The Mneme Persistent Object Store[*]

J. Eliot B. Moss[†]

COINS Technical Report 89-107
October 1989

Object Oriented Systems Laboratory
Department of Computer and Information Science
University of Massachusetts, Amherst
Amherst, MA 01003

[*]This project is supported by National Science Foundation Grants CCR-8658074 and DCR-8500332, and by Digital Equipment Corporation, Apple Computer, Inc., GTE Laboratories, and the Eastman Kodak Company.

[†] Address: Department of Computer and Information Science, Lederle Graduate Research Center, University of Massachusetts, Amherst, MA 01003; telephone (413) 545-4206; Internet address Moss@cs.umass.edu.

Abstract

The Mneme project is an investigation of techniques for integrating programming language and database features to provide better support for cooperative, information-intensive tasks such as computer aided software engineering. We report here on the Mneme persistent object store, discussing the original design, the construction of the initial prototype, and approaches being considered for the next prototype. Mneme stores *objects*, with a simple and general format, and preserves the identity of the objects and their structural relationships. Mneme's goals include portability, extensibility (especially with respect to object management policies), and low overhead. The model of memory that Mneme aims to present is a single, cooperatively shared heap, distributed across a collection of networked computers.

This paper has been submitted to *ACM Transactions on Information Systems.*

1 Introduction

The Mneme project is an investigation of techniques for integrating programming language and database features to provide better support for cooperative, information intensive tasks such as computer aided software engineering. We report here on the Mneme persistent object store. We discuss in turn: goals of the effort, conceptual design of the store, construction of the prototype, lessons learned, directions for the next prototype, related work, and our conclusions.

1.1 Goals for the Store

The Mneme project's overall goal is to provide better support for cooperative, information intensive tasks (which we sometimes more loosely call *design* tasks). These tasks include computer aided design (CAD), ranging from VLSI design through electrical, mechanical, and architectural design, as well as computer aided software engineering (CASE). They also include document preparation/publishing and office automation applications, as well as hypertext and other advanced information systems and tools to support group work. From the standpoint of storing information, salient features of these applications include their need to store and retrieve a considerable volume of highly structured information in a distributed system context, while supporting multiple people cooperating on large, not necessarily well-defined, tasks.

The Mneme store effort takes a particular approach to supporting these tasks, namely to provide the illusion of a large shared heap of objects, directly accessible from the programming language used to build the applications. Our hypothesis is that this model is an adequate foundation; the goal of the Mneme store effort is to produce prototypes that allow the hypothesis to be tested, by ourselves and by others.

Given the overall goal of the effort, we arrived at the following specific objectives for the store:

Goals for the Mneme Store

- The store should provide an appropriate notion of an *object*, such that structure and identity of objects are preserved.
- The store should have low overhead of use, and high performance for retrieval and storage.
- The store should be portable across a wide range of systems and usable from a variety of tools and programming languages.
- The store should provide mechanisms whose policies can be changed and extended.

- The store should be distributed, in a heterogeneous client/server workstation environment, with as much transparency as possible while substantially respecting autonomy of resources.
- The store should support modularity of data within its heap model.
- The store should support further research into cooperative data sharing techniques, integration with programming languages (i.e., persistent programming languages and database programming languages), and models incorporating distributed execution as well as distributed storage.
- The store should allow the use of existing lower level storage managers and servers when that is consistent with the other goals.

We now consider the rationale for each of these goals.

Object Structure. The applications of interest use complex and highly structured data, which can be thought of as directed graphs where the edges are pointers and the nodes are "objects". We must be able to preserve this structure, and it is important that we do so as simply as possible. The additional features of objects (dynamically invoked methods, inheritance hierarchies, etc.) are desirable for many applications. On the other hand, typing and invocation mechanisms vary considerably across programming languages, even in the object-oriented realm. For broadest applicability of the store's features, we restricted its goals to providing object structure and identity only, not object execution semantics. Thus a Mneme object is a collections of fields accessed via an object identifier, but Mneme objects have no types/classes, methods, or inheritance, since we want to allow a variety of type, inheritance, and invocation mechanisms to be built on top of Mneme. For similar reasons, we restricted the store's notion of "edge" or "relationship" to simple pointers. More sophisticated data models can be built in terms of the simple Mneme objects. This keeps Mneme lightweight and general (and somewhat "low level").

Performance. One of the motivations of the overall Mneme project in exploring language-database integration is the poor performance of non-integrated approaches. (Another is the relatively poorer functionality and less desirable semantics of non-integrated systems.) Further, many cooperative, information intensive applications, especially CAD, are quite demanding. Unpublished estimates[1] for performance requirements include being able to do at least 100,000 references per second to fields of objects (to support "dragging" items on a workstation display screen) and being able to retrieve 10,000 "typical size" objects per second from external storage into memory. "Typical size" for languages such as CLU [Liskov *et al.*, 1977; Liskov *et al.*, 1981], Smalltalk-80[2] [Goldberg and Robson, 1983], and

[1]There are no detailed references for these numbers.

[2]Smalltalk-80 is a registered trademark of PARC Place Systems, Inc.

