A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies

Seth J. White
David J. DeWitt

Technical Report #1093

June 1992

A Performance Study of Alternative Object Faulting and Pointer Swizzling Strategies

Seth J. White David J. Dewitt

Computer Sciences Department
University of Wisconsin
Madison, WI 53706
{white,dewitt}@cs.wisc.edu

Abstract

This paper presents a portable, efficient method for accessing memory resident persistent objects in virtual memory in the context of the E programming language. Under the approach, objects are copied from the buffer pool of the underlying object manager into virtual memory on demand, as they are accessed by an E program. The cumulative effects of updates to a persistent object are then propagated back to the object manager via a single write operation at the end of each transaction. The method incorporates a comprehensive pointer swizzling mechanism to enhance performance. Swizzling is done a pointer-at-a-time and software checks are used to detect the use of swizzled pointers. The paper also presents the results of a performance study comparing the method presented here with several alternative software architectures including ObjectStore V1.2, a commercially available OODBMS. The results highlight the tradeoffs between providing software vs. memory-mapped support for pointer swizzling and quantify the effects of pointer swizzling on overall performance. In addition, the significant performance impact of pointer swizzling on the generation of recovery information is examined. The experimental results show that in many situations a software approach can outperform the memory-mapped approach.

1. Introduction

E is a persistent programming language [Rich89, Rich90] that was originally designed to ease the implementation of data-intensive software applications, such as database management systems, that require access to huge amounts of persistent data. The current implementation of E (E 2.0) uses an interpreter, the E Persistent Virtual Machine (EPVM 1.0), to coordinate access to persistent data [Schuh90] that is stored using the EXODUS Storage Manager [Carey89a, Carey89b]. Under the approach taken by EPVM 1.0, memory resident persistent objects are cached in the buffer pool of the EXODUS Storage Manager (ESM) and persistent objects are accessed in-place. In addition, EPVM 1.0 provides support for a limited form of pointer swizzling.

This paper introduces an alternative implementation of EPVM (EPVM 2.0) that is targeted at CAD environments. One common example of a CAD application is a design tool that loads an engineering design into main memory, repeatedly traverses the design while performing some computation over it, and then saves the design again on secondary storage. An important property of design applications is that they perform a considerable amount of focused work on in-memory persistent objects. A major fraction of this work involves the manipulation of persistent objects via pointers.

The basic approach employed by EPVM 2.0 is to maintain a cache in virtual memory of the set of persistent objects that have been accessed by an E program. Objects are copied from the ESM buffer pool and inserted into the cache as they are accessed by a program. In addition, the cumulative effects of updates to a persistent object are propagated back to ESM via a single write operation when a transaction commits (finishes execution). The cache supports a comprehensive pointer swizzling scheme that swizzles inter-object pointer references, i.e. converts pointers from object identifiers (OIDs) to direct memory pointers. Pointers are swizzled one-at-a-time as they are used by an E program. If the volume of data accessed by an individual transaction exceeds the size of real memory, objects are swapped to disk in their swizzled format by the virtual memory subsystem.

To help evaluate the effectiveness of the design of EPVM 2.0 this paper presents the results of a number of performance experiments that were conducted using the OO1 benchmark [Catte91]. The experiments compare EPVM 2.0 with three alternative software architectures. The first of these is ObjectStore V1.2 [Objec90], a commercially available object-oriented DBMS. ObjectStore uses a memory-mapped approach to support pointer swizzling and fault objects into main memory. The second architecture is represented by EPVM 1.0 which supports only a limited form of pointer swizzling. The third architecture does not support pointer swizzling, and corresponds to using a conventional non-persistent programming language, i.e. C++, to call ESM directly.

The experimental results illustrate the tradeoffs between the different implementations of object faulting and pointer swizzling (including doing no swizzling) and examine the impact of the different schemes on the generation of recovery information. In the case of EPVM 2.0, alternative ways of managing the migration of persistent objects from the ESM buffer pool into virtual memory are also examined. All of the systems included in the study are based on a client/server architecture and feature full support for transactions, concurrency control, and recovery. The client/server version of ESM [Frank92, Exodu92] was used to store persistent data for the experiments based on EPVM 2.0, EPVM 1.0, and C++.

Proceedings of the 18th VLDB Conference
Vancouver, British Columbia, Canada 1992

This research was funded by the Defense Advanced Research Projects Agency under contract DAAB07-92-C-Q508.

The remainder of the paper is organized as follows. Section 2 discusses related work on object faulting and pointer swizzling. Section 3 presents a detailed description of the implementation of EPVM 2.0. Section 4 describes the benchmark experiments. Section 5 presents the performance results. Section 6 contains some conclusions and proposals for future work.

2. Related Work

Previous approaches to pointer swizzling can be roughly divided into two groups; those that use memory mapping techniques, similar to virtual memory, and those that use software checks to detect accesses to nonresident objects. Some early work on software implementations of pointer swizzling was done as part of an implementation of PS-Algol [Atkin83, Cock84]. This approach also used pointer dereferences to trigger the transfer of objects from secondary storage into main memory.

[Moss90] presents a more recent study of software swizzling techniques, and also examines the issue of storing persistent objects in the buffer pool of the object manager versus copying them into virtual memory. [Moss90] takes an object-at-a-time approach to swizzling in which objects that are in memory are classified as either swizzled or unswizzled. Under this approach, all pointers in an unswizzled object are swizzled immediately upon the first use of the object. This causes the objects that the pointers reference to be faulted into memory and marked as unswizzled. Finally, the initial object is marked as swizzled.

One advantage of this approach over that of [Wilso90] (see below) is that it should generally perform less unnecessary swizzling and unswizzling work. A disadvantage, however, is that objects that are not accessed by a program can be faulted into memory by the swizzling mechanism, resulting in unnecessary I/O operations. In particular, unswizzled objects, while they are memory resident, have by definition not been referenced.

A restricted form of pointer swizzling is supported by EPVM 1.0 [Schuh90]. Since it maintains memory resident objects in the ESM buffer pool, swizzling inter-object pointer references is difficult to implement efficiently. Hence, only local program variables that are pointers to persistent objects are swizzled.

The advantage of this approach is that it allows objects to be written back to disk in the presence of swizzled pointers. In general, this is very hard to do efficiently when using a software approach, because all swizzled pointers to an object must be found and unswizzled, before the memory space occupied by the object can be reused. However, the E compiler stores local pointers to persistent objects on a special "pointer stack" that is maintained in parallel with the regular procedure activation stack. When space in the buffer pool needs to be reclaimed, EPVM 1.0 scans the pointer stack and unswizzles any swizzled pointers that it contains. This ensures that there are no dangling references to objects that are no longer resident in memory.

A pointer swizzling scheme based on virtual memory techniques is described in [Wilso90]. A similar approach is used in Object Design's ObjectStore [Objec90, Lamb91]. The basic idea presented in [Wilso90] is to allocate virtual memory addresses for pages containing persistent data one step ahead of a program's actual usage of the pages. When a program first attempts to access a page, a virtual memory page fault occurs. This fault is intercepted by the underlying object manager which then loads the page into its preassigned location in memory.

One advantage of this method of swizzling is that programs only see regular virtual memory pointers, allowing accesses to persistent objects to occur at memory speeds. In addition, the same compiled code can be used to access both persistent and non-persistent objects. Objects that span multiple pages in virtual memory can be handled transparently as long as sufficient contiguous virtual memory address space can be reserved for the entire object.

A disadvantage of the basic approach described in [Wilso90] is that programs may incur unnecessary swizzling and unswizzling overhead. This is because swizzling and unswizzling are done at the granularity of individual pages, and it is unlikely that most programs will use all of the pointers located on each page. [Objec90] describes an extension of the basic technique that can avoid this problem by eliminating the need to swizzle and unswizzle pointers in many cases. In effect, pointers are always stored in their swizzled format in [Objec90].

3. EPVM 2.0 Design Concepts

3.1. Object Caching

As mentioned in Section 1, ESM is used to provide disk storage for the persistent objects that are accessible to an E program. EPVM 2.0 copies objects from the ESM client buffer pool into virtual memory as they are accessed. Separate schemes are used to cache objects that are smaller than a disk page, hereafter referred to as small objects, and large objects that can span any number of pages on disk. Small objects are copied from the ESM client buffer pool in their entirety and stored in individual contiguous regions of virtual memory. A bitmap, which is appended to the beginning of each region, is used to record the locations of all swizzled pointers contained in the small object.

