory techniques to implement persistence. Texas stores pointers on disk as 8-byte file offsets, and swizzles pointers to virtual addresses as described in [Wilso90] at fault time. Currently, all data is stored in a single file (implemented on a raw *Unix* disk partition) in Texas

[Singh92]. Although, QuickStore and Texas are different in their implementation details, there are some similarities between the two systems. For example, both systems are implemented as C++ libraries that add persistence to C++ programs without the need for compiler support. Both systems also support the notion of "persistence orthogonal to type". This allows the same compiled code to manipulate both transient and persistent objects. Both systems also allow the database size to be bigger than the size of virtual memory.

Texas, however, is currently a single user, single processor system while QuickStore, since it is built on top of client-server EXODUS, features a client-server architecture with full transaction support including concurrency control, recovery, and support for distributed transactions. QuickStore is also different in that it manipulates objects directly in the ESM client buffer pool, while Texas copies objects into a separate heap area allocated in virtual memory. This limits the amount of data that can currently be accessed during a single transaction by Texas to the size of the disk swap area backing the application process. QuickStore also manages paging in the ESM client buffer pool explicitly, while Texas simply allows pages to be swapped to disk by the virtual memory subsystem when the process size exceeds the size of physical memory. Cricket, on the other hand, uses the Mach external pager facility to map persistent data into an application's address space (see [Shek90] for details).

A very recent related system is Dali [Jagad94] which is designed to be a main memory storage manager. Dali uses memory mapping techniques, but since it is specifically designed to handle main memory databases, it differs substantially from QuickStore. For example, Dali itself performs no pointer swizzling. In Dali, the database is partitioned into units called *database files*, each of which is mapped contiguously into a process's address space. Database pointers in Dali contain a database file identifier and an offset. Dereferencing a *database pointer* usually involves indexing into a fixed size array of open *database files* and adding the starting address of the *database file* (contained in the array) to the offset contained in the pointer. If a large number of database files are being used (which is not expected to happen often) then a search in an in-memory tree structure is required to determine the starting address of the database file.

We next discuss previous performance studies of pointer swizzling and object faulting techniques, and point out how the study presented here differs from them. [Moss92] contains a study of several software swizzling techniques and examines various issues relevant to pointer swizzling. Among these are whether swizzling has better performance than simply using object identifiers to locate objects, and whether objects should be manipulated in the buffer pool of the underlying storage manager, or copied out into a separate area of memory before swizzling takes place. [Moss92] also looks at lazy vs. eager swizzling. Eager swizzling involves prefetching the entire collection of objects into memory so that all pointers can be swizzled, while lazy swizzling swizzles pointers incrementally as objects are accessed and faulted into memory by the application program. We do not consider copy swizzling approaches since [White92] showed that they do not perform well when the database size is larger than physical memory. The study presented here also differs from [Moss92] in that we allow pages of objects to be replaced in the buffer pool, while [Moss92] only considers small data sets where no paging occurs. The systems we examine also include concurrency control and recovery, while those examined in [Moss92] did not.

[Hoski93a] examines the performance of several object faulting schemes in the context of a persistent Smalltalk implementation. [Hoski93a] includes one scheme that uses virtual memory techniques to detect accesses to non-resident objects. The approach described in [Hoski93a] allocates fault-blocks, special objects that stand in for non-resident objects, in protected pages. When the application tries to access an object through its corresponding fault block, an access violation is signaled. The results presented in [Hoski93a] show this scheme to have very poor performance. It is not clear, however, whether this is due to the overhead associated with using virtual memory or is the result of extra work that must be performed during each object fault to locate and eliminate any outstanding pointers to the fault block that caused the fault. This work involves examining the pointer fields of all transient and persistent objects that contain pointers to the fault block. Finally, we note that the effects of page replacement in the buffer pool and updates are also not considered in [Hoski93a].

In [White92] the performance of several implementations of the E language [Rich93, Schuh90] and ObjectStore [Objec90, Lamb91], a commercial OODBMS, are compared. The results presented in [White92] were inconclusive, however, in providing a true comparison of software and hardware-based schemes since the underlying storage managers used by the systems were different and because the systems used different compilers. In the study presented in this paper, all of the systems use the same underlying storage manager and compiler, so any differences in performance are due to the swizzling and faulting technique that was used.

One additional difference between the systems compared in [White92] and those examined here, is that the systems included in the current study are much less restrictive in terms of the amount of data that can be accessed during a transaction, and all systems manage paging of persistent data explicitly. This differs from the approach used by EPVM 2.0 in [White92], which limited the amount of data that could be accessed during a transaction to the size of the disk swap area backing the process, and which allowed objects to be swapped to disk by the virtual memory subsystem when the size of the process exceeded the size of physical memory.

Finally, we note that [Hoski93b] examines the performance of several alternative methods for detecting and recording the occurrence of updates in a persistent programming language, i.e. Smalltalk, that supports recovery from system failures. The techniques studied in [Hoski93b] use differencing to generate log records, as does QuickStore. The study presented here differs from [Hoski93b] in that, the goal of [Hoski93b] is to compare different techniques for detecting and recording updates in the context of a single underlying language implementation, while the goal of our study is to compare the performance of software and hardware-based pointer swizzling. However, [Hoski93b] does examine one scheme that uses virtual memory page faults to detect the occurrence of updates, so we briefly compare QuickStore to the approach examined in [Hoski93b]. [Hoski93b] uses a copy swizzling approach in which objects are copied out the buffer pool of the underlying object manager and into a separate *object cache* located in memory, before they can be accessed by the application program. The hardware-based detection scheme used in [Hoski93b] 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. The technique used in QuickStore doesn't impact the performance of read-only transactions, and we expect that detecting updates is much cheaper in QuickStore than in the scheme studied in [Hoski93b].

## 3. QuickStore Design Concepts

As mentioned in Section 1, QuickStore uses ESM to store persistent objects on disk. ESM features a page-shipping [DeWitt90] architecture, in which objects are transferred from the server to the client a page-at-a-time. Once a page of objects has been read into the buffer pool of the ESM client, applications that use QuickStore access objects on the page directly in the ESM client buffer pool, by dereferencing normal virtual memory pointers. We note that objects are always accessed in the context of a transaction in QuickStore.

### 3.1. Overview of the Memory-Mapped Architecture

This section describes the memory-mapping scheme used by QuickStore to give application programs access to persistent objects. To understand the approach, it is useful to view the virtual address space of the application process as being divided into a contiguous sequence of *frames* of equal length. In our case, these frames are 8 K-bytes in size, the same size as pages on disk. The ESM client buffer pool can also be viewed as a (much smaller) sequence of 8 K-byte frames. To coordinate access to persistent objects, QuickStore maintains a physical mapping from virtual memory frames to frames in the buffer pool. This physical mapping is **dynamic**, since paging in the buffer pool requires that the same frame of virtual memory be mapped to different frames in the buffer pool at different points in time. The mapping can also be viewed as a logical mapping from virtual memory frames to disk pages. When viewed this way, the mapping is **static** since the same virtual frame is always associated with the same disk page during the course of a transaction.

Figure 1 illustrates this mapping scheme in more detail. The buffer pool shown in Figure 1 contains 7 frames (labeled from 1 to 7). Virtual memory frames are denoted using upper-case letters, while disk pages are specified in lower-case. In the discussion that follows, we sometimes refer to the virtual memory frame beginning at address *A* as frame *A*.

Before a page is read from disk by QuickStore, the virtual memory frame corresponding to the page is selected and access protected. When the application first attempts to access an object on the page by dereferencing a pointer into its frame, a page-fault is signaled and a fault handling routine that is part of the QuickStore runtime system is invoked. This

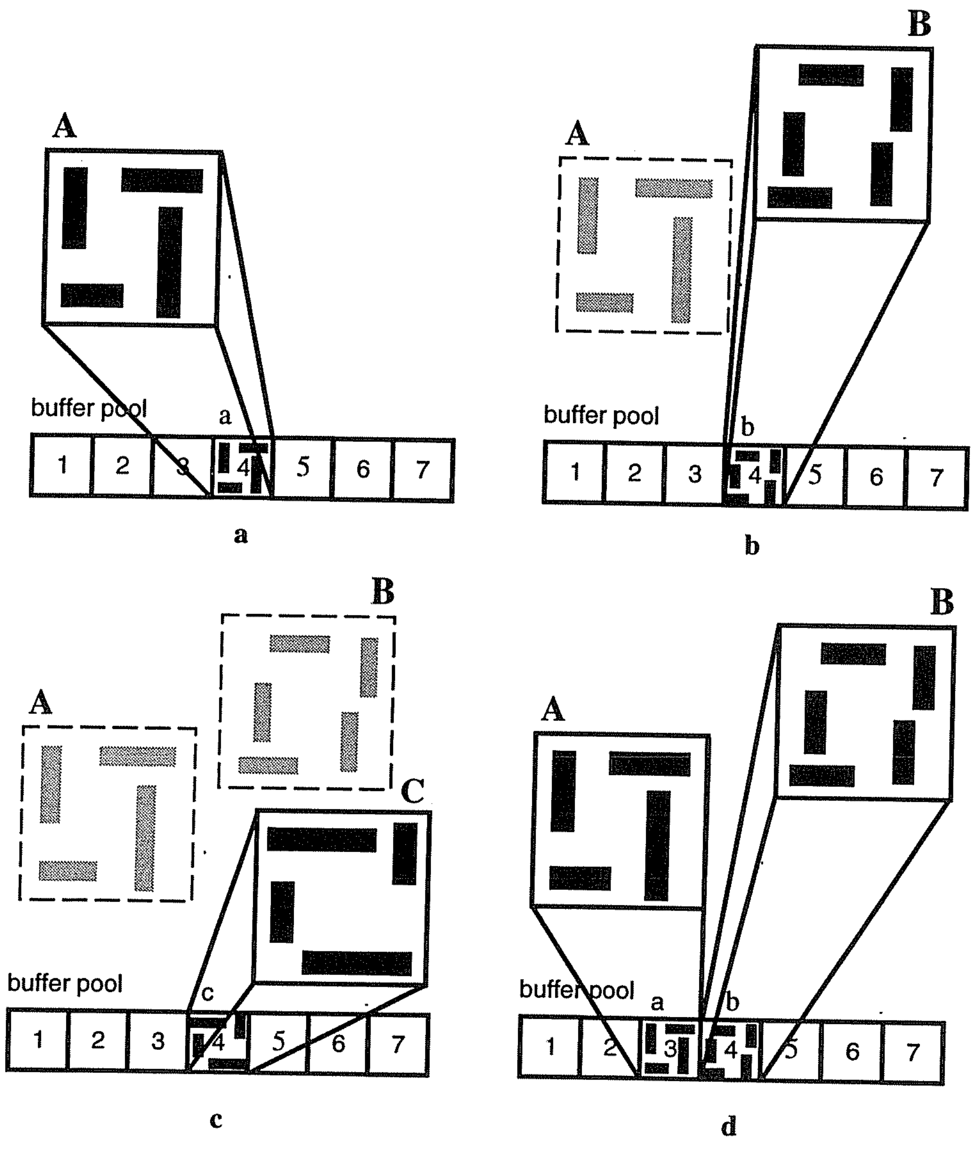


Figure 1. Mapping virtual frames into the buffer pool.

fault handling routine is responsible for reading the page from disk, updating various data structures, and enabling access permission on the virtual frame that caused the fault so that execution of the program can resume. For example, in Figure 1a page *a* has been read from disk into frame *4* of the buffer pool. Page *a* is "mapped" to virtual address *A*. Read access has been enabled on frame *A*, so that the application can read the objects contained on page *a*. We note that once the mapping from virtual address *A* to page *a* has been established, the application can access objects on page *a* by dereferencing pointers to frame *A* at any time. Thus, the mapping from *A* to *a* must remain valid until the end of the current transaction (or longer, if requested) in order to preserve the semantics of any pointers that the application may have to objects on page *a*. (This is not the case for the mapping of virtual frame *A* to buffer frame *4*, however, as we will see in a moment.)

