Persistence in E Revisited — Implementation Experiences

Dan Schuh, Michael Carey, and David Dewitt

Computer Sciences Department
University of Wisconsin
Madison, WI 57306
schuh@cs.wisc.edu

ABSTRACT

This paper discusses the design and implementation of the E Persistent Virtual Machine (EPVM), an interpreter that provides support for persistent data access in the current version of the E programming language. Included are descriptions of both the EPVM interface and the major implementation tactics employed within EPVM. A novel pointer swizzling scheme that has been investigated in the context of E and EPVM is also described. Finally, a performance analysis of the key EPVM primitives is presented.

1. INTRODUCTION

This paper discusses the current implementation of access to persistent data in the E programming language. E is a persistent, object-oriented programming language that was developed as an extension of C++ [Stro86] for use in the EXODUS project at the University of Wisconsin [Care89a]. The E language was originally designed for use in constructing database management system software; an overview of the design and motivation for E can be found in [Rich89a]. Basically, three main innovative features distinguish E from C++: persistent data, generator classes, and iterator functions. Support for persistent data allows program-level data objects to be transparently used across multiple executions of a program without requiring explicit input and output operations. Persistent data in E programs is stored using the EXODUS storage manager [Carey89b]. The current version of E (E 2.0) is based on AT&T C++, version 2.0. The E language processor was built by extending AT&T's C++ 2.0 preprocessor, so it produces C code which is then compiled by a native C compiler to produce object modules and executable programs.

The design and initial implementation of persistence in E, which was the basis for persistent data in E 1.2, is discussed in [Rich89b]. A substantially improved approach to persistence for E 1.2, based on compile-time analysis of E programs, is described in [Rich90]. The topic of this paper is a new approach that is currently employed in E 2.0. The current approach is based on an underlying interpreter for persistent object accesses. This interpreter, called the E Persistent Virtual Machine (EPVM), is invoked through a small set of function calls in the C code that the E 2.0 compiler generates. The interpreter is primarily used to access persistent data; most non-persistent data access and computation in E is compiled into C code and later into the native instruction set of the host computer.

The current approach to persistence in E has several advantages relative to our earlier approaches. Compilation of persistent code is quite simple, and the generated code is compact. The interpretive approach also allows a great deal of flexibility in the underlying implementation strategy. For example, it allows statistics gathering, stronger error checking, and debugging support to be implemented without impacting the normal execution efficiency of E code by simply loading a different version of EPVM. Moreover, the E compiler is no longer tied to the semantics of the underlying storage manager, and could thus be more easily adapted to other storage managers or to different uses.

The advantages discussed above are useful, but they are relatively peripheral in comparison to the main advantage of the interpretive approach — which is the ability of the EPVM implementation to maintain global context over the lifetime of a given program execution. The EPVM interpreter is able to maintain information on all currently accessible objects in order to reduce the number of calls to the underlying storage manager. As such, the interpreter can adapt its behavior based on dynamic (runtime) access patterns. While in principle a pure compiled approach could be based on a global analysis of access patterns, practical considerations usually limit the analysis

---

This research was partially supported by DARPA under contract N00014-88-K-0303, NSF under grant IRI-8657323, and donations from Texas Instruments and Digital Equipment Corporation.

that can be performed to the scope of a single function or procedure. Also, an approach based on compilation must base its code generation decisions on a static analysis of program behavior. The main disadvantage of taking an interpretive approach is runtime inefficiency. However, we believe that the advantages and simplicity of this approach may outweigh its execution time penalty for a broad class of programs. In fact, for programs where a complex global analysis would be required to make the compilation approach perform well, the interpretive approach may well win in terms of performance.

The remainder of this paper is organized as follows. Section 2 discusses the EXODUS storage manager, which is the underlying mechanism used to store persistent objects in E. We give a general overview of its functionality and discuss in detail the most relevant portions of its interface. Section 3 presents a brief overview of the design of persistence in E. Included in the discussion are the E type system, its interaction with persistence, and the various means by which persistent objects are created, destroyed, and addressed. We then turn our attention to EPVM, the runtime interpreter for persistent object accesses. Section 4 describes the EPVM interface and gives examples of its use in C code generated by the E compiler. Section 5 discusses the implementation of the interpreter, showing how it interacts with the EXODUS storage manager; this section also describes extensions to the compiler and EPVM that facilitate a limited version of pointer swizzling in order to allow certain accesses to bypass the EPVM interface. Section 6 contains a detailed cost analysis of the EPVM approach. Finally, in Section 7 we compare our approach with some of the alternatives, reflect on why the approach described here was chosen, and discuss ideas for future work.

2. THE EXODUS STORAGE MANAGER

The EXODUS storage manager (ESM) is the underlying repository used to store and manipulate persistent E objects. ESM manages the allocation of storage for persistent objects, both on disk and in main memory, and it controls the transfer of objects between disk and memory. In this section we provide a brief overview of the ESM programmer's interface. We begin with a brief overview of addressing issues as they apply to ESM storage objects, and follow with detailed descriptions of the key routines used to pin, unpin, and modify objects in the buffer pool. Further details can be found in [Care89b].

2.1. Addressing

There are two forms of persistent object addresses that concern us here. The first is the (permanent) address of an object stored on disk, and the second is the (temporary) address of an object in main memory. Every ESM storage object is identified and accessed though the use of a unique object identifier (OID). An OID is a 12-byte quantity that consists of a 2-byte volume identifier, a 4-byte page number, a 2-byte slot within the page, and a 4-byte unique field. The volume number, page ID, and slot number together specify the physical location of the object in secondary storage, while the unique field is used to prevent reuse of identical OIDs (i.e., to make OIDs safe as unique IDs). The data structure representing an OID is depicted schematically in Figure 1. Finally, ESM supports addressing of subportions of objects; an arbitrary byte within a given object can be addressed by combining its 12-byte OID with a 4-byte offset from the start of the object.

To address storage objects in the main memory buffer pool, client programs depend on user descriptors. A user descriptor is a structure that contains a pointer to a portion of an EXODUS storage object that currently resides in the ESM buffer pool. The user descriptor structure is shown in Figure 2. ESM allocates user descriptors from a pool that it maintains, and calls to access ESM objects return pointers into this pool. Thus, ESM users always

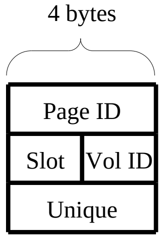

Figure 1: OID Format.