Large objects are cached a page-at-a-time in units of 8K bytes. Individual large object pages are cached on demand, so that only the pages that have been referenced are cached. Each cached page has appended to it a bitmap that keeps track of all swizzled pointers on the page. Different pages of a large object are not necessarily stored contiguously in virtual memory. This fact has important implications for pointer swizzling since it essentially means that pointers to large objects cannot be swizzled because accesses to large objects through such pointers can span page boundaries.

Objects that have been cached in virtual memory are organized using a hash table on each object's identifier (OID). Entries in this hash table are pointers to object descriptors (see Figure 1). In the case of a small object, the object descriptor contains a single pointer to the copy of the object in virtual memory. Paired with this pointer are a low and a high byte count that are used to keep track of the range of modified bytes in the object. For each update of a small object the range is expanded by decrementing and incrementing the low and high byte counts respectively, as needed. Note that this method of keeping track of the modified portion of an object works best when there is some locality of updates to objects. The range of modified bytes together with the bitmap stored at the beginning of the object determines the portion of the object that must be written to disk, and the subset of swizzled pointers in the object that must be unswizzled when a transaction completes.

The object descriptor of a large object contains an array of pointers to pages of the large object. Each large object page has associated with it a low and high byte count that are used to keep

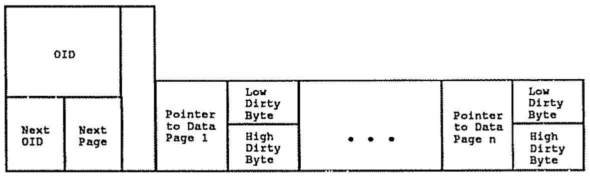

Figure 1. An object descriptor for a large object.

track of the modified portion of the page, in a manner analogous to that used for small objects.

Figure 2 shows an example of small and large objects that have been cached. The small objects' descriptors each contain a single pointer to their respective objects, while the large object's descriptor contains pointers to two pages of the large object. Note that these pointers point to the beginning of the object/page and not to the corresponding bitmap. In Figure 2, the last page of the large object has not been referenced, so the object descriptor points only to the first two pages.

The object descriptors of small objects are organized in a second hash table according to the disk page on which their corresponding objects reside. All small objects that reside on the same disk page are in the same overflow chain. This allows the effects of updates to all objects on the same page to be propagated back to the ESM buffer pool at the same time when a transaction commits. Large objects are kept in a separate linked list that is traversed at the end of a transaction to write back dirty portions of large object pages.

Figure 2 depicts two small objects, residing on the same disk page. The objects are linked together by the next page pointers in their object descriptors. Of course, it is possible that objects from different disk pages may be found in the same overflow chain if their page numbers hash to the same value. In practice, however, the low cost of this strategy, plus the fact that such collisions are rare, allows it to perform well.

3.2. Pointer Swizzling in EPVM 2.0

Since all persistent objects are accessed through pointers in E, it is important to provide an efficient mapping from E pointers to persistent objects. When a pointer is dereferenced by an E program it may be in one of two states: **unswizzled**, in which case it contains the value of an object identifier (OID); or **swizzled**, meaning that it contains a direct memory pointer. Dereferencing an unswizzled pointer basically incurs the cost overhead of a lookup in the OID hash table in order to obtain a pointer to the referenced object. Dereferencing a swizzled pointer avoids this cost as a swizzled pointer contains a direct memory pointer to the object it references.

While the difference in dereferencing cost may seem small, it is important to remember that tens or hundreds of thousands of pointer dereferences can occur during the execution of a program. Hence, the potential savings offered by pointer swizzling is indeed large. A key assumption of any pointer swizzling scheme is that pointers are used often enough on average to justify the costs of doing the pointer swizzling.

EPVM 2.0 supports a pointer swizzling scheme that converts pointers from OID form to direct memory pointers incrementally during program execution. The goal is to quickly and cheaply convert pointers to swizzled format so that a program "sees" only swizzled pointers during the majority of the time it is executing.

Software checks are used to distinguish swizzled and unswizzled pointers. This seems reasonable since the price of such checks should be a very small part of overall program execution time; a fact that has been independently confirmed in [Moss90]. Furthermore, it is possible to do standard kinds of compiler optimizations to eliminate checks from a program (though the E compiler currently does not do this). The software approach combines efficiency with portability, and provides a flexible environment for conducting further research. The swizzling scheme used in EPVM 2.0 is further characterized by the fact that it swizzles pointers one-at-a-time, as opposed to the approach described in [Wilso90] which swizzles a page-at-a-time, and [Moss90] which swizzles

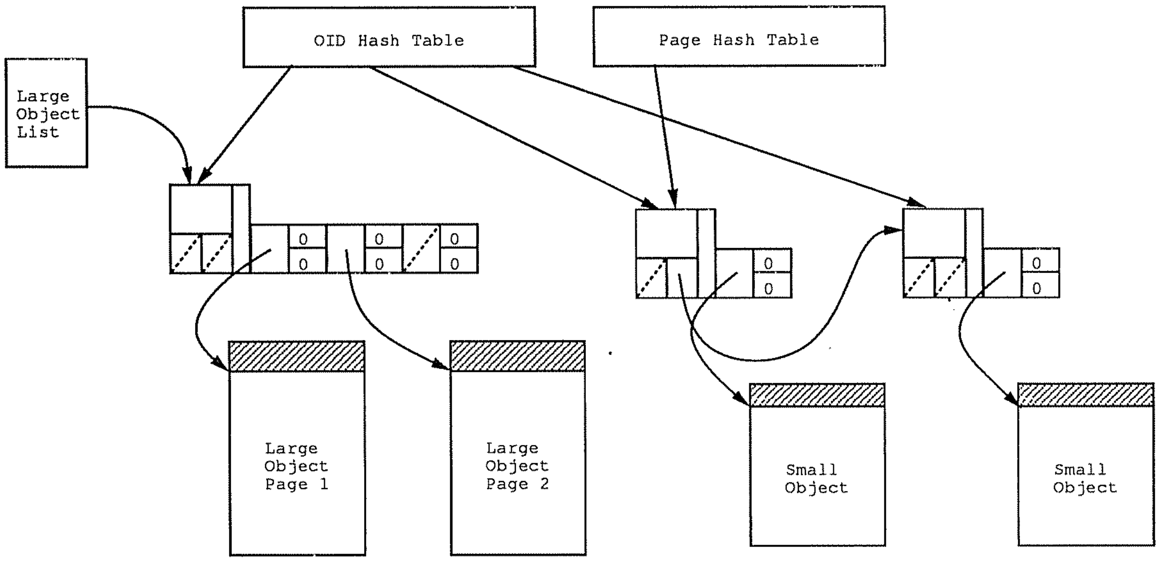

Figure 2. Object cache containing small and large objects.

pointers at the granularity of objects. The type of swizzling scheme used by EPVM 2.0 is referred to as an 'edge marking' scheme in [Moss90].

Implementation Strategy

Since pointers are swizzled dynamically during program execution, the key decision that must be made is when during execution to actually do the swizzling. One possibility is to swizzle pointers when they are dereferenced. To see how this is done, consider in detail what happens when an unswizzled pointer is dereferenced during the execution of an E program. First, the memory address of the pointer is passed to an EPVM function that performs a lookup in the OID hash table using the value of the OID contained in the pointer. An E pointer is composed of a 12 byte OID (volume id: 2 bytes, page id: 4 bytes, slot number: 2 bytes, unique field: 4 bytes) and a 4 byte offset field. If the referenced object is not found, then it must be obtained from the EXODUS Storage Manager (ESM), possibly causing some I/O to be done, copied into virtual memory, and inserted into the cache. The pointer may then be swizzled since the virtual memory addresses of both the pointer and the object it references are known. This type of swizzling will be referred to as **swizzling upon dereference**.

The advantage of this scheme is that pointers that are never dereferenced are never swizzled, so the amount of unnecessary swizzling work is minimized. Furthermore, since only pointers to referenced objects are swizzled, unnecessary I/O operations are avoided. Swizzling upon dereference does present a major problem, however. In particular, when a pointer is dereferenced, it has often already been copied into a temporary memory location, e.g. a local pointer variable somewhere on the activation stack. Swizzling upon dereference, therefore, fails to swizzle pointers between persistent objects and can, in effect, force programs that use these pointers to work with unswizzled pointers throughout their execution.