If the objects on page *a* contain pointers to objects on other non-resident pages, then virtual frames are assigned to these pages when page *a* is faulted into memory (if they haven't been already). The mechanism for assigning virtual frames to disk pages is covered in detail in Section 3.4, which discusses pointer swizzling. A frame for a non-resident page remains access protected until the program attempts to deference a pointer into the frame, which then causes a page fault that results in the page being read into the buffer pool. Figure 1 doesn't explicitly show any of the frames being referenced by pointers on page *a* since they are not important to the current discussion.

When the buffer pool becomes full, paging will occur and page *a* may be selected for replacement by the buffer manager. (See Section 3.5 which discusses buffer pool management for details.) This is what has happened in Figure 1b. Here, page *b* has been read from disk into frame *4* of the buffer pool, replacing page *a*. Page *b* has been mapped to virtual address *B* and read access on *B* has been enabled. Note that since we assume that the buffer pool is full in Figure 1b, additional virtual frames (not shown) will also have been mapped to the remaining 6 frames in the buffer pool other than frame *4*. If the application continues to access additional pages of objects in the database, then the situation shown in Figure 1c may result. In Figure 1c, page *c* has been read into memory and replaced page *b* in frame *4* of the buffer pool. Page *c* has been mapped to virtual frame *C* and read access on *C* has been enabled. This illustrates that, in general, any number of virtual frames may be associated with a particular frame of the buffer pool over the course of a transaction.

At this point, the reader may be wondering, what would happen in Figure 1b if the application attempted to dereference pointers into virtual frame *A* after page *a* has been replaced in the buffer pool by page *b*. Won't these pointers refer to data on page *b*? This problem is avoided by disabling read access on frame *A* when page *a* is not in memory. If the application again dereferences pointers into frame *A*, a page-fault will be signaled and the fault handling routine will be invoked. The fault handling routine will call ESM to reread page *a*, map virtual frame *A* to the frame in the buffer pool that now contains *a*, and enable read permission on frame *A* once again.

To illustrate this, Figure 1d shows what might result if page *a* were immediately referenced after being replaced in Figure 1b. In this case, *a* has been reread by ESM into frame *3* in the buffer pool and frame *3* has been mapped to virtual memory address *A*. This further illustrates the dynamic nature of the mapping from virtual memory frames to frames in the buffer pool as virtual frame *A* is mapped to buffer frame *4* in Figure 1a and then remapped to buffer frame *3* in Figure 1d. Note, however, that the mapping between virtual frames and disk pages is static—virtual frame *A* is always mapped to disk page *a*.

## 3.2. Implementation Details

QuickStore uses the UNIX *mmap* system call to implement the physical mapping from virtual memory frames to frames in the ESM client buffer pool and to control virtual frames' access protections. It was necessary to modify the ESM client software slightly in order to accommodate the use of *mmap* since *mmap* really just associates virtual memory addresses with offsets in a file, while ESM normally calls the UNIX function *malloc* to allocate space in memory for its client buffer pool. To make ESM and *mmap* work together, the buffer pool allocation code was changed so that it would first open a file (and resize it if necessary) equal in size to the size of the client buffer pool. The buffer allocation code then calls *mmap* to associate a range of virtual memory with the entire file. The rest of the ESM client software uses this range of memory to access the buffer pool just as though the memory had been allocated using *malloc.*

The important thing to note is that the file serves as backing store for the buffer pool. Swap space and actual physical memory are never allocated for the virtual frames that are mapped into the file by *mmap*, so mapping a huge amount of virtual memory into the buffer pool doesn't affect the size of the process, although it may increase the size of page tables maintained by the operating system. One should also note that the contiguous range of addresses used by the ESM client to access the buffer pool is different from the 8 K-byte ranges of addresses that the application program uses to access pages in the buffer pool. The former is simply used to integrate an already existing storage manager (ESM) with the memory mapped approach and would not, in general, be required by a memory mapped implementation.

We should point out that using *mmap* in the particular way that we did, caused some minor performance problems in the implementation. Because the workstation used as the client machine in the benchmark experiments (a Sun ELC) had a virtually mapped CPU cache, accessing the same page of physical memory in the buffer pool via different virtual address ranges caused the CPU cache to be flushed whenever the process switched between the address ranges. This increased the number of *min faults*, which are virtual memory page faults that do not require I/O (in Unix terminology), experienced by the application. We note the effects of this phenomena when discussing the performance results in Section 5.

## 3.3. In-Memory Data Structures

QuickStore maintains an in-memory table that keeps track of the current mapping from virtual memory frames to disk pages. At a given point in time, the table contains an entry for every page that has been faulted into memory, plus entries for any additional pages that are referenced by pointers on these pages. We refer to a page for which there is a table entry as being "in the current mapping". In essence, any page containing persistent data that the application program can deference a pointer to must be in the current mapping. Entries in the table are called *page descriptors* and are 60 bytes long. Figure 2 shows the format of a page descriptor. We note that disk pages themselves come in two types: pages that contain sets of objects that are smaller than a disk page which are called *small object pages*, and pages that contain individual pages of multi-page objects, which are called *large object pages*. Table entries for small object pages and large object pages differ in some respects, so they are discussed separately.

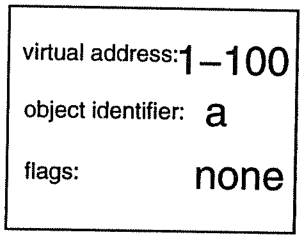
A page descriptor for a small object page contains the range of virtual addresses associated with the page, the physical address of the page on disk, and a pointer to the page when it is pinned in the buffer pool. The physical address of the page, in our implementation, is the OID of a special meta-object (24 bytes) located on each small object page. Page descriptors also contain other fields such as flags that indicate what types of access are currently allowed on the frame associated with the page (read, write, and none), whether an exclusive page lock has been obtained, and whether or not the page has previously been read into memory during the current transaction. This last flag is useful since it is not necessary to do any swizzling work for a page when it is reread during a transaction; the pointers on such pages are guaranteed to be valid. The page descriptor also contains a heap pointer that is used for recovery purposes.

The scheme used for large object pages is somewhat more complicated than the scheme for small object pages. The virtual memory frames associated with a multi-page object must be contiguous, so they are reserved all at once. To avoid maintaining individual table entries for every page of a multi-page object, multi-page objects that have not been accessed—but which are in the mapping—are represented by a single entry in the mapping table. The range of virtual addresses in this entry is the entire range of contiguous addresses associated with the object, and the physical address field contains the OID of the object. When the first page of a multi-page object is accessed by the application program, the table entry is split so that there is one entry in the table for the page that has been accessed, and an entry for each contiguous sub-sequence of unaccessed pages. Table entries for sub-sequences of unaccessed pages of multi-page objects are split in turn when one of the pages contained in the sub-sequence is accessed.

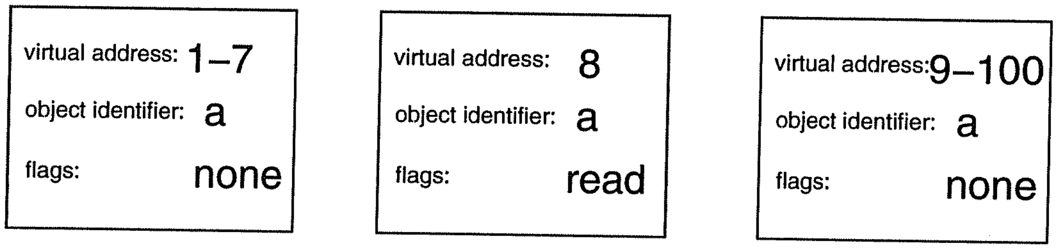
Figure 3 illustrates the splitting process for a large object descriptor. Figure 3a shows the descriptor for a large object that has not yet been accessed. The object is 100 pages long and has been mapped to virtual memory frames 1 through 100.

| |
|---|
| virtual address range |
| physical disk address |
| flags |
| pointer to page in buffer pool |
| recovery heap pointer |
| other in–memory pointers |

Figure 2. Format of a page descriptor.



a



b

Figure 3. Splitting a large object descriptor.

The OID of the object is denoted by the letter *a*. Figure 3b shows what happens when the eighth page of the object is accessed. In Figure 3b a page descriptor has been allocated for the recently accessed page. The virtual address range in the descriptor records the fact that the descriptor is now only associated with virtual memory frame *8* and that read access on frame *8* has been enabled. The descriptor also contains a pointer (not shown) to the copy of the page that is cached in the buffer pool. Figure 3b shows that two additional page descriptors are used to represent the remaining pages of object *a*. One descriptor represents the pages in object *a* that precede page *8*, while the other represents the pages that follow page *8*. We note that the offset (also not shown) of the first page of the object that is represented by a descriptor is stored in the page descriptor as well, so that the descriptor can be associated with the correct pages of the object.

The table organizes page descriptors according to the range of virtual memory addresses that they contain using a height balanced binary tree. One reason for using a binary tree is that it makes the splitting operation associated with large objects efficient. It is also helpful to keep the ranges of addresses currently allocated to persistent data ordered. For example, our current scheme for allocating virtual frames to disk pages uses a global counter (stored on disk) that is incremented by the frame size each time that a frame is allocated to a disk page. This counter needs to be persistent so that successive runs of a program don't reuse the same virtual memory addresses unnecessarily when allocating new objects. If the database becomes bigger than the size of virtual memory then this counter will wrap around and it may become necessary to scan the in-memory binary tree in order to find a virtual frame that is currently not in use.

Page descriptors are also hashed based on their physical address (OID) and inserted in a hash table. (For large objects only the entry containing the first page of the object is inserted in the hash table.) The hash table implements a reverse mapping from physical disk address to virtual memory address. The hash table is used by the fault handling routine as part of the pointer swizzling process (see the next section for details).

## 3.4. Pointer Swizzling in QuickStore

QuickStore stores pointers on disk as virtual memory addresses in exactly the same format that they have when they are in memory. Figure 4 shows the format of a pointer in QuickStore. The high order bits contained in a pointer can be viewed as identifying a virtual memory frame, while the low order bits identify an offset into the frame where the actual object referenced by the pointer is located. (Objects are not allowed to move within pages, so the offset identifies a unique object or portion of an object.) Since virtual memory pointers are only meaningful in the context of an individual process, the system maintains some additional meta-data that associates pointers on disk with the objects that they reference. We first describe how this meta-data is stored in QuickStore, and then briefly touch on some possible alternative implementation strategies.

QuickStore associates meta-data with individual disk pages. In the case of small object pages, each page contains a direct pointer (OID) to a *mapping object* containing the meta-data for the page. (Actually, the pointer is contained in the meta-object located on the page.) The term *mapping object* is used since the object records the mapping between virtual frames referenced by pointers on the page and disk pages that was in effect when the page was last memory resident. Mapping objects are essentially just arrays of <virtual address range, disk address> pairs. Mapping information is stored separately instead of on the disk pages containing objects themselves because the space required to store the mapping information for a page can vary over time. For example, if the pointers on a page are updated, the number of frames referenced by pointers on the page may change, thus changing the number of entries in the mapping object. Multi-page objects are implemented similarly to small object pages, except that there is an array of meta-objects appended to the end of the large

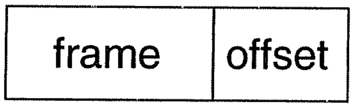


Figure 4. Format of a pointer in QuickStore.

object containing one meta-object for each page of the large object.

Each meta-object also contains a pointer (OID) to a bitmap object that records the locations of pointers on the page so that they can be swizzled. QuickStore uses a modified version of *gdb* to get the type information for objects that is used to maintain the bitmaps associated with pages. The modified version of *gdb* outputs a description of the layout of a type that is stored in the schema for an application on disk. When an application creates an object in a database, the information in the application's schema is used to update the bitmap for the page containing the object, and the schema for the database, if necessary.