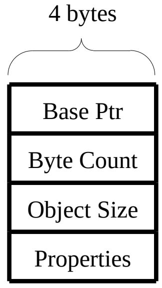

Figure 2: User Descriptor Format.

access objects in the buffer pool through two levels of indirection, first dereferencing a user descriptor pointer to the user descriptor itself, and then dereferencing the user descriptor's data pointer to obtain the actual data in the buffer pool. This form of address is sometimes referred to as a "handle" (though in this case the user descriptor pointer also points to certain other data that may be inspected by the programmer). Since objects are intended to be accessed indirectly through user descriptors, ESM is free to relocate an object within the buffer pool (if needed) by adjusting any affected pointers in the user descriptor pool. When no user descriptors point to a particular object in the ESM buffer pool any more, the object is said to be unpinned and may migrate out to disk.

2.2. EXODUS Storage Manager Interface

The programmer's interface to ESM consists of a set of function calls to support operations on individual objects and collections of objects (files). ESM provides routines to create and destroy objects, to read and overwrite portions of objects, to extend and shrink the size of existing objects, and to sequence through files of objects. Also, ESM provides facilities to enable sophisticated client programs to influence policy decisions such as object clustering and buffer management.

For the purposes of implementing access to persistent objects in E, the key ESM routines are those that bring objects into main memory, update them there, and release them for migration back to disk. These routines are sm_ReadObject, sm_WriteObject, and sm_ReleaseObject, respectively. We discuss each of these routines in more detail below, as they are central to the operation of E as a persistent programming language.

The first routine, sm_ReadObject, provides the interface for establishing a memory reference to an object given its OID. Given an OID and a byte range specified by an offset and a length, sm_ReadObject brings the desired portion of the object into ESM's buffer pool (if it is not already buffer-resident) and returns a pointer to a newly allocated user descriptor to the calling program. The user descriptor itself contains a pointer to the requested portion of the object in the buffer pool. The requested data is considered ''pinned'' in the buffer pool until it released via an sm_Release call, and the user descriptor pointer remains valid for this duration. Again, it is important to remember that what sm_ReadObject returns is a user descriptor pointer, and that ESM reserves the right to relocate the actual data in the buffer pool as long as it updates the user descriptor's data pointer. As a result, data in ESM storage objects must always be accessed indirectly via the user descriptor.

The next routine, sm_WriteObject, is used whenever a storage object is to be updated. Before an object can be updated, it must first be pinned in the buffer pool by calling sm_ReadObject. For the purpose of updates, objects are addressed by user descriptors, not OIDs. The update itself is requested by passing sm_WriteObject a user descriptor, a specification of the range of pinned data that is to be updated, and a pointer to the data to be copied into the object. Objects may not be directly overwritten in memory because ESM uses a recovery scheme that requires it to be given an opportunity to record both the old and new values for updates to objects. Finally, updating an object does not result in the object being unpinned in general, but an option is provided to allow simultaneous unpinning if it is desired.

The third routine, which was already mentioned, is sm_ReleaseObject. This routine accepts a user descriptor pointer and unpins the associated object region, freeing the user descriptor for subsequent reuse. This routine is needed since sm_ReadObject pins a region of the target object in the ESM buffer pool, and both the buffer pool and the user descriptor pool are finite resources.

Note that no explicit disk I/O is specified by clients in the ESM interface. While the sm_ReadObject call requires that the specified object region be in the buffer pool when the call returns, this does not necessarily imply a disk operation. Similarly, sm_WriteObject and sm_ReleaseObject do not require that the referenced data in the buffer pool be written to disk. Although disk I/O must take place sometime, the interface allows ESM considerable flexibility in scheduling the actual disk operations.

An important point of this section is the relationship between addresses specified using OIDs and those specified via user descriptors. An OID is always a valid specifier for an object; it is valid for the object's entire lifetime and across all programs that have access to the storage volume that contains the object. However, there is no permanent relationship between an OID and an object in memory. To access an object in main memory, a user descriptor must first be established. This descriptor is then a valid means of addressing the specified object, but only until the user descriptor is explicitly freed.

3. TYPES AND PERSISTENCE IN E

This section reviews the semantics and usage of persistence in the E language. Persistence of objects in E is based on the method of allocation used when creating an object. Details of the type system limit the set of program data references that may be persistent. In what follows, we describe the E type system as it relates to persistence. In particular, we attempt to clarify four key topics — the database type system, the persistent storage class, dynamic allocation of persistent objects, and the role of ESM files.

Persistence of an object in E is dependent on a combination of the static type of a declared object and on the storage class or method of allocation of the object [Rich89a, Rich89b]. In particular, E uses a dual type system with a concept of database (db) types to statically distinguish the types of objects that may possibly persist from those types that describe objects that are sure to be transient. In E, all primitive types and type constructors of the native type system of C++ have an analogous db form. The declaration of a db type is determined by prepending ‘‘db’’ to the C++ type keyword. For example, the primitive C++ types int, float, and double have the db counterparts dbint, dbfloat, and dbdouble, respectively. Similarly, classes and structs of db types are declared using the keywords dbclass and dbstruct. Pointers to db types are also db types, as are arrays of db types. Structures and classes of db type may only contain data members that are also db types. Pointers are the only non-composite type in which a db type differs in its physical representation from a native type. The db type system was made distinct from the native C++ type system so that variables that are not declared as db types can be treated uniformly as transient, main memory objects, and can thus be manipulated with the same efficiency as is possible in C++.

The fundamental distinction between db types and native C++ types in E lies in the treatment of pointers to these types. For any non-db type, a pointer to an instance of that type is a simple (e.g., 4-byte) memory pointer and is treated as such by the compiler. In contrast, a pointer to a db type object is always a 16-byte db pointer (which is an OID, offset pair). Any pointer that is declared as pointing to a db type object may potentially point to a persistently allocated object, although it may also be used to point to a non-persistent instance of a db type. The treatment of values other than pointers in E’s db type system is just as efficient as the treatment of native C++ values. However, the treatment of pointer values incurs the additional overhead of moving additional bytes around. The treatment of references through db pointers is the most important distinction between db types and non-db types.

For an object to be persistent in E, it must be declared as a db type. This is a necessary but not sufficient condition for persistence. To persist, the object must also be allocated persistently. For named objects, i.e., those directly declared in E programs, this is accomplished using the persistent storage class. This storage class is analogous to other C++ storage classes such as extern, static, and auto. If an object is declared to be persistent, the state of the declared object is maintained across different program runs using the same compiled module and ESM volume. Also, the storage class distinction between persistent and non-persistent db type objects allows transient objects to be created and used efficiently within the db type system; even within the db type system, temporary variables and other objects can be created which do not require the overhead of space allocation and deallocation through ESM.