The approach used by EPVM 2.0 is to swizzle pointers within objects as they are "discovered", i.e. when the location of the pointer becomes known. We shall call this type of swizzling **swizzling upon discovery**. A pointer within an object may be discovered when its value is assigned to another pointer, when it is involved in a comparison operation, or in a number of other ways. In the context of EPVM, pointers are discovered as follows. First, EPVM is passed the persistent address of the pointer that is a candidate for swizzling. The contents of this persistent address are then used to locate the object containing the candidate pointer in the cache. (Note that this initial step may involve actually caching the object that contains the pointer to be swizzled.) Once the virtual memory address of the candidate pointer is known, its contents can be inspected, and if it is not already swizzled, used to perform a lookup in the OID hash table to find the object that it references. If the object denoted by the candidate pointer is found in the cache, then the candidate pointer is swizzled. Note that this swizzling scheme solves the major problem associated with swizzling upon dereference since pointers within persistent objects are swizzled.

Next, consider the case when the object referenced by the candidate pointer is not found in the cache. One alternative would be to go ahead and cache the object. This "eager" approach could result in unnecessary I/O operations, however, since the object referenced by the candidate pointer may in fact not be needed by the program. For example, consider a persistent collection object that is used to store pointers to objects of some other class. The routine that implements deletion from the collection may need to compare the value of a pointer being deleted with an arbitrary number of pointers in the collection. Each of these comparisons discovers a pointer contained in the collection object, so the deletion operation could fault in a large number of objects if eager swizzling upon discovery were used. Since a swizzling scheme should avoid causing unnecessary I/O operations, EPVM 2.0 takes a lazy approach in which it does not swizzle the candidate pointer when the object it references is not already cached.

In summary, the swizzling scheme used by EPVM 2.0 uses only pointer dereferences to fault objects into the cache. Then, once an object is in the cache, pointers that reference the object are swizzled when their locations are discovered. Pointers to an object that are discovered before the object has been referenced are not immediately swizzled. Lastly, note that swizzling on discovery restricts swizzling activity to those pointers that are actually used by a program, so programs that do not use many pointers do not have to pay a big price in terms of swizzling overhead. Also, only those objects actually needed by the program are cached, so no extra I/O activity results from swizzling.

The example in Figure 3 is designed to illustrate the differences between swizzling upon dereference and the eager and lazy variations of swizzling upon discovery that were described above. The function *TotalCost* traverses an assembly of persistent part objects (which is assumed to form a tree for simplicity) in depth first order and calculates the total cost of the assembly. Each part object contains a cost field and three pointers to subparts. We also assume that the collection of part objects is as shown in Figure 4a, i.e. there are eight part objects in the collection whose OIDs are represented by the letters A to H, and the objects form a tree of height two. Figure 4a depicts the format of the part objects when they are stored on disk and the connections between parts are represented by OIDs.

Note that the only pointer that is actually dereferenced in the example is *root*; a transient, local pointer variable. If swizzling upon dereference is used while executing *TotalCost*, then only *root* will be swizzled, and the *subPart* pointers contained within part objects will always remain in their unswizzled form. This implies that repeated traversals of the parts assembly will always encounter unswizzled pointers, i.e. the assembly will remain in the format shown in Figure 4a.

Pointers located within part objects are discovered by the *TotalCost* function when the expression *root->subPart[i]* is evaluated

```
dbstruct part {              // structure of a part
  dbint pCost;
  part *subPart[3];
};

int TotalCost(part *root, int depth) {
  int totCost = 0;
  for (int i = 0; i < 3; i++)
    if (root->subPart[i] && depth)
      totCost += TotalCost(root->subPart[i], depth-1);
  totCost += root->pCost;
  return totCost;
}
```

Figure 3. Example E function.

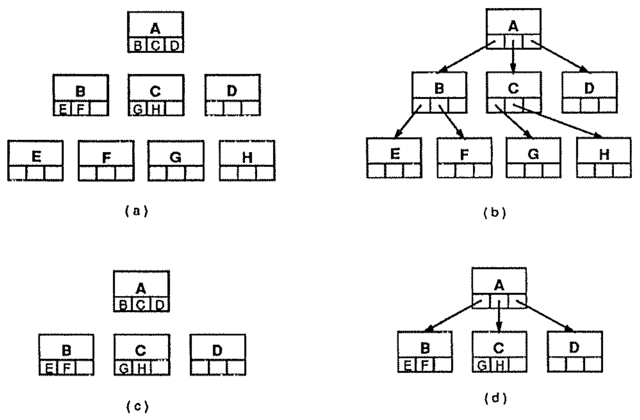

Figure 4. Different representations of a collection of objects.

in line 8. Note that whenever a part object is visited, all three of the *subPart* pointers located in the object are discovered. Suppose that the collection of parts shown in Figure 4a is repeatedly traversed using the *TotalCost* function, beginning at object A, to a depth of 1. If the eager implementation of swizzling upon discovery is used, then all three subparts of each leaf node in the subtree visited by *TotalCost* are cached. Figure 4b shows the basic structure of the part assembly in memory after the first traversal of the parts using this method. In this example, a total of eight part objects are read from disk and cached, which is double the number of objects actually needed.

Next, consider how the swizzling scheme used in EPVM 2.0 behaves when doing the same traversal. After the first traversal of the collection, the part objects that have been cached will appear as in Figure 4c. Note that all of the objects accessed by the program have been cached, but that the pointers among the objects are still in their unswizzled OID form. In this case, none of the *subPart* pointers have been swizzled since, when they are discovered on line 8 during the first traversal, the objects that they reference are not yet in the cache. The objects are faulted into the cache during the first traversal when the pointer *root* is dereferenced on line 8. After a second traversal, the structure of the collection is as in Figure 4d. Note that all of the pointers between objects that have been visited by the program are swizzled, and that further traversals of the collection will dereference only swizzled pointers.

4. Performance Experiments

The performance experiments were done using the traversal portion of the OO1 Benchmark [Catte91]. The traversal portion involves repeatedly traversing a collection of part objects, beginning at a randomly selected part, in a depth-first fashion to a depth of 7 levels. Each of the individual traversals is referred to as an iteration of the benchmark. As each part is visited during an iteration a simple function is called with four values in the part object as parameters. In addition, the ability to update part objects was added, so that each time a part object is visited, a simple update can be performed with some fixed probability. The update operation was defined as incrementing two 4-byte integer fields contained in the part object.

A total of eight different software versions were evaluated. These software versions can be classified into four basic architectures (see Section 4.1). The experiments compare the performance of the different architectures and investigate the relative performance of several versions of the approach used by EPVM 2.0. The usefulness of pointer swizzling is also evaluated. A number of experiments that vary the frequency with which updates are performed on objects were also conducted. This was done to access the impact that the different swizzling approaches have on the generation of recovery information. All of the architectures that are examined offer equivalent transaction facilities, i.e page level locking, atomicity of transactions, and transaction rollback. Some architectures attempt to batch updates of objects together and generate recovery information for all of the updates made to an object at the end of the transaction, while other architectures take the traditional database approach of generating log records for each individual update. Both approaches have important implications for systems that do redo/undo logging. The experiments also compare the different software versions using a small database that fits into main memory and a large database that represents a working set size that is bigger than main memory[Catte91].

4.1. Software Versions

The first architecture, which is shown in Figure 5, results when a conventional non-persistent programming language, i.e. C++, is used to call ESM directly. This approach accesses objects in the client buffer pool of ESM using a procedural interface. The routines that make up the ESM interface are linked with the application at compile time and the client buffer pool is located in the application's private address space. In all of the experiments, the server process was located on a separate machine that was connected to the client over a network.

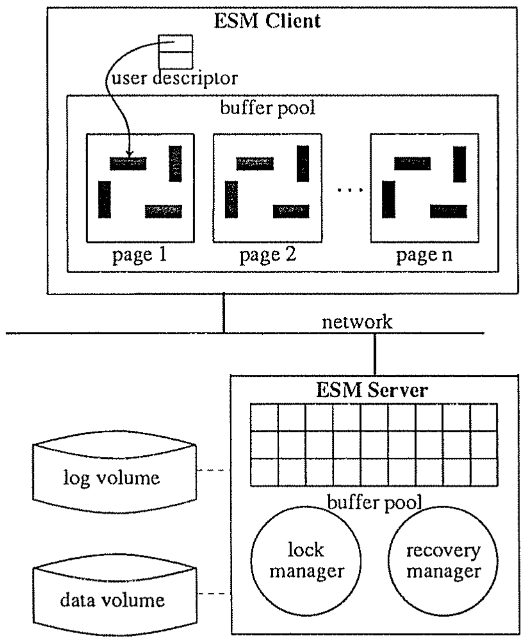

Figure 5. Architecture 1.