Figure 5 illustrates the way that the structures described above are used to ensure that all of the pointers seen by an application program are valid swizzled pointers. In Figure 5a, disk page *a* is mapped to virtual frame *A*; however, page *a* has not been accessed by the application program, so read access on frame *A* has not been enabled. We now consider the actions that are taken by the QuickStore fault-handling routine when page *a* is first accessed. These are illustrated in Figure 5b.

In Figure 5b, the fault-handler has read both page *a* and the page containing the mapping object (as well as other mapping objects) associated with page *a* into main memory. The fault-handler then examines each entry in the mapping object, and uses the disk address contained in the entry to lookup the page descriptor associated with that disk address in the in-memory table. If no entry is found in the table, then one is created using the information contained in the mapping object entry. In the example in Figure 5b, the mapping object indicates that page *a* contains pointers that reference objects on three distinct disk pages. These include page *a* itself and two other pages, page *b* and page *c*. When the fault handler looks up pages *b* and *c*, it discovers that there is not a page descriptor entry for either page in the mapping table, so two entries are created.

When a table entry is created, the disk page (or pages, in the case of a large object) is assigned to its previous virtual frame (obtained from the mapping object entry) if the virtual frame is unused; otherwise a new frame is selected. If an entry for a disk page is found in the table, the system checks to see if the page is currently mapped to the same virtual frame as the one contained in its mapping object entry. If each disk page can be mapped to the virtual frame contained in its corresponding entry, then the swizzling process terminates. In Figure 5b, page *a* is already mapped to the frame *A*. In addition, pages *b* and *c* can be assigned to frames *B* and *C*, since frames *B* and *C* are currently not being used. Since all of the pages referenced by pointers contained on page *a* are assigned to the same memory locations that they occupied the last time page *a* was in memory, it isn't necessary to update (i.e. swizzle) any of pointers on page *a*, and the swizzling process terminates.

If some disk pages had been mapped to new locations in Figure 5b, then additional swizzling work would have been required. For example, suppose that page *c* couldn't have been assigned to frame *C* because another page had already been assigned to that frame. In that case, the pointers on page *a* that reference objects on page *c* would have to have been updated so that they referenced the new frame associated with page *c*. When pointers need to be swizzled, the bitmap object

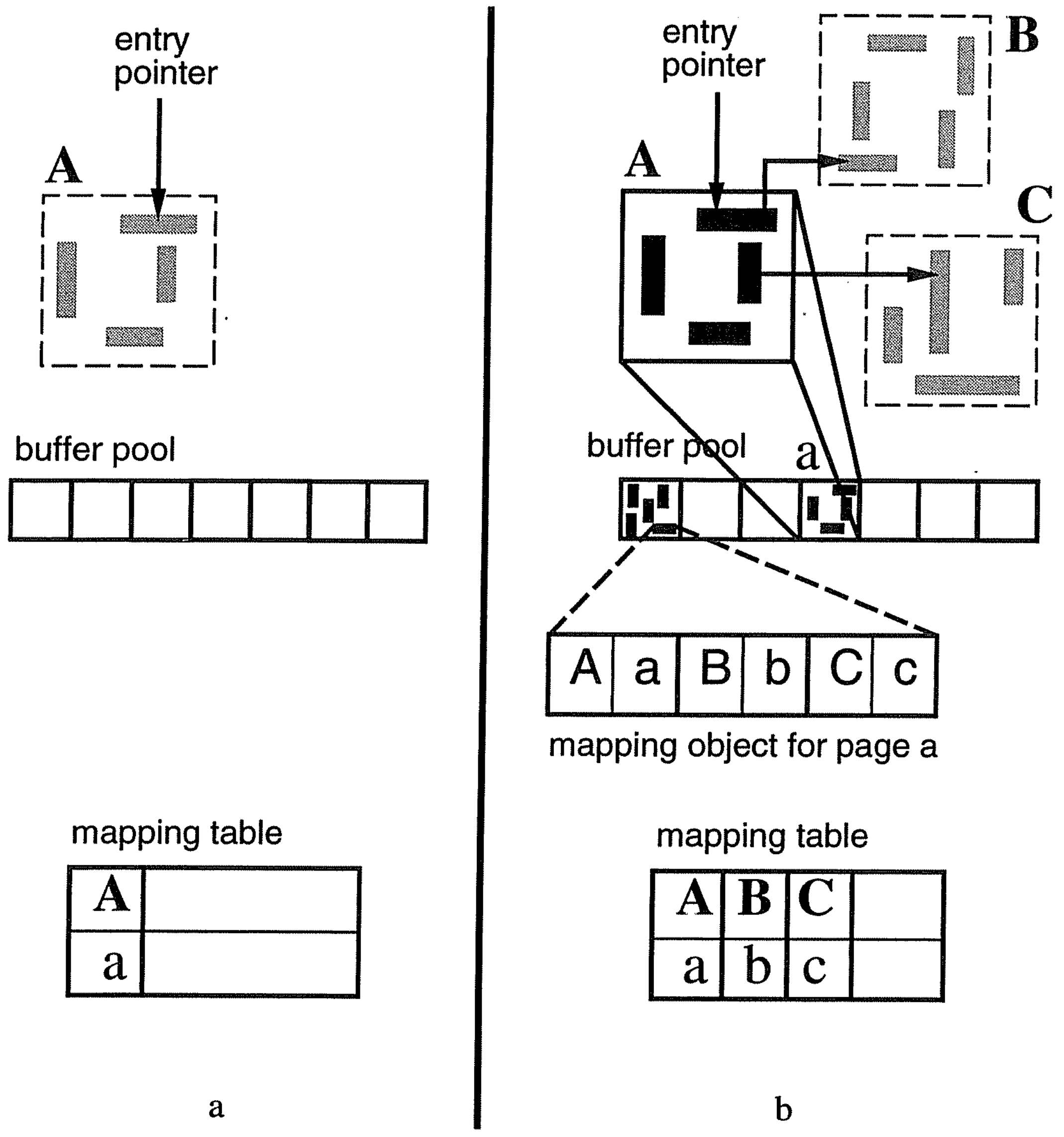


Figure 5. Pointer swizzling with no collisions.

associated with the page is read from disk and used to find and update any pointers on the page that need to be changed. Although only a subset of the pointers contained on a page may need to be changed, all of the pointers on the page must be examined since it is not known in advance which pointers actually need to be updated. Note that even though bitmap objects are fixed in size, they are stored separately from their corresponding data page since they hopefully will not have to be used in most cases.

We now comment on some possible alternative ways of storing the mapping information that is used to support pointer swizzling. Instead of storing information concerning the mapping of virtual memory frames to disk pages on a per page basis, one could also store it for groups of pages. For example, some object-oriented systems group pages into units called segments or clusters. One could even store the information at the level of the entire database or possibly a file. We chose not to follow this approach because it makes keeping the mapping information up to date considerably more complex. For example, suppose that a database is composed of a large number of clusters, each containing 10 pages, and that mapping tables are maintained for individual clusters. If the database is bigger than virtual memory, or is distributed (the same virtual frame may have been associated with different pages in the database in this case), then pages may need to be relocated when they are brought into memory. Page relocations are a problem if only a few pages of a cluster are accessed by an application, since to keep the mapping information for the cluster up to date one must either read the rest of the pages in the cluster and update them so that they are consistent with the new mapping, or keep multiple versions of the mapping information—one version for the pages whose mapping has changed, and one for the pages that are still using the old mapping. If many versions of the mapping information are stored then this scheme could be much worse than storing mapping information for individual pages.

### 3.5. Buffer Pool Management

Buffer managers in traditional database systems typically use an LRU algorithm or a clock style algorithm that approximates an LRU page replacement policy. We also felt that a clock algorithm was the best choice for use in QuickStore. However, implementing this type of scheme turned out to be more difficult in the context of a memory-mapped system, where objects in the buffer pool are accessed by dereferencing virtual memory pointers. The reason for this is that there is less information available to the buffer manager indicating which pages have been accessed recently.

Recall that in a traditional implementation of clock, a bit is usually kept for each frame in the buffer pool to indicate whether or not the frame has been accessed since the clock hand last swept over it. This bit is set by the database system each time the page is accessed and is reset by the clock algorithm. There is no way to set such a flag, however, when dereferencing a pointer in QuickStore.

One solution to this problem is to have the clock algorithm access-protect the virtual frame corresponding to a buffer pool frame when the clock hand reaches it. If the frame is subsequently reaccessed, a page-fault will occur and the fault handling routine can re-enable access to the page. This scheme replaces the usual setting and unsetting of bits in a traditional clock algorithm with the enabling and disabling of access permissions on virtual memory frames. We experimented with this solution, but in our experience the extra overhead of manipulating the page protections and handling additional page-faults made this approach prohibitively expensive in terms of performance.

To avoid the problem described above, QuickStore uses a simplified clock algorithm. Under this scheme the clock hand begins its sweep from wherever it stopped during the previous invocation of the algorithm. As soon as the clock hand reaches a page for which access is not enabled, the algorithm selects that page for replacement. If the clock hand reaches the end of the buffer pool without finding a candidate for replacement, however, then the **entire** virtual address space of the process being used for persistent data is reprotected with a single call to *mmap* and the algorithm is restarted, i.e. the clock hand begins scanning starting from the first frame in the buffer pool. This scheme performed much better than the original scheme outlined above in our experiments, and compared favorably with the more traditional clock replacement algorithm used by E (see Section 5).

## 3.6. Recovery

Implementing recovery for updates poses some special problems in the context of a memory-mapped scheme as well. For example, since application programs are able to update objects by dereferencing virtual memory pointers, it is difficult to know what portions of objects have been modified and require logging. Furthermore, it is desirable to batch the effects of updates together and log them all at once, if possible, since some applications may update the same object many times during a transaction.

Due to the considerations mentioned above, we decided to use a page diffing scheme to generate log records for objects that have been updated in QuickStore. Virtual memory frames that are mapped to pages in the database that have not been updated, do not have write access enabled, so the first attempt by the application program to update an object on a page will cause a page-fault. When the fault-handler detects that an access violation is due to a write attempt, it copies the original values contained in the objects on the page into an in-memory area called the *recovery buffer*. 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 before returning control to the application. The application program can then update the objects on the page directly in the buffer pool.

At transaction commit time, if paging in the buffer pool occurs, or the recovery buffer becomes full, the old values of objects contained in the heap and the corresponding updated values of objects in the buffer pool are compared (diffed) to determine if log records need to be generated. Log records are created and managed using the normal recovery services provided by ESM [Frank92]. The processes of diffing and generating log records are interleaved in QuickStore. To understand why this is the so, consider as an example the case when the first and last byte of an 1 K-byte object have been updated. In this case QS minimizes the amount of data written to the log by generating two log records, one for each modified byte, instead of one big log record for the entire object. On the other hand if the first, third, and fifth bytes had been modified, QS would generate a single log record for the first five bytes of the object. This is cheaper than generating multiple log records since each log record contains a relatively large (~50 byte) header area for storing information needed by the ESM recovery scheme.

The algorithm that minimizes the amount of data written to the log by deciding whether to generate log records for individual modified regions of an object or to combine the regions and log them as a single unit is fairly straight forward, and will not be discussed in detail here. It takes into account things such as the number of bytes separating modified regions of an object, and the size of log records that would be generated to make its decision.

Care must also be taken when processing updates to update the mapping tables associated with each modified page if necessary. Recall that the mapping table for a page keeps track of the set of pages that are referred to by pointers on the page. Updates to objects on a page can change the pages that are members of this set, making it necessary to update the mapping tables as well. Updating the mapping table for a page requires that each pointer contained on the page be examined and the in-memory table consulted to determine which page in the database it references. The bitmap for the page is used to locate the pointers that it contains and from these pointers a new set of referenced pages is constructed. This new set is then compared element by element with the old set to see if the set has changed. If it has, then the mapping object for the page is updated to reflect the new set of referenced pages.