The other method of creating persistent objects in E is dynamic allocation. Db type objects can be dynamically allocated as either persistent or transient objects. If a db type object is allocated using the standard C++ syntax for dynamic allocation, the allocated object will be a transient memory object. The lifetime of the object will then extend until the object is explicitly deleted or until its creating program terminates, whichever is sooner. Dynamic allocation of persistent objects is also possible, and is supported by allowing db type objects to be allocated within collections. Collections are the principle persistent storage management abstraction provided by E, and they are implemented using ESM files. At the simplest, a collection can be viewed as a class that implements a persistent heap. If a db type object is allocated within a persistent collection, its lifetime extends until it is explicitly or implicitly destroyed. An object may be destroyed explicitly by using the delete operator, or it may be destroyed implicitly by destroying the collection that contains it.

Consider the three db type objects allocated in the E code fragment shown in Figure 3. In the absence of explicit destruction by the delete operator, each of these objects will have a different lifetime. The lifetime of the object that `obj1` points to extends until the program that created the object terminates. This object only exists in the transient heap of the program. In contrast, `obj2` points to an object that will exist for the lifetime of the collection that it was created in; this collection will persist for the lifetime of the ESM storage volume on which it resides. This object will thus be maintained on disk across program executions. Finally, the object pointed to by `obj3` is also a ‘‘persistent’’ object of sorts, but its lifetime is limited by that of the transient collection that it was allocated in; when the non-persistent program variable `temp_collection` goes out of scope, all of the objects that it contains will be destroyed as well.

In summary, the key points of this section are (i) the notion of db types, (ii) the distinction between objects and references in db types and native C++ types, and (iii) the conditions under which an object may be persistent. There are no persistent values of native C++ types, and references to such types are simple, main memory pointers. Objects of db types may be, but are not necessarily, persistent; a persistent object must be of some db type. References to db type objects are always treated as if the referenced object may be persistent since all pointers to db types are capable of storing a persistent address. The persistence of named db type objects is determined by their declared

```
dbclass vcollection : collection[dbvoid];
persistent vcollection pers_collection;

{ dbchar *obj1, *obj2, *obj3;
      obj1 = new dbchar[10];
      obj2 = in (pers_collection) new dbchar[10];
      vcollection temp_collection;
      obj3 = in (temp_collection) new dbchar[10];
};
```

Figure 3: Object Lifetime Example.

storage classes, while the persistence of dynamically allocated db type objects depends on their method of allocation. Finally, db type values are handled by E just like native C++ values when they are not accessed via pointers, but all such values are treated as potentially being persistent when accessed via pointers.

4. E PERSISTENT VIRTUAL MACHINE INTERFACE

The E Persistent Virtual Machine (EPVM) provides a set of C function calls. The EPVM interface corresponds to a limited portion of the instruction set of a conventional computer. In this section we describe three main types of functions that EPVM provides. The first type is analogous to the load instructions in an instruction set architecture. These functions read a value from persistent storage given a location which is specified as a db pointer and an offset. The second type is analogous to the store instructions of an instruction set architecture. These functions accept a db pointer, an offset, and a value as parameters and store the value at the indicated persistent address. The third type of functions manipulate persistent pointer values, performing operations like pointer arithmetic and pointer comparisons that would be handled via integer arithmetic on address values in a conventional instruction set architecture. The functions listed in this section comprise the major part of the EPVM interface. There are a number of other auxiliary functions, not listed here, which were added to allow more efficient implementation of certain E language constructs. However, the functions listed here are sufficient to implement all accesses to persistent objects in E.

4.1. A Simple Example

We begin by considering a simple code E fragment that illustrates the three main uses of db pointers that the E compiler and EPVM need to handle, which are (i) dereferencing a pointer to obtain a value, (ii) assigning a value to a location specified via a pointer, and (iii) testing pointer values. The E source code for our example is given in Figure 4. The function shown there sets the elements in a linked list of integers to a sequence of integer values beginning at 1. It returns the length of the list, which is assumed to have been created elsewhere. The linked list element type is a db type, so persistent linked lists may be created and operated upon using this code.

```
dbstruct ilist {  /* simple integer linked list */
      dbint i;
      ilist * next;
};

dbint set_list(ilist * lpt) {
      dbint index = 0;
      while (lpt) {
            index++;
            lpt->i = index;
            lpt = lpt->next;
      }
      return index;
}
```

Figure 4: Pointer Uses in E.

The utility of EPVM can be illustrated by examining the C code that the E compiler would produce as output when presented with the E code of Figure 4. Figure 5 shows the code produced by the E compiler.[1] Since the E variable `lpt` is a pointer to a db type, all uses of it are interpreted through the EPVM interface. In this example, the interpretation is done through calling the functions dbNotNull, dbLongAssign, and dbPtrEval. There are three lines in the E source code that must be translated. First, the condition in the while statement, which tests `lpt` for being null, must be interpreted since db pointers are not the same as simple C pointers. Next, the assignment of the `index` value to the linked list element field is implemented through a call to dbLongAssign. Finally, the pointer `lpt` is reset to point to the next item in the list via a call to dbPtrEval, which extracts the value from the `next` field of the list element and uses it to replace the current value of `lpt`. Note that the variable `index` is of a db type, yet it is treated as a normal C variable in the compiled code since it is not accessed through a pointer.

4.2. Load/Store Function Addressing Format

As described in Section 3, persistent pointers are 16-byte quantities that consists of a 12-byte OID and a 4-byte offset within the object. Because of the size of these addresses, they are passed into the EPVM routines by reference; that is, we always pass in the memory address of the C language variable or expression that contains the persistent pointer. It is also common for a pointer dereference to specify an address at an offset from the pointer variable that the dereference is based on. This is illustrated by the following E code fragment:

```
dbint i;
dbint * j;
 ...
i = j[5];
```

Here, it is necessary to add an offset of 20 to the pointer contained in the variable `j` to compute the address of the value to be assigned to `i`. Rather than creating a new db pointer with the adjusted offset to pass to EPVM, all EPVM functions accept persistent addresses as a (DBREF *, offset) pair. Although the DBREF value contains an arbitrary offset already, the additional offset argument lets the compiler avoid creating copies of pointer values in many cases.