Accesses occur within a particular transaction, and take place during a visit to an object as follows. When an application first wants to read a value contained in an object, it calls an ESM interface function. The interface function requests the page containing the object from the server if necessary (possibly causing the server to perform some I/O on its behalf), and pins the object in the client buffer pool. Next, the interface function returns a data structure to the application, known as a user descriptor [Carey89a], that contains a pointer to the object. The application can then read values in the object any number of times by following the pointer contained in the user descriptor.

Each time the application wants to update a portion of an object it must call an interface function, passing in (among other things) the new value, and a user descriptor pointing to the object as parameters. The update function then updates the specified portion of the object in the client buffer pool and generates a log record for the update using the old value contained in the object and the new value which was passed as a parameter. When an application is finished visiting an object it calls an ESM function to unpin the object in the client buffer pool. If all objects on the page are unpinned at this point, the page becomes a candidate for replacement by the client buffer manager.

Note that in this architecture, a pin/unpin sequence of operations on an object generally takes place during a very short period of time relative to the life of a program, often during a single invocation of a function. This causes an object to be pinned and unpinned multiple times if it is visited more than once by a program. In addition, each update operation causes a log record to be generated.

In the current release of ESM, data pages are cached in the client's buffer pool between transactions. However, the client must communicate with the server to reacquire locks for cached pages that are accessed in succeeding transactions. Transaction commit involves shipping dirty data pages and log pages back to the server, writing log pages to disk, and releasing locks [Frank92]. In the future, ESM will support "callbacks" from the server to the client. This will allow inter-transaction caching of locks at the client and eliminate the need to ship dirty data pages back to the server during transaction commit. No pointer swizzling is done in this architecture. A single software version based on this architecture was used (referred to as CESM). The size of the ESM client and server buffer pools was 5 megabytes.

The second architecture represents the approach taken by EPVM 1.0 [Schuh90]. Figure 6 shows the client portion of this architecture (The server portion is identical to the server shown in Figure 5). EPVM 1.0 avoids calls to the storage manager by maintaining a cache of worthy objects in the ESM client buffer pool. Objects are accessed in the following way. The first time that an object is needed by an application, EPVM 1.0 calls an ESM interface function that pins the object in the client buffer pool, and returns a user descriptor through which the object can be referenced. This may involve communication between the client and the server and the server may in turn perform some I/O on behalf of the client. Next, EPVM 1.0 creates an entry for the object in a hash table based on the object's OID. The hash table maintains a mapping from OIDs to user descriptors that remains valid until either the client buffer pool becomes full or program execution completes.

Objects that are cached in the ESM buffer pool are accessed by doing a lookup in the OID hash table, or by following a swizzled pointer since EPVM 1.0 supports a limited form of pointer swizzling (see Section 2). Updates to objects, however, require EPVM

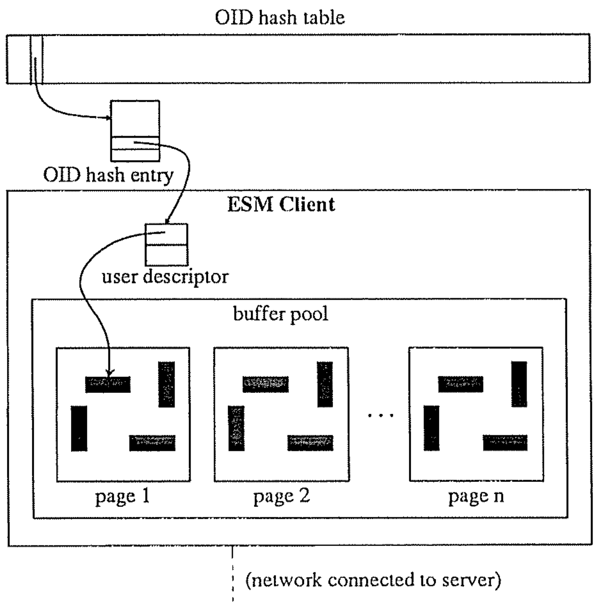

Figure 6. Architecture 2.

1.0 to invoke a storage manager interface function. The interface function updates the object in the buffer pool and generates a log record for the update. Transaction commit requires that EPVM 1.0 scan the OID hash table and unpin all objects, in addition to the usual operations performed by ESM to commit a transaction.

In order to measure the effectiveness of the swizzling technique employed by EPVM 1.0, experiments were performed using two versions of this architecture, the first version had the limited form of pointer swizzling enabled, and the second had swizzling turned off. These versions will be referred to as EPVM1 and EPVM1-NO, respectively. Again, 5 megabyte client and server buffer pools were used.

The third architecture investigated corresponds to the approach taken by EPVM 2.0. Let us briefly review how objects are accessed with this architecture. When an object is first needed by an application program, EPVM 2.0 calls ESM on behalf of the application. ESM then pins the object in the client buffer pool, as shown in Figure 5. Next, EPVM 2.0 uses the user descriptor returned by ESM to copy the object into virtual memory and EPVM 2.0 inserts the object into the cache in the manner depicted in Figure 2. EPVM 2.0 then calls ESM to unpin the object in the client buffer pool. All subsequent reads or updates of the object during the current transaction occur in the cache and are handled exclusively by EPVM 2.0.

During transaction commit EPVM 2.0 scans the page hash table and for each small object that has been updated, EPVM 2.0 calls ESM to pin the object in the client buffer pool and update the object. Note that this may involve communication between the client and the server if the page containing the object is no longer present in the client buffer pool. When ESM updates the object in the client buffer pool, the new value of the modified portion of the object and the old value located in the client buffer pool are used to generate a log record for the update. Updates of large objects are handled in a similar manner, the only difference being that

EPVM 2.0 invokes ESM once for each modified page of the large object.

The performance of two alternative ways of copying objects from the client buffer pool into virtual memory were examined. The first copies objects one-at-a-time from the client buffer pool into virtual memory while the other copies all of the objects on a page when the first object on the page is accessed. These two schemes shall be referred to as object caching and page caching respectively. The tradeoff between the two approaches is that object caching generally requires more interaction with the storage manager, i.e. one interaction per object, while page caching requires only one interaction per page, but has the potential to perform more copying.

Four versions of this architecture were investigated. Two used object caching. In order to study the effect of buffer pool size on object caching, the size of the client buffer pool for one version was set at 5 megabytes while the client buffer pool for the other was set at 1 megabyte (both used a 5 megabyte server buffer pool). These versions shall be referred to as OC5M and OC1M respectively. Both versions did pointer swizzling.

The third and fourth versions were designed to measure the benefit provided by the swizzling technique implemented in EPVM 2.0. Both versions do page caching and each was given a 1 megabyte client buffer pool to make the amount of memory that they used similar to the other versions. Again, a 5 megabyte server buffer pool was used for all of the experiments. The version referred to as PC1M does pointer swizzling, while the version labeled PC1M-NO does not.

The fourth architecture examined was that of ObjectStore V1.2[1] [Lamb91]. Like ESM, ObjectStore uses a client/server architecture in which both the client and server processes buffer recently accessed pages of objects. All interaction between the client and server in ObjectStore was set to take place at the granularity of individual pages, just as in ESM. ObjectStore features basically the same transaction facilities as ESM, i.e. recovery for updates in the event of client or server failure, page level locking, and transaction rollback.

ObjectStore also supports inter-transaction caching of persistent data in the client's main memory[Lamb91]. Callback messages are sent by the server to clients in order to maintain the coherence of cached data. This allows the ObjectStore client to cache locks between transactions as well as data pages. To efficiently support callbacks, the ObjectStore client is divided into two processes[Orens92]: a callback process, and an application process. When only a single client is connected with the server, the two-process architecture does not have a noticeable effect on performance since the application process communicates directly with the server to obtain data pages and locks on those pages.

The most important difference between ObjectStore and the architectures already mentioned is that ObjectStore uses a memory-mapping scheme, similar to virtual memory, to implement pointer swizzling and fault objects from secondary storage into main memory (see Section 2). Another important difference is that ObjectStore generates recovery information for updates of persistent data by logging entire dirty pages. Full page logging is used to implement recovery largely due to the fact that ObjectStore applications are allowed to update objects by dereferencing normal virtual memory pointers. Because of this, ObjectStore is not able to keep track of the modified portions of pages (or objects) as is done in EPVM 2.0. The amount of real memory available to the client for caching pages of objects during a single transaction is fixed. We used 5 megabyte client and server buffer pools for all of the experiments. This architecture will be referred to as OS.

[1]We have recently received ObjectStore V1.2.2 which is said by the manufacturer to offer improved performance. However, we lacked sufficient time to obtain reliable results using the new version, so the results presented in the paper are for ObjectStore V1.2.