## 4. Performance Experiments

This section describes the structure of the OO7 benchmark database and the benchmark operations that were included in the performance study. The hardware and software systems included in the study are also discussed. We include a brief description of the current implementation of the E programming language to aid in understanding the performance results presented in Section 5.

### 4.1. The OO7 Benchmark Database

The OO7 database is intended to be suggestive of many different CAD/CAM/CASE applications. There are two sizes of the OO7 database: small and medium. Table 1 summarizes the parameters of the database.

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. The number of composite parts per module is controlled by the parameter *NumCompPerModule* which was set to 500. Each composite part has a number of attributes, including the integer attributes **id** and **buildDate**. Associated with each composite part is a *document* object that models a small amount of documentation associated with the composite part. Each document has an integer attribute **id**, a small character attribute **title** and a character string attribute **text**. The length of the string attribute is controlled by the parameter *DocumentSize*.

| Parameter | Small | Medium |
|---|---|---|
| NumAtomicPerComp | 20 | 200 |
| NumConnPerAtomic | 3 | 3 |
| DocumentSize (bytes) | 2000 | 20000 |
| Manual Size (bytes) | 100K | 1M |
| NumCompPerModule | 500 | 500 |
| NumAssmPerAssm | 3 | 3 |
| NumAssmLevels | 7 | 7 |
| NumCompPerAssm | 3 | 3 |
| NumModules | 1 | 1 |

Table 1. OO7 Benchmark database parameters.

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". In the small database, each composite part's graph contains 20 atomic parts, while in the medium database, each composite part's graph contains 200 atomic parts.

Each atomic part has the integer attributes **id**, **buildDate**, **x**, **y**, and **docId**. The **buildDate** values in atomic parts are randomly chosen in the range *MinAtomicDate* to *MaxAtomicDate*, which is 1000 to 1999. Each atomic part is connected via a bi-directional association to three other atomic parts according to the parameter *NumConnPerAtomic*. The connections between atomic parts are implemented by interposing an information-bearing connection object between each pair of connected atomic parts. A connection object contains the integer field **length** and the short character array **type**. Figure 6 depicts a composite part, its associated document object, and its associated graph of atomic parts.

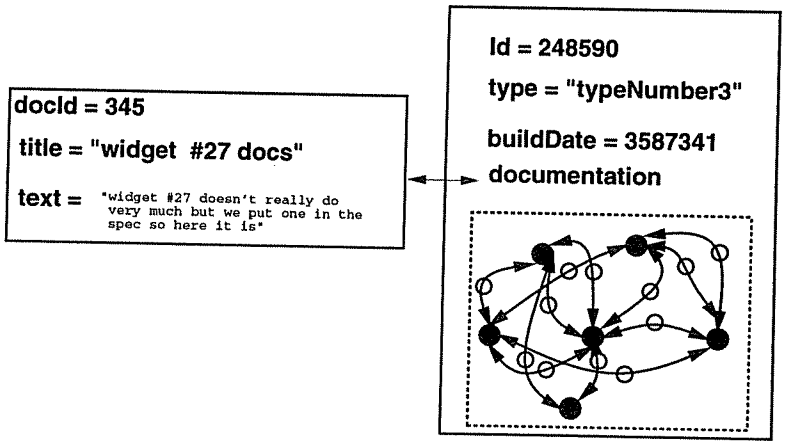

docId = 345
title = "widget #27 docs"
text = "widget #27 doesn't really do very much but we put one in the spec so here it is"
Id = 248590
type = "typeNumber3"
buildDate = 3587341
documentation

Figure 6 A composite part and its associated document object.

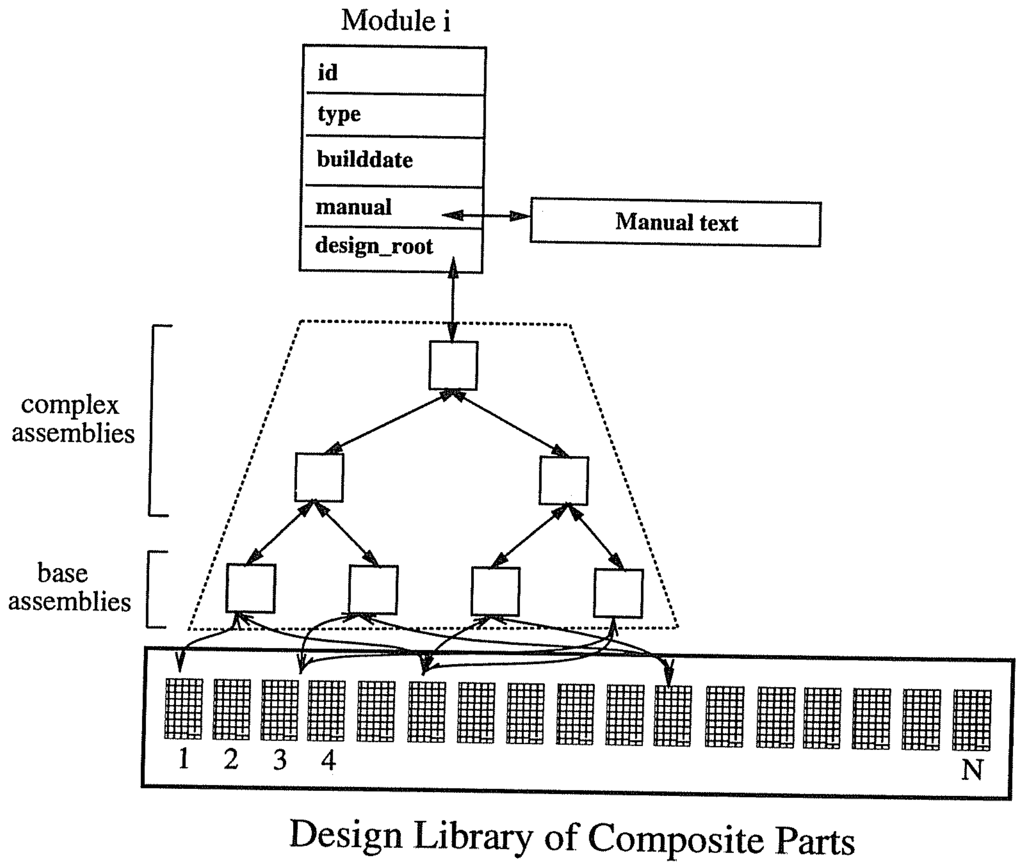


Figure 7. Structure of a module.

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. Base assembly objects have the integer attributes **id** and **buildDate**. 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). There are seven levels in the assembly hierarchy.

Each assembly hierarchy is called a *module*. Modules are intended to model the largest subunits of the database application. Modules have several scalar attributes. Each module also has an associated *Manual* object, which is a larger version of a document. Manuals are included for use in testing the handling of very large (but simple) objects. Figure 7 roughly depicts the full structure of the single user OO7 Benchmark Database. The picture is somewhat misleading in terms of both shape and scale; the actual assembly fanout used is 3, and there are only $(3^7-1)/2=1093$ assemblies in the small and medium databases, compared to 10,000 atomic parts in the small database and 100,000 atomic parts in the medium database.

## 4.2. The OO7 Benchmark Operations

This section describes the OO7 benchmark operations used in the study. Some of the OO7 operations were omitted because they didn't highlight any additional differences among the systems being compared. The operations, which are referred to by type and number, consist of a set of 10 tests termed traversals, and another set of 8 query tests. The queries were hand coded in C++ since neither QuickStore or E provide a declarative query language. We comment on the implementation used to execute the queries, although this is not part of the OO7 specification [Carey93].

### 4.2.1. Traversals

The T1 traversal performs 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. The traversal returns a count of the number of atomic parts visited, but otherwise no additional work is done during the traversal. The T6 traversal is similar to T1, except that instead of visiting the entire graph of atomic parts of each composite part, T6 just visits the root atomic part and returns.

T2 and T3 are also similar to T1, but they include updates. Each T2 traversal increments the (x,y) attributes contained in atomic parts as follows[1]:

T2A Update the root atomic part of each composite part.

T2B Update all atomic parts of each composite part.

T2C Update all atomic parts of each composite part four times.

The T3 traversals are similar to T2 except that the **buildDate** field of atomic parts in incremented. This field is indexed, so T3 highlights the cost of updates of indexed fields. We also used three traversals that are not based on T1. T7 picks a random atomic part and traverses up to the root of the design hierarchy. T8 scans the manual object associated with the module and counts the occurrences of a specified character, and T9 compares the first and last characters of the manual to see if they are the same.

[1][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 scheme used for recovery by QuickStore will always generate a log record.

### 4.2.2. Queries

Query Q1 randomly retrieves 10 atomic parts. This is done by using an index based on part id. Q2 selects the most recent 1% of atomic parts based on buildDate, while Q3 looks up the most recent 10%. Both queries do an index scan to find the parts. Query Q4 looks up 10 document objects at random using the index on their title field. It then visits all base assemblies that use the composite part corresponding to each document. This is done by traversing pointers from documents to composite parts and then processing a collection of pointers to base assemblies that is maintained as part of each composite part object. Q5 is referred to as a single level make operation. Q5 finds all base assemblies that use a composite part with a build date later than the build date of the base assembly. Q5 is implemented as a nested loops pointer join between base assemblies and composite parts, i.e it iterates over the collection of base assemblies maintained for the module and as each base assembly object is visited, the references to composite parts that it contains are traversed.

## 4.3. Hardware Used

As a test vehicle we used a pair of 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 disks. For the client we used a Sun Sparc ELC workstation (about 20MIPS) configured with 24 megabytes of memory and one 207 megabyte disk drive (model Sun0207). This disk drive was used to hold system software and as a swap device.

## 4.4. Software Used

The systems examined in the study use ESM V3.0 to provide disk storage for persistent objects. ESM provides files of untyped objects of arbitrary size and B-tree indices. ESM uses a page-server architecture where client processes request pages that they need from the server via TCP/IP. If the server cannot satisfy the request from its buffer pool, a disk I/O is initiated by invoking a disk process to perform the actual I/O operation. After the disk process has read in the page, the server process returns it to the requesting client process and keeps a copy in its own buffer pool. ESM also provides concurrency control and recovery services. Locking is provided at the page and file levels with a special non-2PL protocol being used for index pages. Recovery is based on logging the changed portions of objects.

ESM used a disk page size of 8 K-bytes (which is also the unit of transfer between a client and the server). The client and server buffer pools were set to 1,536 pages (12 MBytes) and 4,608 pages (36 Mbytes) respectively. Release 4.1.3 of SunOS was run on both of the workstations used in the experiments. QuickStore was compiled using the GNU C++ compiler V2.3.1. The E compiler is a modified version of the GNU compiler.

### 4.5. Database Systems Tested

#### 4.5.1. E

This section briefly describes the current implementation of the E language. E and QuickStore both offer basically the same functionality, however, E implements persistence using a software interpreter, EPVM 3.0. EPVM 3.0 has a functional interface, so operations such as dereferencing an unswizzled pointer in E are handled by calling an EPVM function of perform the dereference. As part of handling a reference to a persistent object, EPVM may in turn call ESM if the page containing an object is not in memory, and update its own internal data structures before returning control to the application. In addition to calls of EPVM functions, the code generated by the E compiler (a modified version of the gnu C++ compiler) contains in-line code sequences to handle certain basic operations. For example, residency checks and dereferences of swizzled pointers are done in-line and do not require a function call, which improves performance.

Like QuickStore, EPVM 3.0 accesses memory-resident persistent objects directly in the ESM client buffer pool. The interpreter maintains a hash table that contains an entry for each page of objects that is currently in memory. The pointer swizzling scheme used in EPVM 3.0 is similar to the scheme used in EPVM 1.0 [Schuh90] except that swizzled pointers point directly to objects in the buffer pool. This swizzling scheme only swizzles pointers that are local variables in C++ functions. Pointers within persistent objects are not swizzled because this makes page replacement in the buffer pool difficult [White92].