4.3. Load Functions

There are seven EPVM functions that ‘‘load’’ values from persistent memory. Each accepts a persistent address (specified by a db pointer) and an integer offset and returns a value derived from the data stored at the specified address. These functions are specified as C function prototypes in Figure 6. Each of these routines returns a value

```
struct ilist {    /* sizeof ilist == 20 */
      int i ;
      struct DBREF next ;
};

int set_list( lpt )
struct DBREF lpt ;
{
      int index ;
      index = 0 ;
      while ((dbNotNull ( & lpt , 0 ) ))
      {
            index ++ ;
            dbLongAssign ( & lpt , 0 , index ) ;
            dbPtrEval ( & lpt , & lpt , 4) ;
      }
      return index ;
}
```

Figure 5: Pointer Uses in C + EPVM.

[1] Variable and data member names have been cleaned up somewhat for human consumption.

```
char        dbCharEval(DBREF *, int);
short       dbShortEval(DBREF *, int);
long        dbLongEval(DBREF *, int);
float       dbFloatEval(DBREF *, int);
double      dbDoubleEval(DBREF *, int);
DBREF *     dbPtrEval(DBREF *, DBREF *, int);
void *      dbArbEval(DBREF *, int, int);
```

Figure 6: EPVM Load Functions.

of the indicated type. Note that the types of the values returned are a subset of the primitive data types of E. This subset is sufficient to cover all of the primitive types with additional casting. For example, unsigned integers would be evaluated with the call:

```
(unsigned int)dbLongEval(...)
```

For the first five functions, the value returned is the value that currently resides at the persistent address indicated by the (DBREF *, int) pair passed in as parameters. The interface for returning DBREF values is somewhat different, however. Since a DBREF value is a large, 16-byte C structure, we chose to return the value for dbPtrEval through a pointer rather than as the function return value. Thus, the first parameter of dbPtrEval is the memory address where the db pointer value will be placed, and this parameter is also returned as the value of the function. Finally, the function dbArbEval is used for cases not covered by the set of primitive values returned by the other six functions (e.g., structure values).

4.4. Store Functions

There are six functions that accept a persistent address and a value argument representing data to be stored in persistent storage. These functions store the value at the indicated address. Figure 7 specifies the interface to these EPVM functions. The first five of these functions each take a persistent address (specified by a db pointer and an offset) and a primitive value as arguments; they store the value into persistent storage at the specified address. They also return the value for convenience, allowing C-style chained assignment statements to be handled via nested function calls. For example, the E statement

```
*a = *b = 3;
```

would compile into the following C code fragment:

```
dbIntAssign(&a, 0, dbIntAssign(&b, 0, 3));
```

The dbPtrAssign function differs from the first five only in that its pointer arguments, which are db pointers, are passed and returned by address rather than by value.

4.5. Pointer Manipulation Functions

Although many of the operations on pointer values are simple enough to be compiled directly into C code, for modularity and abstraction we chose to implement the manipulation of these values as primitive EPVM operations as well. Their function prototypes are shown in Figure 8. The function dbCmpPtr returns a truth value (0 or 1) based on the comparison of two db pointers. The sense of the comparison is specified by the argument op, which specifies one of the 6 comparison operators of C (<, <=, >, >=, ==, !=). For example, consider the E code fragment at the top of the next page:

```
char        dbCharAssign(DBREF *, int, char);
short       dbShortAssign(DBREF *, int, short);
long        dbLongAssign(DBREF *, int, long);
float       dbFloatAssign(DBREF *, int, float);
double      dbDoubleAssign(DBREF *, int, double);
DBREF *     dbPtrAssign(DBREF *, int, DBREF *);
```

Figure 7: EPVM Store Functions.

```
dbchar *p1, *p2;
 ...
if (p1 < &p2[12]) { .... }
```

This would generate the interpretive EPVM call:

```
DBREF p1, p2;
 ...
if ( dbCmpPtr(&p1, 0, &p2, 12, CMP_LT) { .... }
```

Here, CMP_LT is a literal constant defined by the compiler. The next pointer manipulation function, dbNotNull, returns 1 if the address specified by its arguments is not null and 0 otherwise; i.e., it returns 0 if the persistent pointer indicates transient memory address 0, returning 1 otherwise.[2] The function dbPtrVal is used to copy a memory-resident db pointer value to another memory location with an optional adjustment of the offset portion of the address. The function dbMemPtr is use to create a db pointer from a main memory pointer. Finally, dbPtrSub performs C pointer subtraction, scaled by the appropriate data type size.

5. IMPLEMENTATION OF THE EPVM INTERPRETER

In Section 3, on persistence in E, we noted that all access to persistent objects involves access through db pointers. In the E 2.0 version of the E language, dereferencing of db pointers is totally encapsulated by the EPVM interpreter functions. Thus, the key to a usable implementation of E is an efficient implementation of the EPVM functions on top of the EXODUS Storage Manager (ESM).

A naive EPVM implementation would pin and release the byte range necessary to satisfy each persistent data reference in each of the Eval and Assign functions described in Section 4. In fact, the initial implementation of EPVM took this approach. However, the minimal costs for the ESM pin and unpin instructions are hundreds of machine instructions, so another approach was needed. The current approach is based on maintaining a cache of the storage object regions that are known to be buffered in ESM's buffer pool. In particular, EPVM maintains a table of active user descriptors through which it can access each buffered object region, and it performs a simple hash lookup into this table in order to map each persistent address into a corresponding user descriptor. User descriptors are established on demand, when a particular object region is first accessed; once a user descriptor has been established, it is used for all subsequent accesses to the region of the target object that it is valid for.

In the remainder of this section of the paper, we describe the methods used to establish new user descriptors in this table and to map db pointers into user descriptors. We also describe an enhancement that allows the OID to user descriptor mapping operation to be bypassed in certain cases.

5.1. What to Pin

Before we can access any data in an ESM object, we must decide what to pin. The sm_ReadObject interface permits arbitrary byte ranges within objects to be pinned in the buffer pool. In making this decision for EPVM, we wanted to choose the region to pin in such a way as to maximize the usability of the user descriptor established by the pin call. At the same time, though, we wanted to avoid making ESM perform excessive disk accesses for data that may never be used. For small objects, the decision was easy — since the underlying ESM I/O calls are page-based, small objects are alway buffered in the ESM buffer pool in their entirety. Thus, EPVM always pins the entire small object, which allows it to use a single user descriptor to perform any allowable access to the object.