4.2. Benchmark Database

For ESM, the small benchmark database [Catte91] consumed a total of 489 8K-byte disk pages (3.8 Mg) and consisted of a collection of 20,000 part objects (each object is an average of 176 bytes in size). Additionally, the Sun benchmark requires that objects be indexed, so the parts were indexed using an array of 20,000 object pointers (OIDs). An array of pointers was used instead of a B-tree index in order to keep performance differences due to differing B-tree implementations from influencing the results. The total size of the index for ESM was 320,000 bytes.

The small database, including the part index, required 422 8K pages (3.3 Mg) using ObjectStore. Each part object contains connections to three other part objects in the database. These connections were implemented using pointers in both systems. The database required more disk space when using ESM largely because of differences in the way that pointers to persistent data are stored by the two systems.

The large benchmark database is identical to the small database except that 125,000 part objects were used. The large database occupied 3,057 disk pages with an index whose size was 1.9 megabytes when using ESM. For ObjectStore the large database required 2,559 pages. 125,000 objects were used for the large database instead of 200,000 as specified in [Catte91] due to limitations in the amount of available swap space. Using 125,000 objects eliminated this problem while still providing a database that would not fit into the real memory of the workstations that were used.

4.3. Hardware Used

All experiments were performed using two identically configured SUN SPARCstation ELCs (approximately 20 mips). One was used as the client machine and the other was used as the server. The two machines were connected via a private Ethernet. Both machines had 24 megabytes of main memory. Data was stored at the server using 70 megabyte raw disk partitions located on separate SUN0207 disk drives. One partition was used for the transaction log and the other was used to store normal data. The virtual memory swap area on the client machine was also located on a SUN0207 and was 32 megabytes in size.

5. Benchmark Results

5.1. Small Database Results

This section contains the results of running several variations of the traversal portion of the OO1 benchmark using the small benchmark database of 20,000 objects. All of the experiments were repeated 3 times and then averaged to obtain the results that are shown. All times are listed in seconds.

Tables 1, 2, and 3 present the individual cold, warm, and hot iteration times when no updates are performed and the entire benchmark run is executed as a single transaction. The cold time is the execution time for the first iteration of the benchmark when no persistent data is cached in memory on either the client or server machines. The warm time is the execution time for the tenth iteration of the benchmark. The hot times were obtained by repeating the traversal done during the warm iteration, so that all of the objects were in memory and all swizzling was done prior to the beginning of the hot iteration. The number of I/O operations performed by the client during each iteration is also given. The times in Tables 1, 2, and 3 do not include the overhead for transaction begin and commit.

Table 1 compares one version from each of the four software architectures discussed in Section 4.1. The versions selected are generally comparable in the sense that each uses a similar amount of memory, though PC1M does use slightly more memory than the others. CESM has the best time in the cold iteration, but EPVM1 does almost as well. The small difference between CESM and EPVM1 is likely due to the overhead of inserting objects into the OID hash table for EPVM1. PC1M is 7% slower than EPVM1 due to the overhead of caching full pages of objects. OS does the worst during the cold iteration despite the fact that it performs the fewest I/O operations. Given our understanding of how OS works, we believe this is partially due to the overhead of mapping data into the client's address space.

The ordering of times for the warm iteration in Table 1 is just the reverse of that for the cold iteration. OS is much faster than the other versions in the warm iteration since it incurs essentially no overhead for accessing in-memory objects in this case. PC1M is next in terms of performance. PC1M is 33% faster than EPVM1 because EPVM1 incurs the overhead of inserting a large number of objects into the OID hash table while PC1M caches only 2 pages (80 objects). Since PC1M is more aggressive at caching than EPVM1, it has already cached the additional objects during previous iterations. CESM has the worst performance in the warm iteration due to the overhead of calling ESM for each object that is visited during the iteration.

Table 2 presents the results for each of the versions based on EPVM 2.0. OC1M has the worst performance during the cold iteration because its small client buffer pool size forces it to reread pages from the server. It may seem surprising that OC1M is only 10% slower than OC5M given that it performs 33% more I/O operations. This is due to the fact that the server buffer pool is large enough to hold all of the pages read by the client in this case and shipping pages from the server is much faster than reading them from disk. 0C5M is 6% faster than PC1M due to the overhead that PC1M incurs for copying full pages into virtual memory. The similarity of PC1M and PC1M-NO shows that there is essentially no advantage or disadvantage to doing swizzling during the cold iteration.

In the warm iteration, Table 2 shows that PC1M has the best performance. PC1M and PC1M-NO do better than the object caching versions during the warm iteration because many more objects are being cached than are pages. More precisely, 2 pages are cached during the warm iteration by the page caching versions, while 1097 objects are cached by the object caching versions. PC1M caches fewer objects in the warm iteration because it has already cached the additional objects during previous iterations. This accounts for the somewhat strange fact that PC1M-NO (which does page caching and no swizzling) is 40% faster than OC5M (which does full swizzling). OC1M continues to reread pages from the server in the warm iteration and consequently has the worst performance. Turning to swizzling, in the warm case Table 2 shows that swizzling provides a 12% reduction in execution time for page caching. The times for EPVM1-NO are not shown in Tables 1 and 2. There was essentially no difference between EPVM1 and EPVM1-NO in the cold iteration for this experiment. In the warm iteration, swizzling made EPVM1 8% faster than EPVM1-NO.

The hot times in Table 3 represent the asymptotic behavior of each of the versions, when no further conversion or copying of in-memory objects is taking place. An additional version, labeled C, has been added to Table 3. C represents an implementation of the benchmark coded in non-persistent C++ using transient in-memory objects. C represents the best performance that a persistent system could hope to achieve in the hot case.

We first examine architectural differences. OS does the best during the hot iteration. The fact that the performance of OS is identical to C shows that the memory-mapped architecture of OS imposes no additional overhead in the hot case. OS is 33% faster than PC1M because of the overhead for swizzle checks and EPVM 2.0 function calls that PC1M incurs. In addition, the fact that pointers to persistent objects in E are 16 bytes long as opposed to 4 bytes in OS further slows the performance of PC1M.

EPVM1 is third in terms of performance and is 21% slower than PC1M. This is because EPVM1 does not swizzle pointers

Traversal without updates				
Version	Cold	I/Os	Warm	I/Os
CESM	10.586	325	0.285	2
EPVM1	10.655	327	0.180	2
PC1M	11.386	327	0.120	2
OS	12.530	217	0.066	1

Table 1. Single transaction without updates (times are in seconds).

Traversal without updates				
Version	Cold	I/Os	Warm	I/Os
OC5M	10.750	327	0.227	2
OC1M	11.799	434	1.979	171
PC1M	11.386	327	0.120	2
PC1M-NO	11.384	327	0.136	2

Table 2. Single transaction without updates (times are in seconds).

Traversal without updates		
Version	Hot	Hot w/o random
C	0.039	0.005
OS	0.039	0.005
PC1M	0.058	0.024
PC1M-NO	0.078	0.044
EPVM1	0.074	0.040
EPVM1-NO	0.082	0.048
CESM	0.230	0.196

Table 3. Single transaction without updates (times are in seconds).

between persistent objects and also because of the extra level of indirection imposed upon it by user descriptors. CESM has the worst performance in the hot iteration. Its hot time is approximately 3 times that of EPVM1 and nearly 6 times that of OS. This is due to the fact that CESM calls ESM to pin and unpin each object that is visited during the iteration. OC5M and OC1M were identical to PC1M in the hot iteration, so they are not shown. Comparing PC1M with PC1M-NO, we see that swizzling has improved performance by 26% in the hot case for page caching while swizzling makes a difference of just 10% for EPVM 1.0.

It seemed surprising that in the hot column of Table 3, OS is only 33% faster than PC1M. Upon closer inspection of the benchmark implementation, it was noticed that the Unix function *random* was being called during each visit of a part object as part of the overhead for determining whether or not to perform an update. The last column of Table 3 shows the results for the hot traversal when the overhead for calling *random* is removed. Note that OS now has approximately 5 times the performance of PC1M. This is closer to what one would expect given the differences between these two architectures. Similarly, the difference between PC1M and EPVM1 increases to 40%. Both sets of hot results have been included since we believe that they illustrate how quickly the difference in performance between the architectures diminishes when a small amount of computation is performed on each object access.