Update operations on persistent objects are always handled by an interpreter function in EPVM 3.0. The update scheme used copies the original values of objects into a side buffer, and updates the objects in place in the buffer pool. Original values of objects and updated values are used to generate log records at transaction commit, or sooner if the side buffer or the buffer pool become full. However, no diffing is performed as in QuickStore. EPVM 3.0 employs a scheme that breaks large objects into 1K chunks for logging purposes. Objects that are smaller than 1K are logged in their entirety.

#### 4.5.2. QuickStore

A detailed description of QuickStore was given in Section 3. Although QuickStore and E offer nearly the same functionality, it is important to point out one fundamental way in which the two systems differ. This has to do with the degree to which the two systems support the notion of object identity [Khosh86]. Section 3 described the scheme used by QuickStore to implement a mapping between virtual memory frames and disk pages. This mapping is maintained for pages when they are in memory as well as when they reside on disk. Because of this mapping, pointers to persistent objects can be viewed as a <virtual frame, offset> pair, where the high order bits of the pointer identify the virtual memory frame referenced by the pointer and the low order bits specify an offset into the frame. Virtual memory frames are mapped to disk pages, so pointers really just specify offsets or locations on pages.

To see why this scheme doesn't support object identity, consider what happens when an object, for which there are outstanding references, is deleted. The page that contained the object can be faulted into memory by subsequent program runs and mapped to some virtual memory frame. If the program then dereferences dangling pointers to the deleted object, no error will be explicitly flagged. If a new object occupies (or overlaps) the space on the page previously used by the deleted object then the dangling pointers will reference this object.

QuickStore doesn't fully support object identity or "checked references" to objects because the overhead would be prohibitive. For example, the meta-data that would be required to associate every unique pointer on a page with its corresponding OID would likely be an order of magnitude greater than the current scheme used by QuickStore. Furthermore, we are aware of no commercial or research system (including ObjectStore) that supports these types of references for normal pointers in the context of a memory-mapped scheme. E, on the other hand, supports object identity fully, including "checked references". E implements this by storing pointers as full 16 byte OIDs within objects. This is a reasonable approach, but it does incur certain costs. For example, since objects are larger in E than in QuickStore, the database as a whole is larger, and E generally performs more I/O as a result. Also, dereferencing big pointers is more expensive even in terms of CPU requirements than dereferencing regular virtual memory pointers.

Because of these differences, we included a third system in the performance study. This system is identical to QuickStore, except that the size of each object has been padded so that it is the same as the corresponding object in E. Comparing the performance of this system to the performance of E in the experiments where faults take place gives insight into the overhead of faulting for the memory-mapped approach, while comparing it with QuickStore indicates the advantage gained by QuickStore due to its smaller object size. In addition, one can think of this system as approximating the performance of a hybrid memory-mapped scheme that allows large pointers to be embedded within objects, thus supporting both checked and unchecked references.

## 5. Performance Results

### 5.1. Database Sizes

The size of the OO7 database is important to understanding the performance results. Table 2 shows the database sizes for E, QuickStore (QS), and QuickStore with big objects (QS-B)[2]. The QS database is roughly 60% as big as the E database

[2]Objects in QS-B are padded to the same size as the corresponding objects in the E implementation.

for both the small and medium cases. This is because of the different schemes used by the systems to store pointers. The QS-B database is slightly bigger than the E database due to the overhead for storing bitmaps that indicate the locations of pointers on pages and mapping objects. Mapping objects accounted for approximately 3% of the database size for QS and QS-B, while bitmaps generally accounted for an additional 3%.

| | Small | Medium |
|---|---|---|
| QS | 6.6 | 54.2 |
| E | 10.5 | 94.1 |
| QS-B | 11.5 | 98.5 |

Table 2. Database Sizes (in megabytes)

### 5.2. Small Cold Results

This section presents the the cold results for the small database experiments. The cold results were obtained by running the OO7 benchmark operations when no data was cached in memory at either the client or server machines. The times presented represent the average of 10 runs of the benchmark operations, except where noted otherwise. The times were computed by calling the Unix function *getimeofday* which had a granularity of several microseconds on the client machine.

#### Read-Only Results

Figure 8 and Table 3 list the cold response times and the number of client I/O requests, respectively, for the read-only traversals. As Figure 8 shows, QS is 37% faster than E during T1[3]. This difference in performance is largely due to the smaller database size for QS which causes it to read 53% fewer pages from disk than E (Table 3). Almost all of the I/O activity during T1 resulted from reading clusters[4] of composite parts. Each composite part cluster occupied a little less than one page for QS, while two pages were required for E. This accounts for the roughly 2 to 1 ratio in the number of disk reads between the two systems. Comparing E with QS-B, we see that QS-B is 15% slower than E during T1. QS-B always issues slightly more I/O requests than E since QS-B must read mapping objects to support the memory-mapped scheme.

The performance of QS is only 4% better than E during T6[5]. Again, this difference is largely due to the number of disk reads that each system performs. Table 3 shows that the amount of I/O for QS is almost the same during T1 and T6, while the number of disk reads for E decreases by 41% during T6. This is due to the size difference of composite part

[3]T1: DFS of assembly hierarchy visiting all atomic parts.

[4]The atomic part objects and connection objects associated with a composite part were clustered together on disk together with the composite part object itself in our implementation of OO7.

[5]T6: DFS of assembly hierarchy visiting only the root atomic part.

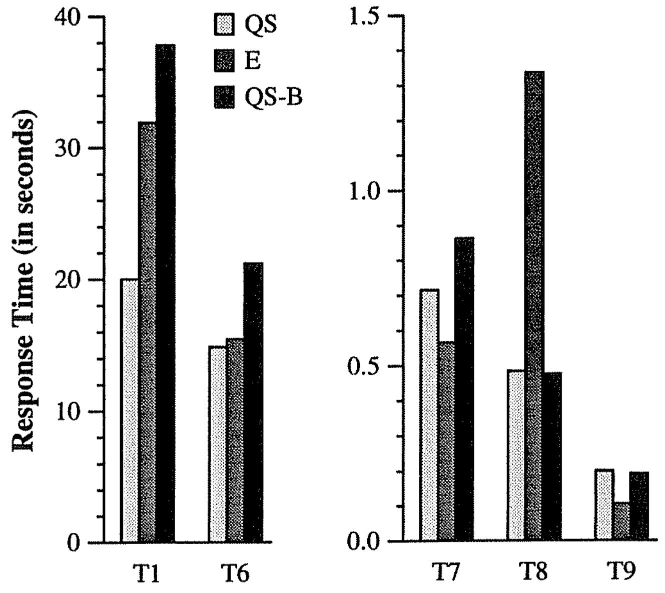


Figure 8. OO7 traversal cold times, small database.

| | T1 | T6 | T7 | T8 | T9 |
|---|---|---|---|---|---|
| QS | 474 | 467 | 26 | 19 | 9 |
| E | 1018 | 600 | 25 | 18 | 7 |
| QS-B | 1047 | 639 | 31 | 19 | 9 |

Table 3. Client I/O requests, traversals, small database.

clusters between the two systems. E does noticeably fewer I/O operations during T6 because, unlike QS, it generally doesn't read entire clusters. The performance of QS-B is 27% slower than E during T6. As the detailed faulting times shown below will illustrate, this difference in performance is close to the actual percentage difference in individual page fault costs for the systems, as the CPU cost of performing the traversal itself has less of an overall impact on performance during T6 than during T1.

E has the best performance during T7[6]. QS is 26% slower than E because of increased faulting costs relative to T1 and T6. One reason faults are more expensive for QS during T7 is that a large fraction of faults (86%) are due to reading pages of *base assembly* objects. These pages have larger mapping tables because pointers from *base assembly* objects to *composite part* objects are uniformly distributed among all *composite part* objects in the database. This increases the average I/O cost for reading the mapping tables and the number of table entries (139 on average for T7 vs 20 on average for T1) that must be

[6]T7: Traverse up the assembly hierarchy starting from a randomly selected atomic part.

examined per fault. The relative performance of QS also slows during T7 because, as in T6, objects are accessed in an unclustered fashion. This increases the amount of I/O operations required to read mapping objects, since mapping objects are clustered on disk in the order of the pages to which they correspond. As Table 3 shows, QS actually performs more I/O during T7 and E. Finally, we note that QS-B is 34% slower overall during T7 relative to E.

Turning to T8[7], Figure 8 shows that E is roughly 3 times slower than QS. This is because the E interpreter is invoked once for each character of the manual that is examined by T8, while QS simply has to dereference a virtual memory pointer to access the manual. By contrast, Figure 8 shows that E is nearly twice as fast as QS on T9[8]. This difference is due almost entirely to faulting costs since very little work is done on the objects faulted in during T9. It is not surprising that faulting costs for QS are relatively high in this case. T9, like T7, touches only a few pages in the database and the pages that are accessed are not clustered. QS and QS-B have similar performance during T8 and T9 since character data is the same size for both systems.

The cold response time and client I/O requests for the queries are shown in Figure 9 and Table 4, respectively. E is 24% faster than QS and 26% faster than QS-B during Q1[9]. The memory mapped scheme is slower in this case because the relatively small number of atomic part objects that are accessed during Q1 are accessed randomly. This causes one fault to be performed by all of the systems for each atomic part object that is accessed. In addition, Table 4 shows that QS performs several additional I/O operations (16%) to read mapping objects. The number of additional I/Os required to read mapping objects is relatively high during Q1 because, as mentioned above, mapping objects are clustered according to the disk pages to which they correspond, so when accesses to pages in the database are unclustered, more pages containing mapping objects tend to be read.

E also has the best performance during Q2[10]. As Table 4 shows, QS and QS-B again do more I/O than E during Q2 which contributes to their worse performance. The reasons for this are similar to those given for Q1. Since only a small fraction of atomic part objects (1%) are accessed during Q2, the accesses are unclustered to the point that roughly one fault is required per object accessed (and several faults require additional I/O to read mapping objects for QS). When a larger percentage of objects (10%) are accessed as in Q3[11], QS has the best performance. Figure 9 shows that QS is 27% faster than E during Q3. QS faults in fewer pages than E in this case because object accesses are more highly clustered. E is 22% faster than QS-B during Q3 again illustrating the effect on performance of differences in faulting costs.

[7]T8: Scan the manual object counting occurrences of a specified character.

[8]T9: Compare first and last characters of the manual to see if they are equal.

[9]Q1: randomly retrieve 10 atomic parts.

[10]Q2: retrieve most recent 1% of atomic parts.

[11]Q3: retrieve most recent 10% of atomic parts.

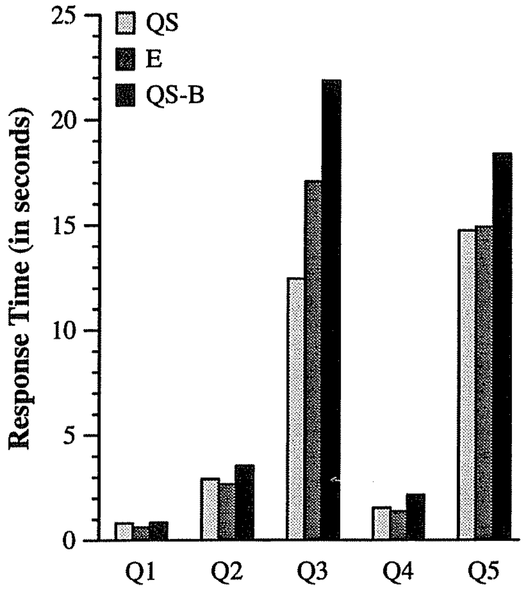


Figure 9. OO7 query cold times, small database.

| | Q1 | Q2 | Q3 | Q4 | Q5 |
|---|---|---|---|---|---|
| QS | 31 | 109 | 413 | 62 | 467 |
| E | 26 | 104 | 641 | 59 | 558 |
| QS-B | 33 | 121 | 663 | 74 | 583 |

Table 4. Client I/O requests, queries, small database.