ESM supports storage objects that may be almost arbitrarily large, so it is unreasonable to pin entire objects in all cases. Large objects are therefore handled using a paging scheme that works as follows. The EPVM interpreter

```
int          dbCmpPtr(DBREF *, int, DBREF *, int, op)
int          dbNotNull(DBREF *, int)
DBREF *      dbPtrVal(DBREF *, DBREF *, int);
DBREF *      dbMemPtr(DBREF *, void *);
int          dbPtrSub(DBREF *, offset, DBREF *, offset, scale);
```

Figure 8: EPVM Pointer Manipulation Functions.

[2] Transient memory addresses are denoted using OIDs with a disk volume of 0.

is compiled with a maximum object size that it will pin entirely. This size is also used as a block size for objects bigger that the maximum, and large objects are pinned by the EPVM in chunks equal to the block size (always beginning at an offset within the object that is a multiple of the block size). A separate user descriptor is established for each object block when it is accessed, and the hash table used to map from persistent addresses into user descriptors is based on both the OID and the offset. Of course, this makes the mapping process somewhat more complex; each reference must establish a block number based on the offset of a given reference within an object, and this block number is then used together with the unique field of the OID in calculating the hash function and matching the hash table entry for a given reference.

5.2. OID to User Descriptor Mapping

Given these simple policy decisions on how objects are pinned as they are accessed, we can now describe how EPVM maps persistent addresses to previously established user descriptors and thus how persistent values are accessed and updated. The process used to extract a value from a persistent address specified by a db pointer and an offset is illustrated in Figure 9. First, the internal and external offsets are combined and the result is split into an object block number and an offset within the block (step 1). Next, the block number and the unique field of the OID are used to compute an index into the interpreter's user descriptor hash table (step 2 in diagram). Then, the OID and block number of the hash table entry specified by this index are compared to those of the specified database address (step 3 in diagram). If these match, the user descriptor pointer in the hash bucket is used to get the correct base pointer for the object region (step 4), and the base pointer is combined with the object block offset to form the address of the requested data in the ESM buffer pool (step 5). Finally, this address may be used to obtain the requested data value. In step 3, if the comparison fails, EPVM repeats the comparison for any other entries in the same hash chain. If all of the comparisons fail, or if there was no initial entry in the hash table, a fault handling routine is called to establish a user descriptor for the requested address. For updates, the user descriptor pointer and object block offset are used as parameters and the sm_WriteObject call is used to update the object.

5.3. Buffer Management Policy

So far we have explained how object pages are pinned in the buffer pool and how persistent addresses are mapped to established user descriptors. Since the persistent object space maintained by ESM is potentially much larger that its buffer pool, however, the buffer pool may eventually become full. If a program accesses more objects and object blocks than the buffer pool can hold at one time, at some point EPVM will be unable to pin a newly accessed object. This condition is handled by releasing currently pinned objects and object blocks until sufficient space becomes

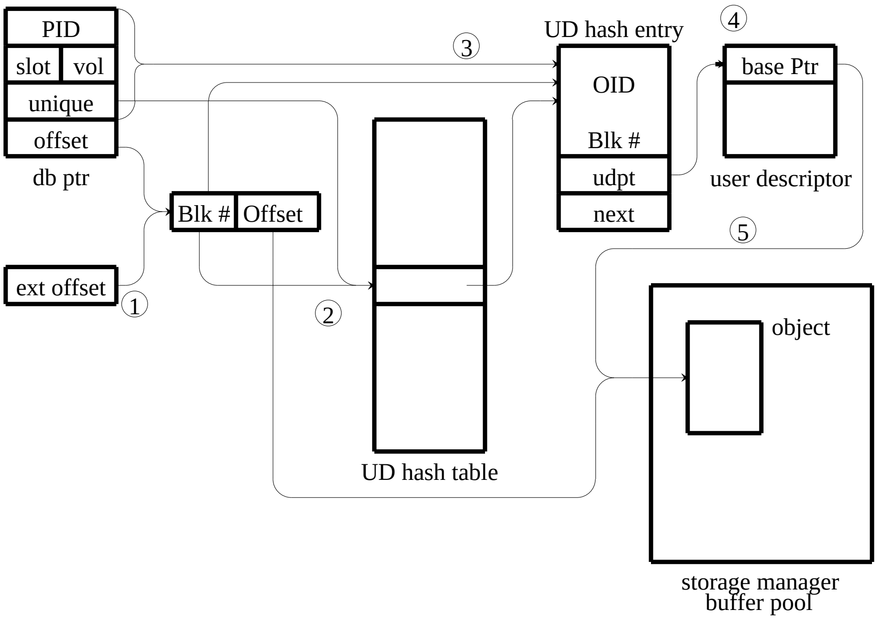

Figure 9: Access Through a Persistent Pointer.

available to handle the new request. The policy currently used by EPVM for releasing objects is a Clock-based LRU approximation based on reference bits kept in the hash table entries. When space is exhausted, EPVM sequentially scans all entries in the user descriptor hash table, freeing entries that have a previously cleared reference bit, and clearing any set reference bits, until sufficient space has been made available.

5.4. Pointer Swizzling

The general idea of ''pointer swizzling'' is to maintain two distinct forms for addressing persistent objects, one for disk-resident objects and another for memory-resident objects [Cock84, Maie86]. Pointers are transformed back and forth during the course of program execution. For example, a pointer in disk format could be transformed to memory format at its first use within a given program run, or all the pointers contained in a disk-based object could be transformed to the memory form when the object is first read from disk into main memory. (Note that the second method requires either a transitive closure reachability analysis or memory protection tricks to work properly, though.) Of course, if a persistent object contains pointers that are transformed into main memory form, its pointers must be transformed back to disk form before the object may be written back to disk.

Certain aspects of the design of the E language make a comprehensive pointer swizzling scheme difficult to implement. One problem is that the type system of E is based on C++, and ultimately C, making it difficult to determine what portion of an object represents a pointer. Union types can result in ambiguity about whether a particular data area represents a pointer or not, and casts can result in arbitrary data areas being treated as pointers regardless of their declared type. These issues could be reasonably dealt with by language restrictions and by maintaining additional data structures indicating the location of valid pointers.