Table 4 contains the cold and warm iteration times for traversal without updates over the small database when each iteration is executed as a separate transaction. In Table 4 the relative performance of the different architectures is identical to Table 1 during the cold iteration. Comparing the cold iteration times of Table 4 with Table 1 also shows that the overhead of transaction commit is relatively minor for all of the versions when no updates are done. The warm iteration results in Table 4 highlight the effects of inter-transaction caching. OS has the best performance in large part because it caches both data pages and locks between transactions. The OS client, therefore, only has to communicate with the server process once, to read the one page which was not accessed during the previous nine iterations. The versions using ESM, on the other hand, must communicate with the server to read uncached data pages and to reacquire locks on cached pages. PC1M caches the fewest pages between transactions because it only has a 1 megabyte client buffer pool. This causes it to have the worst performance. We also ran this experiment without inter-transaction caching for ESM. Inter-transaction caching improved performance by 40% for CESM and EPVM1 and by just 7% for PC1M during the warm iteration.

The cold and warm times for the four versions based on EPVM 2.0 are not shown for the multiple transactions experiment. The cold iteration times were all within 1% of those shown in Table 2.

Traversal without updates				
Version	**Cold**	**I/Os**	**Warm**	**I/Os**
CESM	10.669	325	1.962	133
EPVM1	10.712	327	2.075	135
PC1M	11.430	327	3.669	283
OS	12.734	217	0.404	1

Table 4. Multiple transactions w/o updates (times are in seconds).

The warm iteration times were, of course, slower than the warm times in Table 2 since locks on pages had to be reacquired. OC5M had the best performance in the warm iteration. It was 42% faster than PC1M and 50% faster than OC1M. Pointer swizzling made essentially no difference for either PC1M or EPVM1 in this experiment. In addition, the hot times were within 2% of the warm times for PC1M and OC1M. The hot time for OS was 0.377 seconds which is 7% faster than the warm time for OS (Table 4). The hot times for CESM, EPVM1, and OC5M were approximately 50% faster than their corresponding warm times.

We next consider the effect of adding updates to the traversal. Figure 7 presents the total execution time for a single transaction consisting of 1 cold, 9 warm, and 10 hot iterations when the update probability ranges between 0 and 1. The non-swizzling versions EPVM1-NO and PC1M-NO where each within 1% of EPVM1 and PC1M, respectively, and so are not shown. In addition, the performance of CESM was roughly 7% faster than EPVM1 throughout.

OS has the fastest time when no updates are done, however, the relative performance of OS degrades as updates are added due to the high cost of transaction commit. We believe that transaction commit is more expensive for OS because full page logging is used. The performance of OS levels off once the frequency of updates is high enough so that all pages are being updated. PC1M is always faster than OS when updates are performed. The performance of EPVM1 continually degrades as the update probability is increased because it generates a log record for every update. It is a little surprising that EPVM1 is better than PC1M and OS in many cases. This is due in large part to the fact that the log records generated by EPVM1 can be processed asynchronously by the server while the transaction is running. PC1M is faster than EPVM1 when the update probability is greater than about .3. The commit time for PC1M is constant once all of the objects visited during the transaction have been updated.

OC1M has the worst performance overall in Figure 7 since it must reread pages from the server while the transaction is running and also during the commit phase. The performance of OC5M shows that object caching can perform quite well when its client buffer pool is large enough to avoid having to reread data pages. The difference between PC1M and OC5M is because PC1M must reread pages during transaction commit in order to generate recovery information. If PC1M is given a bigger client buffer pool then its performance is nearly identical to the performance of OC5M.

Figure 8 presents the overall execution time for traversal with a varying write probability when each iteration of the benchmark constitutes a separate transaction. The curves for PC1M-NO, EPVM1-NO, and CESM are again omitted due to their similarity to the curves for PC1M, and EPVM1. OS has the best performance when the update probability is low. As the update probability is increased, however, the relative performance of OS degrades due to the high cost of transaction commit. OC1M is a little slower than PC1M in Figure 8 because OC1M must reread pages from the server while a transaction is executing. This overhead is greater than the cost of the extra copying done by PC1M. The curve for OC5M shows that if enough memory is available, then object caching performs the best in most cases. EPVM1 does quite well because it avoids the extra copying overhead of PC1M and the object caching versions and also does not have to reread data pages from the server in the context of any single transaction. Finally, we note that inter-transaction caching improved

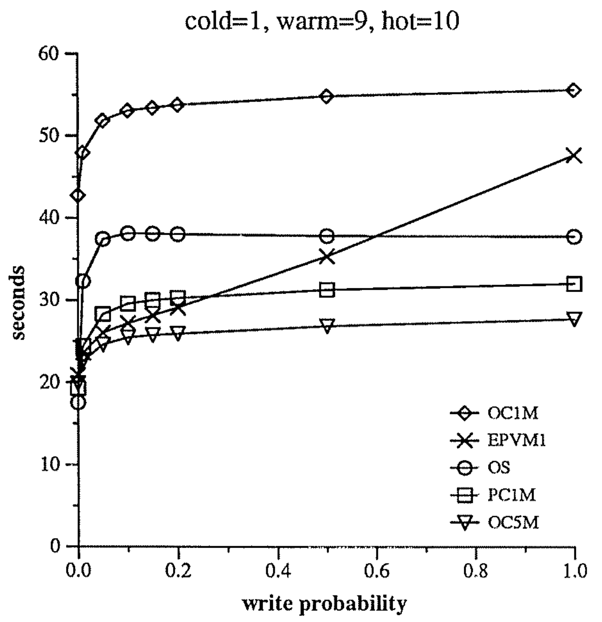

Figure 7. Benchmark run as a single transaction.

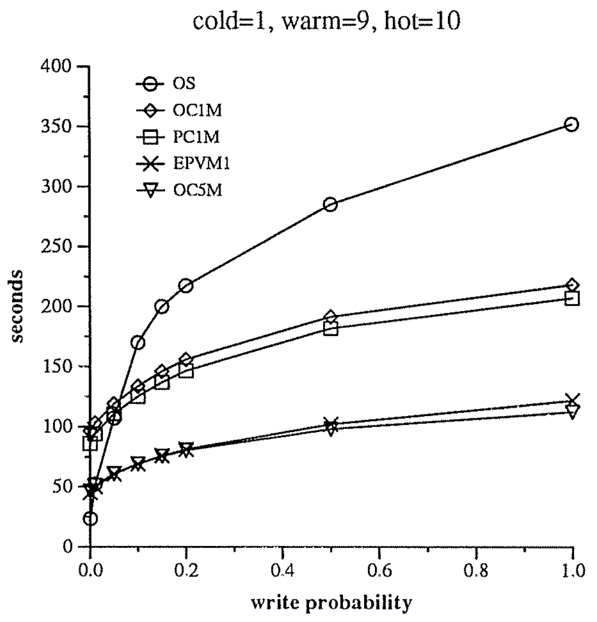

Figure 8. Benchmark run as multiple transactions.

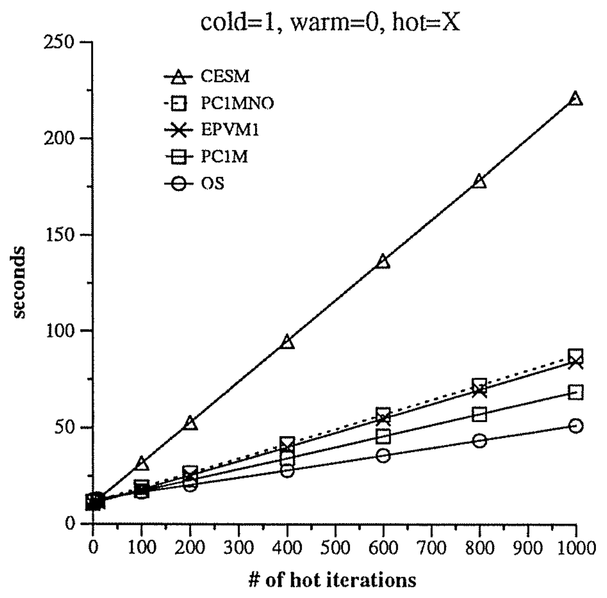

Figure 9. Single read-only transaction.

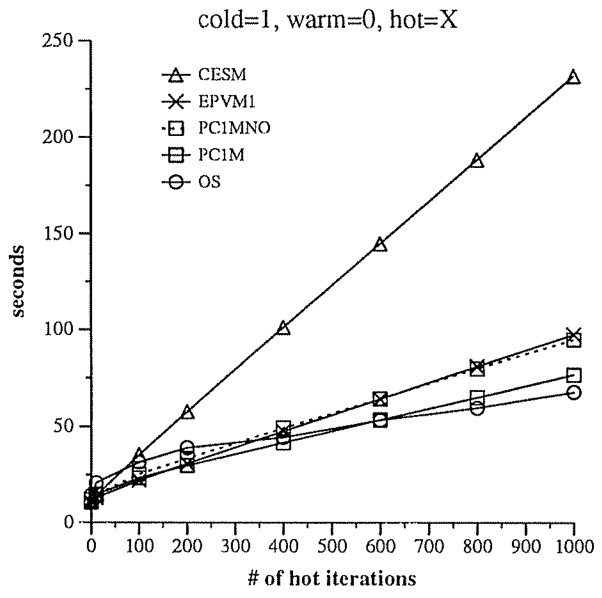

Figure 10. Single transaction (update prob. = .01).

performance for OC5M and EPVM1 from 43% (read-only) to 29% (write prob. = 1). The improvement was smaller when updates were done because of the fixed overhead for sending dirty data pages back to the server at the end of each transaction. PC1M and OC1M posted a 5% gain in performance when caching was added.