QS is 11% slower than E during Q4[12]. This is partly due to unclustered accesses to document and composite part objects which require roughly one fault per object accessed in both systems. A second factor is that a large fraction of the objects accessed by Q4 are base assembly objects, producing higher faulting costs due to the relatively large amount of mapping information associated with them. QS and E have basically the same overall performance during Q5[13] (Figure 9). QS performs 16% fewer I/O operations than E does, but the higher per fault cost for QS prevents it from performing faster than E. Finally, we note that E is 36% and 19% faster than QS-B during Q4 and Q5, respectively.

The results presented in Figures 8 and 9 for QS and E show that the memory-mapped scheme of QS offers better performance when objects are accessed in a clustered fashion. When objects are accessed in an unclustered fashion, however, E has better performance. In this case, the smaller object size of QS doesn't provide a noticeable savings in I/O, and QS actually performs slightly more total I/O operations than E because of the need to read mapping objects. The results presented in

[12]Q4: lookup 10 document objects and find the base assemblies that use their associated composite part.

[13]Q5: pointer join between base assemblies and composite parts.

Figures 8 and 9 also demonstrated that the per page faulting cost for the memory mapped approach is noticeably higher than for the software approach. For example, except for T8 where other differences in CPU cost determined relative performance, QS-B always has slower performance than E in Figures 8 and 9.

**Detailed Faulting Results**

We next examine the differences in faulting costs between the systems in more detail. Table 5 shows the average cost per fault in milliseconds for each of the systems during T1 and T6. These times were calculated by subtracting the time required to execute a hot traversal from the time required for a cold traversal, and then dividing the result by the number of page faults to get the average per fault cost. To make sure that the results obtained were accurate, the numbers used to perform the calculation represented the average of 100 runs of each traversal experiment.

According to Table 5, the faulting cost for QS-B is slightly higher than for QS. We speculated that this was due to a larger number of pages that were being referenced on average by outbound pointers for QS-B, since there are more pages in the QS-B database. (An outbound pointer is a pointer that refers to an object on a page other than the page containing the pointer.) This would increase the size of the mapping tables for QS-B. However, this turned out not to be the case since the average number of outbound pointers per page in the small database was 16 for QS and only 12 for QS-B. The comparison between QS and E in Table 5 is more interesting. It shows that individual page faults are roughly 20% more expensive for QS during T1 and T6. The corresponding figure for QS-B and E averages 26%, which correlates closely with the difference in response time between QS-B and E during T6.

| system | time(ms.) | |
|---|---|---|
| | T1 | T6 |
| QS | 29.4 | 33.1 |
| E | 23.7 | 26.5 |
| QS-B | 31.6 | 34.5 |

Table 5. Average Faulting Cost.

| description | time(ms.) | |
|---|---|---|
| | T1 | T6 |
| min faults | 1.8 | 1.6 |
| page fault | .8 | .7 |
| misc. cpu overhead | .5 | .2 |
| data I/O | 24.8 | 28.5 |
| map I/O | 1.1 | 1.1 |
| swizzling | .4 | .4 |
| mmap | .8 | .8 |
| total | 30.2 | 33.3 |

Table 6. Detailed QS Faulting Times.

To better understand the additional faulting overhead of the memory mapped scheme, Table 6 shows a detailed breakdown of the average faulting time for QS. As a check we present detailed numbers for both T1 and T6. One would except most of the costs for T1 and T6 to be similar since they fault in many of the same pages. The *min fault* entry in Table 6 is present due to the way our implementation interacts with the virtually mapped CPU cache of the client machine (see Section

3.2). This effect increased the average fault time by 6% and 5%, respectively for T1 and T6. The entry labeled *page fault* in Table 6 is the time that was required to detect the illegal page access and invoke the fault handler. Page faults comprised 3% of average faulting time for T1 and 2% for T6. We note that it was not possible to measure the times given for the *min fault* and *page fault* entries directly by running the benchmark. These times were obtained instead by measuring a test application that performed the operations several thousand times in a tight loop.

The remaining table entries break down the time spent in the fault handling routine. The entry for *misc. cpu overhead* includes time for looking up the address that caused the fault in the in-memory table, various residency and status checks to determine the appropriate action to take in handling the fault, and other miscellaneous work. *Data I/O* is the time needed to read the page of objects from disk and update the buffer manager's data structures. This accounted for largest fraction of faulting time, 82% for T1 and 85% for T6. The portion of time spent reading mapping tables (*map I/O*) was 3.5% for T1 and 3.2% for T6. The *swizzling* entry gives the time needed to process the mapping table entries. Swizzling costs were quite low, accounting for 1% to 2% of the faulting cost on average. Since all of the pages read were mapped to the locations in memory that they occupied previously, the swizzling time doesn't include any overhead for updating pointers on pages that are inconsistent with the current mapping. The final entry, labeled *mmap*, gives the average time taken by the mmap system call to change the access protections. This accounted for a modest 3% of the faulting time. Finally, we note that the sums of the detailed times given in Table 6 correlate closely with the total per fault times given in Table 5.

**Update Results**

We next consider traversals T2 and T3 which include updates. Figure 10 shows the total response time for these traversals run as a single transaction. The read requests for T2 were nearly identical to T1 (Table 3), while the T3 traversals performed a few additional I/Os to read index pages. During T2A, which updates the root atomic part of each composite part, QS is 4% faster than E (Figure 10). This may seem surprising given that QS was 37% faster than E during T1 which does the same traversal as T2A, but without updates. The difference in performance between QS and E diminishes during T2A because the page-at-a-time scheme for handling updates of QS is more expensive than the object-at-a-time approach of E when sparse updates are done.

Part of the increase in response time for QS is due to the fact that the number of page access violations increases from 454 during T1 to 878 during T2A, nearly doubling. The additional access violations occur when the first attempt is made to update an object on a page during the transaction. When this happens, a fault-handling routine is invoked to handle the access violation. As mentioned in Section 3.6, this routine performs several functions. First, it copies the objects contained on the page into the recovery buffer so that the original values contained in the objects may be used to generate logging records for updates (by diffing) at a later time. Next, it calls ESM, if necessary, to obtain an update (exclusive) lock on the

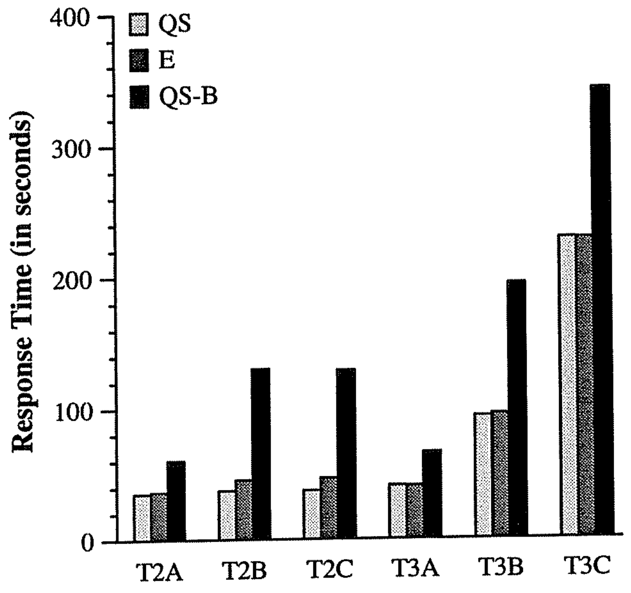


Figure 10. T2 and T3 Response Times.

page, and finally, it changes the virtual memory protections on the page so that the instruction that caused the exception can be restarted.

Our measurements showed that during T2A, a total of 5.198 seconds were needed to carry out this work for QS, which amounted to roughly 12.3 ms for each of the 423 pages updated. Of this, 7.3 ms was spent copying the objects on the page, 2.8 ms was used to upgrade the lock on the page, and .9 ms on average was spent calling *mmap* to change the page's protection to allow write access.

The response time of QS also increases relative to E during T2A because transaction commit is more expensive for QS (Figure 11). The commit time for QS can be broken down into the time required to perform three basic activities, plus a small amount of additional time to perform minor functions like reinitializing data structures, etc. The first of the basic operations involves *diffing* objects on pages that have been updated and calling ESM to generate log records when it is determined that updates did occur. The diffing phase required a total of 3.035 seconds during T2A, of which .182 seconds was spent calling ESM to generate the 491 log records needed. Thus, the time needed on average to *diff* each of the 423 modified pages (not counting time to generate log records) was 6.7 milliseconds.

The second major task performed during transaction commit is to update the mapping object associated with each modified page. Our measurements showed that 3.084 seconds (7.2 ms per page) were required for this phase of commit processing. The final step in committing a transaction is performed by ESM. This involves writing all log records to disk at the server and flushing all dirty pages back to the server from the client. This phase of commit processing required 3.501

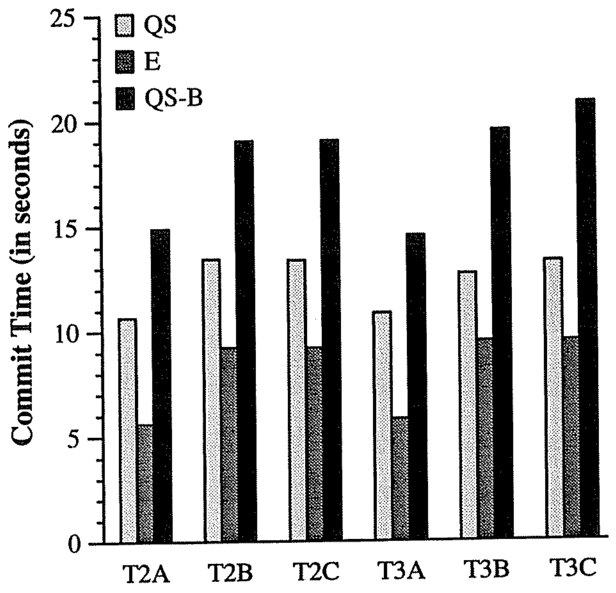


Figure 11. T2 and T3 Commit Times.

seconds during T2A.

Turning to T2B and T2C, we see that QS is 17% and 20% faster than E, respectively. As one would expect, QS does better relative to E during T2B and T2C when updates are more dense since QS copies and *diffs* fewer objects unnecessarily. In fact the absolute performance of QS degrades only slightly during T2B relative to T2A. This is due almost entirely to increased time during commit for *diffing* objects and generating log records. More precisely, 5.804 seconds was required do the *diffing* during T2B (.280 seconds of this was for generating log records). The average *diffing* cost per page was 12.9 ms during T2B (not counting logging). We also note that the performance of QS was basically the same during T2B and T2C, while the performance of E was 5% slower. This is because repeatedly updating an object is very inexpensive in QS, as objects are accessed via normal virtual memory pointers while updating an object under the approach used by E requires a function call per update.

The performance difference between QS and E narrows further during T3 relative to T2 and T1. QS has better performance than E in all cases, but nearly similar overheads for index maintenance make this difference less noticeable. In contrast to the relatively stable performance of the systems during T2, the response times of the systems steadily increase when going from T3A to T3B to T3C. This is because each update of an indexed attribute results in the immediate update and logging of the update to the corresponding index. Although the schemes employed by QS and E in using the ESM B-tree indices differ slightly, their performance is basically the same. Neither of the systems supports automatic index maintenance, so the index updates are coded as C++ method invocations on a class variable of type *index*. QS-B is always much slower than the other systems during T2 and T3, especially during the B and C traversals. This is because the 4Mb area used to hold recovery data wasn't big enough to hold all of the objects from modified pages during these traversals for QS-B.

## 5.3. Small Hot Results

The hot results were obtained by re-running the OO7 benchmark operations after all of the data needed by each operation had been cached in the client's main memory by the cold traversal. Figure 12 shows hot times for the traversals and Figure 13 shows hot times for the queries run on the small database. The times for QS-B are omitted since they were identical to those shown for QS.