A more difficult problem arises if one wishes to do incremental unswizzling. A fundamental goal of EXODUS is to reasonably handle object spaces that are larger than physical or even virtual memory, and to allow programs to access such spaces without restrictions. Thus, we did not wish to restrict the number or size of objects that a program may access. This implies that EPVM must be able to unpin objects in the buffer pool at rather arbitrary points during program execution in order to allow buffer pool space to be reclaimed. In order to unpin a particular object under a pointer swizzling scheme, it would be necessary to keep track of all valid pointers that could possibly point to the object so that they can be unswizzled (or marked invalid) when the object is unpinned or flushed to disk. While this may be possible in principle, the practical difficulties of doing this in a C-based language are considerable.

Another problem with pointer swizzling in E lies in the indirect nature of the ESM interface to buffered objects. Recall that when the sm_ReadObject call is used to pin an object or a region of an object in the buffer pool, ESM returns a pointer to one of its internal user descriptors rather than a direct pointer into the buffer pool. Because of this, it is impossible to transform persistent E pointers into direct memory pointers. Instead, their swizzled form contains the user descriptor pointer returned by ESM, and references through a swizzled pointer must indirect through the associated user descriptor.

5.4.1. What to Swizzle

Due to the considerations discussed above, rather than attempting a general solution, we implemented a limited form of pointer swizzling in EPVM. The solution implemented is based on transforming a limited set of persistent pointers — local variables which are db pointers. Space for keeping track of these variables is allocated from a global array of db pointers, with a simple stack discipline being employed at function entry and exit. To illustrate the approach, consider the following simple E code fragment:

```
f() {
    dbint *ipt, *ipt2;
     ...
    ipt = ipt2;
}
```

At function entry, a runtime support routine is called to allocate space in the global array for the db pointer variables `ipt` and `ipt2`. A main memory pointer is then used as a frame pointer for accessing the db pointer variables. In the example above, space must be allocated for two db pointers, and they will be accessed using the frame pointer with an appropriate compiler-assigned index. The resulting C code is shown in Figure 10. There, references to the variables `ipt` and `ipt2` have been replaced by indirect references through the memory pointer variable `_db_fpt`. This variable acts as a frame pointer for the portion of the db pointer stack active in this function.

By restricting the location of db pointers that may be swizzled to a single fixed size array, we simplify some of the problems associated with pointer swizzling. The set of currently active local pointer variables is easily

```
/* internal globals for db pointer stack */
DBREF _db_pstack[MAX_DBSTACK];
DBREF * _db_stacktop = _db_pstack;

f() {
    DBREF * _db_fpt = _db_stacktop; /* set frame pointer */
    _db_stacktop += 2; /* make room for 2 local pointers */
     ...
    _db_fpt[0] = _db_fpt[1];  /* ipt = ipt2 */
    _db_stacktop = _db_fpt; /* release stack space on exit */
}
```

Figure 10: Managing Local Db Pointers.

determined from the bounds of the allocated portion of the pointer stack. If object release becomes necessary due to buffer pool overflow, the problem of invalidating pointers to released objects is reduced to scanning the active portion of the pointer stack for references to those objects. The pointer stack can also be used as a hint indicating what objects are likely to be accessed as the program continues execution. Lastly, all EPVM routines are passed memory addresses for their db pointer arguments, so EPVM routines can easily determine whether their arguments are candidates for swizzling by checking to see if the address of the pointer lies within the range of the local pointer stack. This allows the swizzling operation to be handled easily within the normal EPVM pointer interpretation code; no special code needs to be generated by the compiler.

5.4.2. Swizzled Pointer Format

As mentioned earlier, due to the indirect nature of the ESM interface, the swizzled form of a 16-byte db pointer contains a user descriptor pointer that can be used to access the data that the swizzled pointer refers to. When a pointer is swizzled, the 4-byte unique field in its OID is replaced with the user descriptor pointer, and the volume ID field within the OID is used as a tag to indicate the swizzled state of the pointer. In addition, the page ID field of the OID is used to store a pointer to the user descriptor hash table entry that corresponds to the user descriptor. As a result, swizzled pointers always contain sufficient information to be directly converted back into their unswizzled form.

5.4.3. Using Swizzled Pointers

The process for accessing data through a swizzled pointer under this scheme is shown in Figure 11. First, the volume ID field of the OID is tested. If its value is 0 (the transient memory volume ID), the user descriptor pointer stored in the unique field of the OID is used to obtain the address of the object in the ESM buffer pool (step 1). The offset portion of the pointer is then added to the address obtained in step 1, along with any external offset (step 2). The resulting address can then be used to obtain the referenced data. The swizzled form of a db pointer provides a direct enough access path so that the compiler can reasonably generate inline code to access objects in the ESM buffer pool if desired, rather than always using the EPVM interpretive interface. For example, consider the following E code fragment:

```
dbint i;
dbint * ipt;
  ...
i = ipt[3];
```

Given our stack-frame-based approach to allocating local db pointers, the assignment expression can be compiled into C code as follows:

```
_db_fpt[0].volid ==0 ?
    (*(long *)_db_fpt[0].udpt->basePtr + _db_fpt[0].offset + 12) :
    dbLongEval( &_db_fpt[0], 12);
```

In this example, if the pointer is not in the swizzled form, the EPVM function dbLongEval is called to get the value (as if no swizzling were being done). If the pointer can legally be swizzled, swizzling will take place as a side effect of the dbLongEval call. Pointers are transformed into the swizzled form by the first access using the unswizzled form of the pointer; they are transformed back to unswizzled form only when the object that the pointer refers to needs to be released from the ESM buffer pool. Finally, even if inline code generation (a compile-time option) is

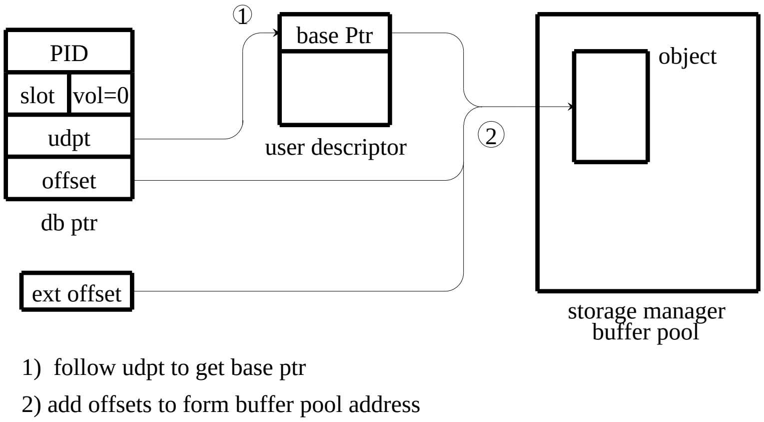

Figure 11: Access Through a Swizzled Pointer.

not enabled, swizzled pointers allow improved efficiency within the EPVM interpreter itself as well.