Figures 9 and 10 fix the number of cold and warm iterations at 1 and 0, respectively, and vary the number of hot iterations between 0 and 1000. In both figures each benchmark run was a single transaction. In Figure 9 the update probability was 0 and in Figure 10 it was .01. Both figures illustrate the large difference in CPU requirements between CESM and the other versions when a large number of hot traversals are performed. Although it is not easy to see in Figure 9, EPVM1 has the best performance when the number of hot iterations is between 1 and 60 and OS does the best when the number of hot iterations is greater than 60. PC1M is better than EPVM 1.0 after approximately 60 hot iterations have been performed as well.

When 1000 hot iterations are done, OS is 25% faster than PC1M, 39% faster than EPVM1 and posts a 76% improvement over

CESM. PC1M always does better than PC1M-NO and shows an improvement of 21% when 1000 iterations are done. The results for EPVM1-NO are not shown, however, EPVM1 was 7% faster than EPVM1-NO after 1000 iterations. The times for OC5M and OC1M were within 1% of PC1M in Figure 9, so their times have been omitted as well.

In Figure 10, when the number of hot traversals is between 1 and 160 EPVM1 has the best performance. PC1M is the best when the number of hot traversals is between 160 and 600. After 600 hot iterations OS is always the fastest. It is surprising that 600 hot traversals must be performed in order for OS to perform the best, but this is due to the relatively high cost of transaction commit for OS. Turning to swizzling, after 1000 iterations PC1M-NO was 24% slower than PC1M and EPVM1-NO (not shown) was 7% slower than EPVM1. OC5M and OC1M are also not shown in Figure 10. OC1M was within 1% of PC1M in all cases. The performance of OC5M was initially 15% faster than PC1M and 6% faster than PC1M after 1000 iterations.

Figure 11 demonstrates what happens when one varies the fraction of each part object that is updated while the update probability remains fixed. In this experiment part objects were defined to contain an array of 19 integers (76 bytes) instead of the usual non-pointer data specified by the OO1 benchmark. The x-axis shows the percentage of this array that was updated. Not surprisingly, OS has relatively flat performance once any updates are done. This is because it does full page logging. The versions based on EPVM 2.0 also show little change in performance once updates are added. This is because the number of log pages that were generated only varied from 51 to 160 as the update fraction was increased. ESM required roughly two seconds to process these extra log pages. The performance of EPVM1 degrades quickly as a larger portion is updated because it generates a log record for each update. The number of log pages generated by EPVM1 varied from 171 (update 1 integer) to 3,325 (update whole array).

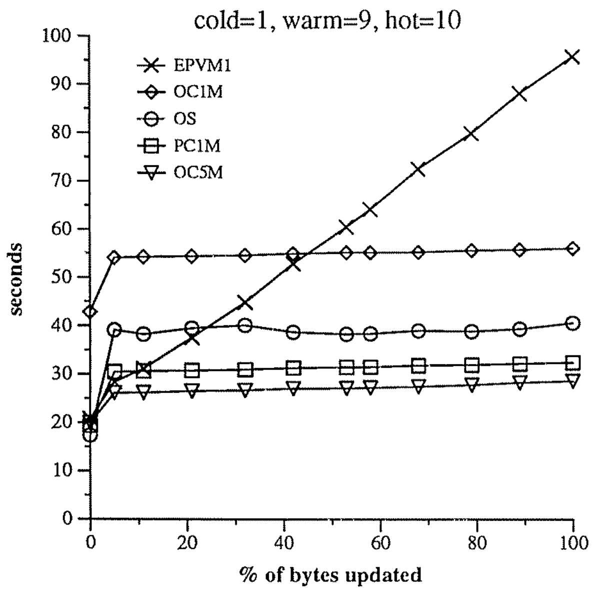

Figure 11. Single transaction (update prob. = .3).

5.2. Large Database Results

In the large database (125,000 parts) experiments the number of page faults that occurred was important for some versions. Page faults are listed in parentheses next to the number of normal I/O operations done by the client for the versions that experienced page faults. The number of page faults was obtained by using the Unix *getrusage* system call.

Tables 5 and 6 present the cold and warm times observed when the benchmark was executed as a single transaction, so the time for transaction begin and commit is not included. CESM has the best performance in the cold iteration of Table 5. PC1M does fewer I/O operations, but is slower than CESM due to copying costs. EPVM1 is slower than CESM primarily because it does a less effective job of buffer management. OS has the worst performance in the cold iteration. We believe this is due to the cost of mapping data in and out of the client's address space.

PC1M performs the best in the warm iteration, but comparing PC1M to the other architectures is not strictly fair in this case since it is allowed to use all of available memory, as shown by the number of page faults that it experiences. CESM and EPVM1 are close in terms of performance, but EPVM1 is a little slower due to the fact that it performs more I/O and must insert objects into the OID hash table. OS is surprisingly 12% slower than EPVM1 in the warm iteration. As with the cold iteration, this is likely due to data mapping costs[Orens92].

In Table 6 OC1M has the worst performance in the cold iteration because it performs more I/O operations. PC1M is a little slower than OC5M due to the overhead of copying full pages. Swizzling makes no difference for PC1M in the cold iteration. The relative times in the warm iteration are similar to the cold iteration. However, the performance of PC1M-NO is a little better than PC1M since swizzling dirties pages in virtual memory causing them to be written to disk more often. This fact doesn't show up in the number of page faults shown in Table 6 since these numbers only give the number of pages read from the swap area by the process. The times for EPVM1-NO were essentially identical to EPVM1 in both the cold and warm iterations and so are not shown.

Traversal without updates				
Version	Cold	I/Os	Warm	I/Os
CESM	38.643	1093	30.973	909
EPVM1	39.582	1149	32.716	928
PC1M	38.894	1014	24.180	33 (699)
OS	48.614	839	36.567	615

Table 5. Single transaction without updates (times are in seconds).

Traversal without updates				
Version	Cold	I/Os	Warm	I/Os
OC5M	38.098	1082	23.401	675
OC1M	43.107	1516	27.035	988
PC1M	38.894	1014	24.180	33 (699)
PC1M-NO	38.838	1014	22.540	33 (705)

Table 6. Single transaction without updates (times are in seconds).

When each iteration was executed as a single transaction, the cold times were all within 2% of the times shown for the versions in Tables 5 and 6. In the warm iteration the times for the versions included in Table 5 were also all within 2%, except for PC1M whose performance was slower by 23%. The decrease in performance for PC1M was due to the fact that it was not able to cache as much data in virtual memory and it also performed a lot of unnecessary copying. OC1M was 12% slower than PC1M during the warm iteration and OC5M was just 2% faster than PC1M. PC1M-NO and EPVM1-NO were each within 1% of PC1M and EPVM1 respectively in the multiple transactions experiment. Repeating the experiment without inter-transaction caching showed that caching had much less impact on performance when using the large database. Caching improved the performance of EPVM1 by 4% and PC1M by just 2% during the warm iteration.

Figure 12 presents the total execution time for traversal when 1 cold, 9 warm, and 0 hot iterations are run as a single transaction. OS has the worst performance in most cases. It may be surprising, given the results presented in Table 5, that OS is better than PC1M in the read only case. PC1M is slower in this case because when it scans the page hash table during transaction commit to determine which objects have been updated, it causes a significant amount of virtual memory swapping activity. This poor performance during the commit phase makes PC1M slower than the other versions as well.

It should be noted that in the large database case it is not strictly fair to compare OS, EPVM1, and CESM to the page caching and object caching versions since the caching versions are allowed to use more memory. The comparison between EPVM1, CESM, and OS is fair, however, since these versions were given equal amounts of memory. The times for EPVM1-NO (not shown) were all within 1% of EPVM1. PC1M-NO, which is also not shown in Figure 12, was 4% faster than PC1M in the read only case because swizzling dirtied pages in virtual memory for PC1M which caused an increase in paging activity. The difference between PC1M and PC1M-NO gradually diminished as more updates were performed and PC1M-NO was well within 1% of PC1M when the update probability was 1.