As one would expect, the performance of QS is generally better than E. It is somewhat surprising, however, that E is just 23% slower than QS during T1. To determine the reasons for this relatively small difference, we used *qpt* [Ball92] to profile the benchmark application. Table 7 presents the profiling results for T1. The T1 hot traversal time has been broken down in Table 7 based on the percentage of CPU time spent in several groups of functions. Table 7 shows that E spent 33% of the time executing EPVM 3.0 interpreter functions. Most of this time was spent dereferencing unswizzled pointers. Both QS and E spent a considerable amount of time allocating and deallocating space in the transient heap (see the entry for *malloc*). This is because an "iterator" object is allocated in the heap for every node (assembly object, composite part, and atomic part) in the object graph that is visited during the traversal. The "iterator" object establishes a cursor over the collection of

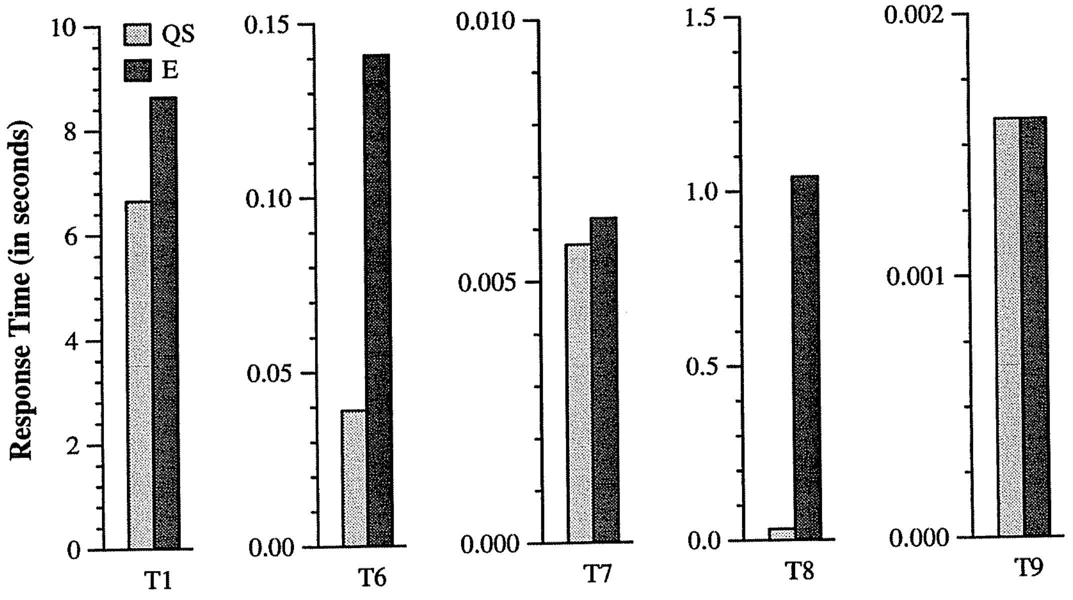


Figure 12. Traversal Hot Times.

pointers to sub-objects so that the sub-objects can be traversed.

The entry labeled *part set* in Table 7 gives the time spent executing functions that maintain the set of atomic part ids that have been visited in each composite part's subgraph of atomic parts. This set is needed so that the same atomic part is not visited more than once. The amount of time spent in other functions that implement the traversal, such as functions that iterate over collections of pointers to sub-objects and that implement the recursive traversal, was 8% for QS and 17% for E. The higher percentage for E reflects the additional cost of dereferencing large pointers in E. When each node in the object graph is visited, a simple function is called that examines a field in the object to make sure that the object is faulted into memory. The time spent in these functions was .7% for both systems. The detailed numbers in Table 7 are surprising because the small amount of additional work involving transient data structures that were needed to implement T1 accounts for a such a large percentage of the overall cost. The results show how quickly a small amount of additional computation can mask differences in the cost of accessing persistent data between the systems.

E is 3.6 times slower than QS during T6. QS performs better relative to E during T6 (the sparse traversal) than during T1 (the dense traversal) since there is less overhead for maintaining transient data structures during T6. For example, the sets of part ids are not maintained since only the root part is visited during T6. The performance of the systems is very close during T7. T7 visits very few objects in the database (10 to be precise) since it simply follows pointers from a single atomic part up to the root of the module. Thus, differences in traversal cost are easily diminished by other costs, such as the overhead to look the atomic part up in the index, etc. Figure 12 shows that E is a factor of 32 slower than QS during T8. T8 scans the manual object, a large object spanning several pages on disk. In the case of E, an EPVM 3.0 function call is performed for each character of the manual that is scanned, while QS accesses each character of the manual via a virtual memory pointer. E spent 91% of its time executing EPVM functions during T8. During T9 the systems have identical performance. T9 does very little work on persistent data that involves pointers, so the time shown in Figure 12 largely reflects the similarity in

| **description** | **% of time** | |
|---|---|---|
| | **QS** | **E** |
| EPVM 3.0 | - | 33.31 |
| malloc | 56.13 | 24.99 |
| part set | 35.18 | 24.57 |
| traverse | 8.03 | 17.12 |
| do nothing | 0.64 | 0.70 |
| misc. | 0.02 | 0.01 |
| total | 100.00 | 100.00 |

Table 7. T1 hot traversal detail.

index lookup costs between the systems.

We now discuss the query hot times shown in Figure 13. QS and E have nearly the same performance for all of the queries except Q5, where E was 3.6 times slower than QS. QS is faster than E during Q5 because Q5 does a lot of pointer dereferences as part of the pointer join between base assemblies and composite parts. The times for Q1 and Q2 reflect the cost of index lookups, which are the same between the systems. QS is actually a little slower then E during Q3 because the index scan code for QS is slightly less efficient than for E. This was due to coding differences and not to any fundamental difference between the systems. Q4 also performs several index lookups which make the performance of QS and E similar.

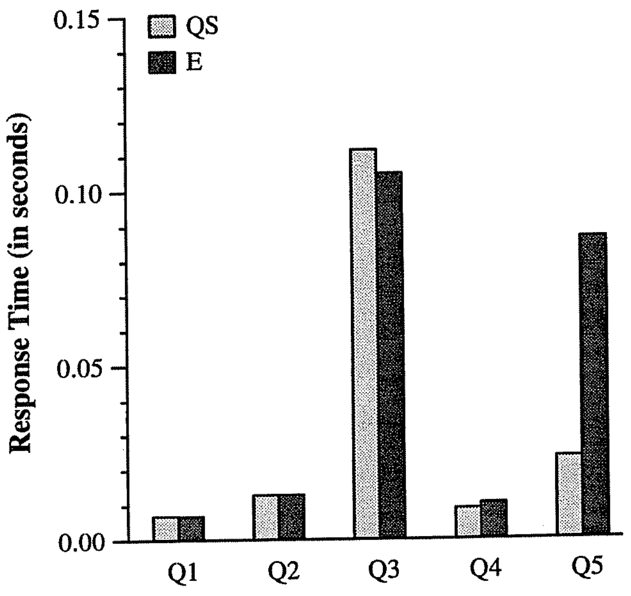


Figure 13. Query Hot Times.

### 5.4. Medium Cold Results

This section presents the cold times for the OO7 benchmark operations run on the medium database. The results presented represent the average of 5 runs of the benchmark experiments. Figure 14 presents the cold response times for the traversal operations and Table 8 gives the number of client I/O requests. We see in Figure 14 that, as in the case of the small database, QS has the best performance during T1. QS is 41% faster than E during T1 while it performs 63% fewer I/Os. E, on the other hand, is 36% faster than QS-B during T1. Comparing the relative performance of E and QS-B during T1 when using the small database (Figure 8), we see that the gap in performance between E and QS-B widens when the medium database is used. This is because of the additional cost to QS-B for managing paging in the client buffer pool.

E has better performance than QS during T6 and T7. QS is slower during T6 because one page fault is required to read each composite part in all of the systems and QS has higher per fault costs. The times shown for T7 and T8 are similar to the small database case. QS is slower due to higher per fault costs during T7. E is slower than QS during T8 due to the overhead of calling EPVM to scan each character of the manual. The results for T9 (not shown) were identical to the small case.

Figure 15 and Table 9 show the cold response times and client I/O requests for the queries run on the medium database. E always has the best performance during the queries. During Q1 and Q2 E benefits because accesses of atomic parts are very unclustered. The difference in performance between QS and E narrows during Q3 because accesses are more clustered in this case. It is interesting that QS is slower than E during Q3 since it performs significantly fewer I/O operations. QS is slower because of the overhead for managing paging in the client buffer pool. In Q4 and Q5 object accesses are again unclustered resulting in a high number of page-faults per object access, which causes QS to have slower performance than E.

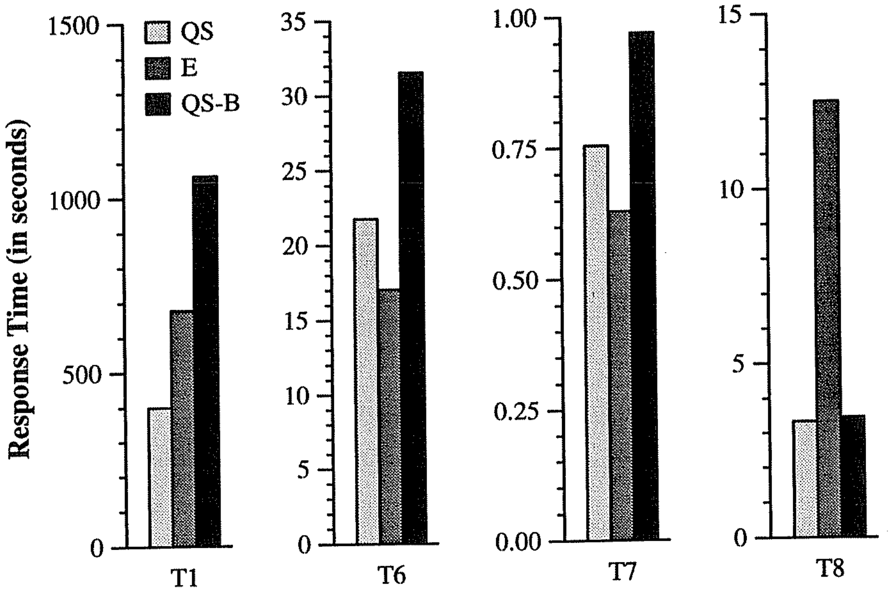


Figure 14. Medium Database, Traversal Cold Times.

| | T1 | T6 | T7 | T8 |
|---|---|---|---|---|
| QS | 13216 | 610 | 27 | 130 |
| E | 35622 | 558 | 25 | 129 |
| QS-B | 36963 | 802 | 32 | 130 |

Table 8. Traversal Cold I/Os.