5.4.4. Swizzling Limitations

As previously stated, pointer swizzling in E 2.0 is limited to local pointer variables. In some cases, this limitation can be overcome by the programmer though the use of explicit temporary pointers if efficiency is especially critical. There is another serious limitation of pointer swizzling in this environment, however, which arises when EPVM has not cached an entire object. Recall that large objects are cached in blocks, and that user descriptors for large objects point to blocks rather than whole objects. Since a db pointer may point to an arbitrary offset within an object, the range of bytes accessible from a swizzled large object pointer is limited by the block boundaries that surround the location indicated by the pointer. Since this can vary arbitrarily at runtime, pointer swizzling is not implemented for pointers into large objects. Pointers to large, multi-block objects are always kept in their db form and are not swizzled.

6. EPVM PERFORMANCE

In this section we present a detailed, low-level analysis of the performance we have obtained using the EPVM-based approach to persistence described here. We give the costs of the basic interpreter functions under best case conditions, which is when a reference is made to an object that was previously cached by the interpreter when no hashing collisions occur. The costs are presented in terms of execution cycles for a MIPS-based processor; for the most part, one cycle corresponds to the execution of a single machine instruction. The results presented here were obtained using the ‘‘pixie’’-based profiling facility on a DECstation 3100. One cycle corresponds to 80 nanoseconds of clock time on a 12.5 MHz machine.

6.1. Access Function Costs

In the preceding section, we described the implementation of EPVM in terms of the mapping of db pointers to ESM user descriptors. As we shall see, this mapping is the most significant cost in the current EPVM implementation, but the cost of accessing ESM's buffer pool through user descriptors is also significant. We begin our analysis by discussing the cost of executing a single evaluation function in the interpreter, dbLongEval. This cost is typical of the set of functions that provide access to persistent data. The cost was measured at 47 cycles, including both call setup and function execution, broken out as indicated in Table 1.

Of the 47 total cycles, we estimate that about 9 cycles (address computation and dereference, tests for non-persistence) are costs that would occur in some form under any implementation using the user descriptor based interface of ESM. The marginal cost associated with EPVM is thus about 38 cycles per reference. This marginal cost is considerable, and thus needs to be justified. The main justification for this cost is that it is relatively small compared to the ESM costs for establishing and releasing user descriptors. The minimal cost measured for this pair of ESM operations was 497 cycles, not including call setup costs. Viewed another way, avoiding one ESM pin/unpin operation pair is sufficient to compensate for the overhead of about 13 EPVM object value accesses using this interface. The cost of using the EPVM interface and implementation is clearly justified compared to the naive alternative of creating and deleting a user descriptor for every access; of course, this is a strawman argument. We

Operation	Cycles
Call setup	3
Function entry and return overhead	7
Test for non-persistence	2
User descriptor hash lookup	26
Set reference bit	2
Address computation and dereference	7
Minimal reference cost	47

Table 1: Access Function Costs.

do note, though, that for a working set that fits entirely in the ESM buffer pool, each object will be pinned and unpinned exactly once by EPVM. Under these conditions, neglecting the cost of the initial pin operation, any approach based on dynamically establishing user descriptors would have to, on average, use each user descriptor for at least 13 accesses in order to be more efficient than the EPVM implementation.

The analysis above shows that the dominant cost in EPVM is that of the hash-based OID to user descriptor mapping process. As such, we analyze this cost in more detail. As mentioned previously, the object caching scheme handles both entire small objects and blocks of large objects; the address mapping scheme must therefore consider both the OID and the offset within an object when translating a particular reference. Moreover, this cost is incurred for both large and small objects, as a db pointer by itself does not indicate whether it points to a small or large object. If the large/small object distinction were eliminated, and objects were always cached in their entirety, the cost of the hash lookup would be reduced to 19 cycles. In the limit, the best possible performance would be obtained by using 4-byte OIDs instead of the current 12-byte OIDs. We estimated the performance of such a best-case system by modifying EPVM to utilize only the unique field of the OID when hashing. This reduced the cost of the user descriptor hash lookup to 12 cycles. Thus, paging large objects and using 12-byte OIDs each impose a 7-cycle cost in interpretation relative to the simplest possible scheme.

6.2. Update Function Critical Path Costs

The basic costs of the assignment functions include all of the basic costs of a typical access function, plus the cost of calling ESM to update the referenced data and executing some additional code to distinguish persistent and non-persistent references. For dbLongAssign, which is typical, the total cost is 63 cycles in the function interpreter plus an additional 109 cycles for the ESM update routine. The additional 19 cycles here, as compared to dbLongEval, reflects the cost of setting up the ESM update call as well as the cost of passing an additional argument. As can be seen from the cycle counts quoted above, the cost of calling ESM to update the referenced object is considerably greater than the costs associated with the functional interface. Moreover, the overhead of updating objects is likely to increase considerably with the forthcoming ESM release (with concurrency control and recovery); the relative cost of using the EPVM interface is thus not likely to be the performance bottleneck for updates.

6.3. Costs Using Swizzled Pointers

We can now compare the costs just discussed with the cost of references using the limited form of pointer swizzling described in Section 5.4. We consider four methods of accessing a persistent value here — dereferencing a swizzled db pointer using inline code, dereferencing a swizzled db pointer using the EPVM interface, dereferencing an unswizzled pointer and swizzling it via the EPVM interface, and dereferencing an ‘‘unswizzleable’’ pointer. The total cost for each of these cases is listed in Table 2.

The 11-cycle cost for the inline dereference of a swizzled pointer is a clear improvement, though it does carry a cost in terms of added code size. The additional cost associated with using a swizzled pointer within EPVM reflects function call overhead; this correlates well with the estimate of the call overhead made in Section 6.1. The difference between the swizzled and unswizzled reference costs using the EPVM interface also correlates with our previous estimate of the cost of the hash-based user descriptor lookup used for persistent pointers. The 56-cycle cost for an initial access reflects the cost of both the unswizzled access process and the swizzling operation on the pointer used for the access. Thus, the additional cost for performing the swizzling operation is seen to be 9 cycles. The additional 1-cycle cost for the unswizzleable case here, relative to the cost of 47 cycles in the non-swizzling case in Section 6.1, is due to a failed test for swizzleability that does not take place if swizzling is not used at all.