Figure 13 presents the total execution time for traversal when 1 cold, 9 warm, and 0 hot iterations are executed as separate transactions. OS has the worst performance in all cases in Figure 13. CESM has the best performance, but is only slightly faster than EPVM1. Turning to object and page caching, the performance of page caching is intermediate between OC1M and OC5M. This again illustrates the tradeoff made by object caching which must reread pages from the server and page caching which caches more objects and copies more data into virtual memory. EPVM1-NO (not shown) and PC1M-NO (not shown) were always within 1% of EPVM1 and PC1M respectively in Figure 13.

6. Conclusions

This paper has presented a detailed discussion of the implementation of pointer swizzling and object caching in EPVM 2.0. The paper then analyzed the relative performance of several versions of EPVM 2.0 using the OO1 benchmark. EPVM 2.0 was also compared to some alternative methods of supporting persistent data access, including the memory-mapped approach of ObjectStore V1.2.

The OO1 cold iteration times for ObjectStore were slower than the cold times for the architectures based on ESM when using both a small and a large database. ObjectStore had the fastest warm iteration time when using the small database, but when the large database was used ObjectStore had the worst warm performance. These results suggest that either the I/O performance of ObjectStore is worse than that of ESM or that mapping data into a process's address space is a relatively expensive operation. The hot iteration results (done using the small database) showed that the memory-mapped scheme used by ObjectStore is five times

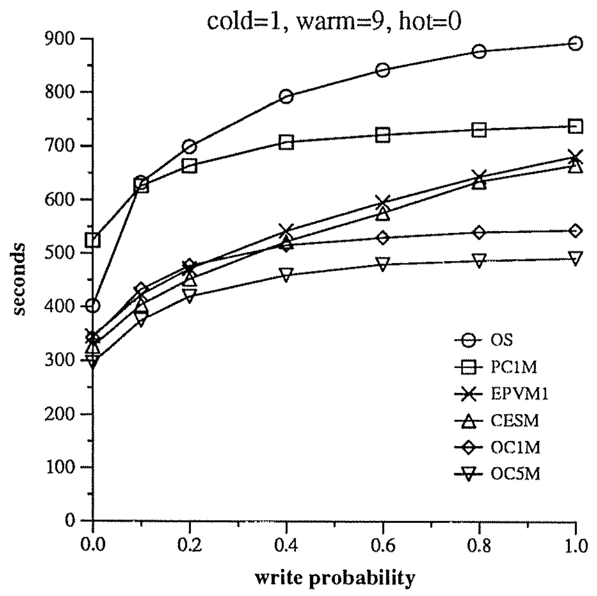

Figure 12. Benchmark run as a single transaction.

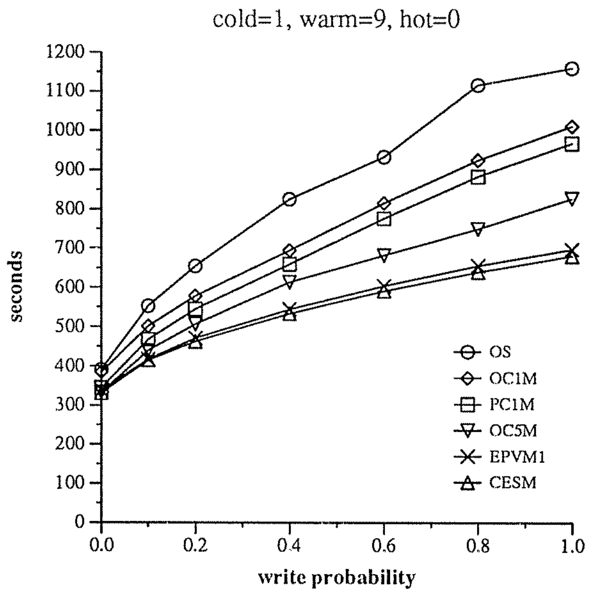

Figure 13. Benchmark run as multiple transactions.

faster than the software approach of EPVM 2.0 when operating on in-memory data. However, it was observed that the difference in performance was only 33% when a small amount of additional computation was added.

The paper also compared the total elapsed time of the different architectures using several transaction workloads. When a small database was used (Figures 7 and 8), ObjectStore had the best performance in the read-only case. It was shown, however, that PC1M generally performed better than ObjectStore when updates were performed. The main reason for this appears to be that ObjectStore does full page logging in order to support crash recovery. EPVM1 and CESM performed better than ObjectStore and PC1M when the frequency of updates was low and when multiple transactions were used. When a large database was used, the memory-mapped approach of ObjectStore always had slower performance than EPVM1 and CESM.

Among the versions based on EPVM 2.0, PC1M had better overall performance than OC1M when the small database was used. PC1M does well because the cost of copying full pages is relatively small compared to the cost of copying individual objects in this case. PC1M also avoided the need to reread pages from the server during normal transaction execution as was done by OC1M. When the large database was used, however, OC1M generally performed better than PC1M. PC1M performed a lot of unnecessary copying work and experienced paging of virtual memory which lowered its performance in this case.

The swizzling scheme used by EPVM 2.0 never noticeably hurt performance when the small database was used and improved performance by as much as 45% in some cases (see Table 3 column 4). When the large database was used, swizzling did not improve performance and resulted in a 4% decrease in a few cases due to the fact that it caused an increase in the amount of virtual memory paging activity. Lastly, we note that the swizzling scheme used by EPVM1 improved performance by 16% in some cases when using a small database and had no effect when using the large database.

We feel that an important conclusion that can be drawn from the results presented in the paper is that it is important to look at overall performance when comparing the different architectures. For example, simply comparing the speed with which the architectures manipulate in-memory data or comparing them without considering recovery issues does not capture the true differences in performance between the systems. In the future, we would like to explore variations of the object caching and page caching schemes studied here in the context of EPVM 2.0 to see if an approach combining their relative strengths can be found. We are also interested in finding more efficient ways of generating recovery information both in the context of EPVM and the memory-mapped approach. If a more efficient method of generating recovery information for the memory-mapped approach can be found, then we feel that its performance could be improved substantially.

References

[Atkin83] M. Atkinson, K. Chisholm, and P. Cockshott, "Algorithms for a Persistent Heap," Software Practice and Experience, Vol. 13, No. 3, pp. 259-272, March 1983

[Catte91] R. Cattell, "An Engineering Database Benchmark," in *The Benchmark Handbook For Database and Transaction Processing Systems*, Jim Gray ed., Morgan-Kaufman, 1991.

[Carey89a] M. Carey et al., "The EXODUS Extensible DBMS Project: An Overview," in *Readings in Object-Oriented Databases*, S. Zdonik and D. Maier, eds., Morgan-Kaufman, 1989.

[Carey89b] 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.

[Cock84] P. Cockshott et al., "Persistent Object Management System," Software Practice and Experience, Vol. 14, pp. 49-71, 1984

[Exodu92] Using the EXODUS Storage Manager V2.0.2, technical documentation, Department of Computer Sciences, University of Wisconsin-Madison, January 1992.

[Frank92] M. Franklin et al., "Crash Recovery in Client-Server EXODUS", Proc. ACM SIGMOD Int'l Conf. on Management of Data, San Diego, California, 1992.

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

[Moss90] J. Eliot B. Moss, Working with Persistent Objects: To Swizzle or Not to Swizzle, COINS Object-Oriented Systems Laboratory Technical Report 90-38, University of Massachusetts at Amherst, May 1990.

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

[Orens92] J. Orenstein, *personal communication*, May 1992.

[Rich89] J. Richardson, M. Carey, and D. Schuh, The Design of the E Programming Language, Technical Report No. 824, Computer Sciences Dept., University of Wisconsin, Feb. 1989.

[Rich90] J. Richardson, "Compiled Item Faulting," *Proc. of the 4th Int'l. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, September 1990.

[Schuh90] D. Schuh, M. Carey, and D. Dewitt, Persistence in E Revisited---Implementation Experiences, in *Implementing Persistent Object Bases Principles and Practice*, The Fourth International Workshop on Persistent Object Systems.

[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.

Acknowledgements

We wish to thank Jack Orenstein, Dan Weinreb, and Benson Margulies of Object Design, Inc. for their many helpful comments and feedback concerning the results presented in the paper. Special thanks should be given to Dan Schuh who implemented EPVM 1.0 and the E compiler. Mike Zwilling and Mike Franklin implemented the ARIES based recovery algorithm used in the EXODUS storage manager. Nancy Hall implemented inter-transaction caching.