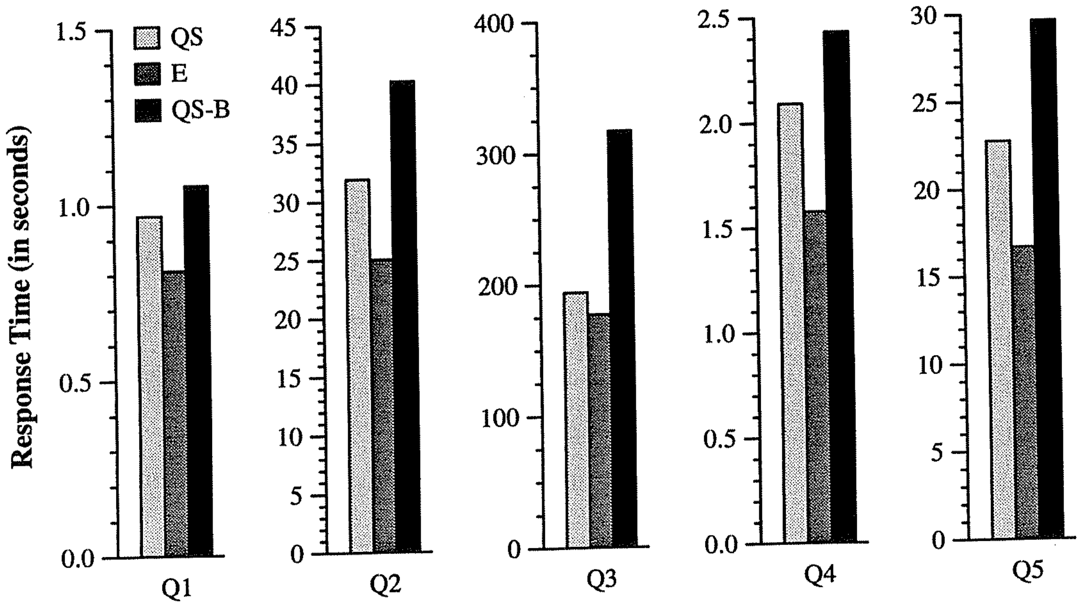


Figure 15. Medium Database, Query Cold Times.

| | Q1 | Q2 | Q3 | Q4 | Q5 |
|---|---|---|---|---|---|
| QS | 34 | 901 | 5997 | 68 | 595 |
| E | 26 | 919 | 8045 | 58 | 558 |
| QS-B | 35 | 1095 | 10951 | 81 | 751 |

Table 9. Query Cold I/Os.

Turning now to traversals T2 and T3 (Figure 16) which perform updates, we see that QS outperforms E during the T2A and T3A traversals which only update the root atomic part of each composite part. This is understandable when one considers that both QS and E have to do basically the same amount of work to process the updates that they did in the small database case. This makes the cost difference of doing the traversal itself the main factor effecting their relative performance. The relative performance of QS worsens during T2B and T2C causing QS and E to have similar performance. Recovery is more expensive for QS during T2B and T2C since the buffer used for recovery is much smaller than the fraction of the database that is updated. QS-B has much worse performance than both QS and E in Figure 16. This is caused by the fact that in addition to higher traversal costs, QS-B has higher costs for recovery as well.

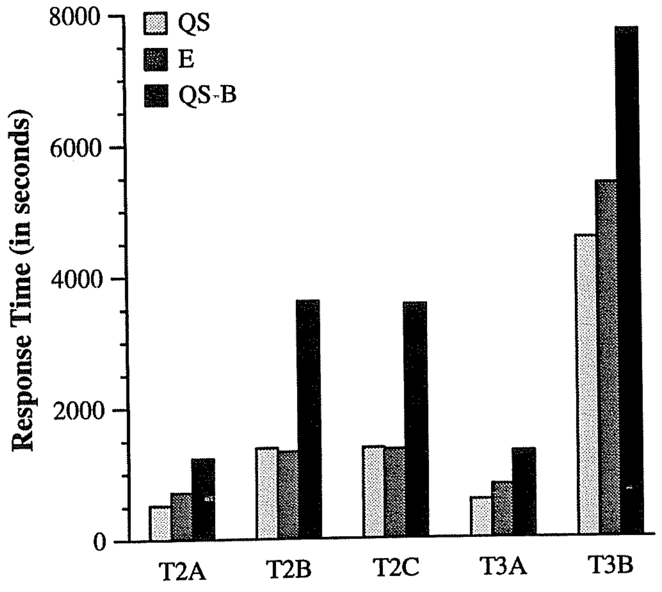


Figure 16. Medium Database, Traversal Cold Times.

### 5.5. Effect of Collisions

QuickStore always tries to assign a disk page to the virtual memory location that it last occupied when in memory. This section considers the effect on performance of relocating pages at different memory addresses when they are faulted into memory. This increases faulting costs because pointers between persistent objects must be updated to reflect the new assignment of disk pages to virtual memory addresses. We consider two approaches to dealing with page relocations. The first approach updates or swizzles pointers that need to be modified when pages are faulted into memory, but these changes are not written back to the database. This implies that the changes will have to be made again if the same data is accessed in subsequent transactions. We refer to this system as QS-CR (QuickStore with continual relocation). The second approach commits the changed mapping to the database. This approach is more costly initially, but may be able to avoid further relocations in the future. This approach also has the disadvantage that it can turn a read-only transaction into an update transaction. We refer to this approach as QS-OR (QuickStore with one-time relocation).

Figure 17 presents the results for T1 run on the small database when the percentage of pages that are relocated in memory is varied from 0 to 100%. The pages that were relocated in the experiment were picked at random. Figure 17 shows that when the number of relocations is small (5%), the performance of the systems is not significantly affected. However, when the relocation percentage is 20%, QS-OR is 25% slower than when no relocations occur. The difference in performance between QS-OR and QS-CR at this point is also about 25%. The performance of QS-CR slows by 7% and 38% when the percentage of relocated pages is 50% and 100%, respectively, while the corresponding decrease in performance for QS-

OR is 67% and 116%, respectively. QS-OR is much slower than QS-CR when all pages are relocated since it must commit updates for all of the pages in the database.

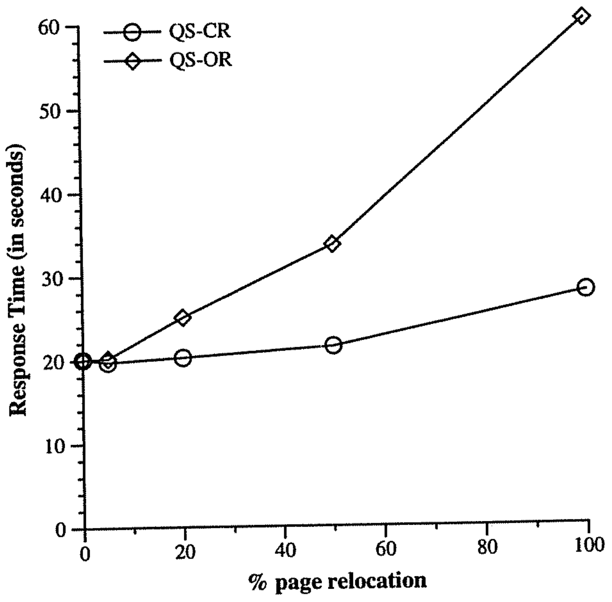


Figure 17. T1, vary % of relocations.

## 6. Conclusions

This paper compared the performance of QuickStore, which uses memory-mapping techniques to implement persistence, to the performance of E, a persistent version of C++ that uses a software interpreter. The OO7 benchmark was used as a basis for comparing the performance of the two systems. The results of the study give a clear and accurate picture of the tradeoffs between the two approaches, which we summarize below.

The results of the cold traversal experiments showed that when object accesses are clustered (T1), QuickStore has the best performance. This is because object sizes in QuickStore are smaller than in E, due to the different schemes used by the systems to represent pointers on disk. QuickStore's smaller object size allows it to perform significantly fewer disk I/O operations to read the same number of objects when accesses were clustered. When object accesses were unclustered (i.e. in T6, Q1, Q2, etc.), the performance of QuickStore was comparable or worse than the performance of E. The reason for this change, relative to the clustered case, was that there was less difference between the systems in the number of pages faulted into memory per object access during the unclustered experiments. This exposed the fact that QuickStore has higher per faults costs than E. In addition, there were some cases (T7 and T9) when QuickStore always had lower performance due to higher faulting costs. The lower performance for QuickStore during T7 was influenced by the cost of reading a relatively large amount of mapping information to support the memory-mapping scheme that it uses.

The higher faulting costs for the memory-mapping scheme were also highlighted by the performance of QS-B (QuickStore with big objects). Except during the experiments that scanned large objects (T8), QS-B's performance was lower than E's during the read-only cold experiments. The memory-mapped schemes had better performance than E when large objects were accessed because large object accesses require significantly more CPU work with the software approach. This additional cost caused E to be slower even in the cold case.

For the traversals where faulting costs were examined in detail, it was shown that the average cost per fault for QuickStore was roughly 20% higher than for E. The largest component of the additional faulting cost for the memory mapping scheme was the time required to read mapping information from disk. This comprised 4% of the average cost per fault. The detailed cost analysis also showed that the overhead for handling page protection faults and manipulating page access protections were each 3%. The smallest component of the faulting cost for QuickStore was the CPU cost for swizzling pointers. This was just 1% of the average cost per fault.

The performance of QuickStore was generally better than E when updates were performed. The results of the update experiments showed, however, that the page-based diffing scheme used by QuickStore to generate log records was more expensive when updates were sparse and when the update activity was heavy enough to cause log records to be generated before transaction commit. QuickStore performed better relative to E when a higher percentage of objects were updated on each page, as QuickStore copied and diffed fewer objects unnecessarily in this case. The detailed times for the update experiments showed that the cost of diffing was ranged from 7 to 12 milliseconds per page for the OO7 update operations.

The hot results helped to quantify the performance advantage of the memory-mapped scheme when working on in-memory objects. In some cases (T1) the difference in performance between QuickStore and E was only 23%, while in others (T6) QuickStore was over 3 times faster than E. This showed how quickly the performance of the systems converged when a small amount of additional work was performed. The results also showed that E was significantly slower than QuickStore when doing in-memory work on large objects since this required all accesses to be handled by the E interpreter.

Finally, we also examined the performance of QuickStore when pages of objects must be relocated in memory, which increases the amount of swizzling work performed by QuickStore. When the percentage of pages relocated was small, the performance of the systems did not noticeably worsen. However, a high percentage of relocations did have a noticeable effect on overall performance, particularly when the new mapping tables were written back to the database. The best approach overall appeared to be to avoid writing the changed mapping tables to disk and to continually relocate pages in memory since the negative impact on performance of this approach was small.

## References

[Ball92] T. Ball, J. Larus, "Optimally Profiling and Tracing Programs", *POPL 1992*, pp. 59-70, January 1992.

[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", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Washington, DC, May 1993.

[DeWitt90] D. DeWitt, P. Futtersack, D. Maier, F. Velez, "A Study of Three Alternative Workstation-Server Architectures for Object-Oriented Database Systems", *Proceedings of the 16th International Conferece on Very Large Data Bases*, Brisbane, Australia, August, 1990.

[Frank92] M. Franklin, M. Zwilling, C. Tan, M. Carey, D. DeWitt, "Crash Recovery in Client-Server EXODUS", *Proceedings of the ACM SIGMOD International Conference on Management of Data*, California, 1992.

[Hoski93a] A. Hosking, J. E. B. Moss, "Object Fault Handling for Persistent Programming Languages: A Performance Evaluation", *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pp. 288-303, 1993.

[Hoski93b] A. Hosking, E. Brown, J. Moss, "Update Logging in Persistent Programming Languages: A Comparative Performance Evaluation", *Proceedings of the 19th International Conferece on Very Large Data Bases*, Dublin, Ireland, 1993.

[Jagad94] H. Jagadish, D. Lieuwen, R. Rastogi, A. Silberschatz, "Dali: A High Performance Main Memory Storage Manager", to appear in *Proceedings of the 20th International Conferece on Very Large Data Bases*, Santiago, Chile, September 12-15, 1994.

[Khosh86] S. Khoshafian, G. Copeland, "Object Identity", *Proceedings of the ACM Conference on Object-Oriented Programming Systems and Languages (OOPSLA)*, pages 408-416, November 1986.

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

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

[Objec90] Object Design, Inc., "ObjectStore User Guide", Release 1.0, October 1990.

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

[Schuh90] D. Schuh, M. Carey, and D. Dewitt, "Persistence in E Revisited---Implementation Experiences", *Proceedings of the Fourth International Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Shek90] E. Shekita and M. Zwilling, "Cricket: A Mapped Persistent Object Store", *Proceedings of the Fourth International Wor kshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Wilso90] Paul R. Wilson, "Pointer Swizzling at Page Fault Time: Efficiently Supporting Huge Address Spaces on Standard Hardware", Technical Report UIC-EECS-90-6, University of Illinois at Chicago, December 1990.