Operation	Cycles
Inline dereference of swizzled pointer	11
Interpreted dereference of swizzled pointer	20
Interpreted dereference and swizzling of unswizzled pointer	56
Dereference of unswizzleable pointer	48

Table 2: Swizzled Pointer Costs.

6.4. Faulting Costs

The cost of actually reading an object into memory from disk is difficult to estimate, as it is highly dependent upon the state of ESM at the time. We thus did not try to estimate that cost here. Instead, we estimated a minimal cost for performing the pin and unpin operations that establish and release a user descriptor. This was done by measuring a simple test program that called ESM directly, repeatedly pinning and then unpinning the same small object via calls to sm_ReadObject and sm_ReleaseObject. After the initial call, the object is resident in the ESM buffer pool, so the only costs involved after that should be the bookkeeping needed to manage the user descriptor. The simplest sm_ReadObject/sm_ReleaseObject pair was measured as taking 497 cycles, as mentioned earlier. The additional cost in the EPVM interpreter for establishing a hash table entry for a new user descriptor was measured at 77 cycles (in addition to the normal cost of the EPVM call creating the fault). Thus, the minimal cost for establishing an OID to user descriptor mapping is 574 cycles.

6.5. Discussion

From these discussions, we conclude that the cost of accessing persistent objects through the EPVM interface is moderately expensive but not prohibitive. Without pointer swizzling, on a 12.5 MHz MIPS architecture processor such as the DECstation 3100, the cost can be as low as about 4 microseconds per access to persistent storage. The cost of using the functional EPVM interface for persistent pointer evaluation is expensive, but the overhead is not unreasonable compared to the cost of using user descriptors for access, especially when compared to the ESM costs for establishing user descriptors and updating objects. The cost of translating OIDs to user descriptors is the dominant cost in the current implementation of EPVM; this cost could be reduced if the OID format used by ESM were simplified. Finally, swizzling pointers can also reduce access costs considerably, although indirect access through user descriptors still imposes a significant additional cost compared to a normal main memory pointer access. We expect the current implementation of E to be reasonably efficient for applications that operate on threaded, persistent data structures with sufficient locality of reference that a reasonable working set can be kept buffer-resident.

7. CONCLUSIONS

We have described in detail one approach to implementing persistent data access in the E programming language. This approach, which is based on interpretive access to persistent objects though a functional interface, is quite general. Implementation of compiler support for this approach was straightforward; the efficiency of the implementation depends largely on the efficiency of the underlying interpreter. We described the design, implementation, and initial performance measurements of an EPVM interpreter that is reasonably efficient as compared to the underlying EXODUS storage manager. We argued that the costs of EPVM interpretation are reasonable given the complexity of the underlying persistent object store. Finally, we also showed that pointer swizzling techniques can improve performance markedly, even given the indirect nature of the EXODUS storage manager interface.

The approach that we have described here is somewhat similar in flavor to the abstract machine approach used in the implementations of PS-Algol and Napier [Atki83, Dear88], but EPVM is much more limited in scope. The abstract machines for PS-Algol and Napier implement many aspects of their languages, and they interact much more strongly with the language type systems and programming environments. EPVM is primarily restricted to providing access to persistent storage. The implementation of E depends on support provided by the underlying C compiler for code generation, and on the Unix linkers and debuggers for programming environment support; most type-dependent issues are handled statically by the E compiler.

It is also interesting to compare EPVM to the approach taken by Richardson in the 1.2 E compiler [Rich90]. In 1.2 E, the compiler generated explicit pin and unpin calls to establish the user descriptors corresponding to each pointer value used by an E program to access data. The goal was to use static analysis techniques to establish the minimal number of user descriptors possible, and to be able to amortize the cost of pin and unpin operations over many accesses through each user descriptor. It is easy to construct examples where this approach is clearly superior, but one can also construct examples that traverse threaded data structures and require many pin and unpin operations relative to the number of persistent accesses performed. For example, under the 1.2 E approach, a traversal of a list of persistent objects, where the list is linked by db pointers, would likely require one pin and unpin operation for each node traversed. Under the EPVM approach, the first traversal of the list would also need to pin all of the objects on the list; however, subsequent traversals would require no pin or unpin operations. If few data accesses were made on each traversal, the pin and unpin overhead might be the dominant cost. We conjecture that the class of problems handled reasonably well by the EPVM-based implementation of E is somewhat broader than that which would be handled well, at least without interprocedural analysis, by the 1.2 E approach.

Despite these arguments, it is clear that the cost of persistent object accesses using the techniques described here is quite high compared to that of normal memory accesses. Even our pointer swizzling costs did not come close to memory access costs. In contrast, a virtual memory based approach offers the possibility of performing accesses to persistent storage with a relatively small marginal cost per accessed object [Shek90]. However, such approaches have other problems, such as address space limitations and the difficulty of managing a shared, persistent address space. We feel that the approach shown here may prove to be a reasonable compromise, and it also offers a flexible environment for further research.

REFERENCES

[Atki83] M. Atkinson, M., *et. al*, "Algorithms for a Persistent Heap," *Software — Practice & Experience*, Vol. 13, 1983.

[Care89a] 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.

[Care89b] 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] W. Cockshott, *et. al*, "Persistent Object Management System," *Software — Practice and Experience*, Vol. 14, 1984.

[Dear88] A. Dearle, *On the Construction of Persistent Programming Environments*, Ph.D. Thesis, Computational Science Dept., University of St. Andrews, St. Andrews, Scotland, June 1988.

[Maie86] D. Maier, "Why Object-Oriented Databases Can Succeed Where Others Have Failed," *Proc. of the 1st Int'l. Workshop on Object-Oriented Database Systems*, Pacific Grove, CA, Sept. 1986.

[Rich89a] 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.

[Rich89b] J. Richardson and M. Carey, "Persistence in the E Language: Issues and Implementation," *Software — Practice & Experience*, Vol. 19, Dec. 1989.

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

[Shek90] E. Shekita and M. Zwilling, "Cricket: A Mapped Persistent Object Store," *Proc. of the 4th Int'l. Workshop on Persistent Object Systems*, Martha's Vineyard, MA, Sept. 1990.

[Stro86] Stroustrup, B., *The C++ Programming Language*, Addison-Wesley, Reading, MA, 1986.