Trellis/Owl[3] [Schaffert *et al.*, 1986] appears to be about 30–40 bytes, but this is rather informally collected evidence. In any case, the desired retrieval rate is a substantial fraction of the bandwidth of a magnetic disk on a typical workstation.

Portability. In order to justify the effort of building the store, as well as to encourage others to experiment with it and develop evidence as to the appropriateness and value of the shared distributed heap approach, the store should be usable on as many systems as possible, and from a reasonable collection of existing programming languages. While one intent is to support integration with programming languages, integration should not be *required* for effective use of the store.

Extensibility. The applications we wish to support vary considerably in their semantics, and have performance characteristics and demands that are not well understood. Because of this variability and lack of knowledge, it is important that the store allow its object management policies, especially those affecting performance, to be tuned, controlled, and extended. Such policies include clustering, pre-fetch, caching, and concurrency control. To the extent possible and consistent with the other goals, the store should also anticipate and support application specific extensions to its basic semantics.

Distribution. The hardware setting (workstations with local area networks) and general software framework (the client/server model) are dictated by what is available, both to the target applications, and to us in our own research environment. Physical distribution *per se* is more of an implementation concern. The deeper issue is supporting and trading off between sharing (multiple concurrent users) and autonomy (individual control of resources, ranging from the physical (servers and disks) to the conceptual (subcollections of objects)). Note that the tradeoff must not be fixed, since different organizations and different applications have different relative needs for convenient sharing and for autonomy.

Modularity. In a large space of objects it is crucial for users and applications to be able to identify and manipulate meaningful subsets of objects. It should also be possible to extract and insert such subsets of objects, so as to send useful collections of data from one place to another, for backup, etc. Modularity will also tend to support autonomy and reduce the scope of possible damage to the store by a runaway program.

Basis for further research. To a certain extent, this goal summarizes the overall goal of the effort to build the store, since that effort is part of a larger project. There are specific implications, though, deriving from the project direction. The store should be suited to integration with some programming languages (current efforts include Smalltalk-80 and Modula-3 [Cardelli *et al.*, 1988]), it should allow experimentation with techniques for sharing data cooperatively, and it should be a suitable basis for an architecture examining issues of distributed execution as well as distributed storage.

[3]Trellis is a registered trademark of Digital Equipment Corporation.

Existing storage managers. The main justifications for this goal are preventing duplication of effort through building on others' work where possible, and encouraging other researchers to use and evaluate the store by increasing the chances that they can use it with their existing software and data. We feel it important, though, that inter-operability with existing software should not be allowed to compromise the primary research goals of the effort.

2 Concepts of the Mneme Store Design

Before undertaking implementation of the Mneme store prototype, we considered the desired concepts and semantics, which we now discuss. We have treated the design as an ideal to be approached via a series of prototypes, and to be changed as experience is gained from those prototypes. In presenting the concepts and their semantics below we refer back to our goals for the store to help explain the rationale for some of the design features and choices.

2.1 Objects

A Mneme *object* consists of three parts: *slots*, *bytes*, and *attributes*. The attributes are a small number of bits (on the order of 8) intended for indicating such properties as whether an object is read-only. The design does not specify their use—attributes are more of a hook for extensions, though some specific potential uses will be mentioned. The bytes are simply a vector of 8-bit bytes, indexed from 0 to $b-1$, where b is the numbers of bytes in the specific object, which is specified when the object is created. The ability to change the size of an object is not part of the original design; we did not feel it important enough to justify the design and implementation effort.

The slots part is perhaps the most interesting aspect of an object. This part is a vector of 32-bit slots, indexed from 0 to $s-1$, where s is the number of slots in the object, specified at object creation time, analogous to the number of bytes. Each slot contains one of three things: a distinguished *empty* value, an immediate 31-bit integer value, or an *object identifier* (id). We will have considerably more to say about object identifiers later; for the moment, it should suffice to know that each object has a distinct identifier, and that the identifier allows the object to be located and accessed.

We developed this object concept for several reasons, most easily discussed by considering alternative notions of object. Perhaps the simplest notion of object, used in some well-known designs (see, e.g., [Skarra *et al.*, 1987; Hornick and Zdonik, 1987; Carey *et al.*, 1986]), is as a vector of bytes alone. We separate the slots so that we can find and manipulate the object ids. This allows us to garbage collect in the store. It also allows us to change the stored form of the identifiers. In short, we had some implementation techniques in mind, which will be discussed later. Another reason the slots/bytes style of object was appealing is that it closely matches the requirements of Smalltalk and Trellis/Owl.

Comparing with Smalltalk and Trellis/Owl, our object concept omits any notion of "class" or "type". This is not a problem for our format, though, one can simply establish a convention for those languages that the first slot contains the class/type information. We omitted any class/type (or for that matter, method or method invocation) mechanism from the design to keep things simple, and perhaps more importantly, not to impose any particular type or inheritance model, increasing the number of languages with which the store could be used conveniently. This is not to say that our object model directly supports sharing of the *same objects* between programs written in different languages, since the variation in language semantics may make that very difficult to do in any automatic and transparent way, but the Mneme store does nothing to *prevent* such sharing if the languages would otherwise support it.

Finally, the contents of slots are tagged. The main reason this was done was to fit with Smalltalk and Trellis/Owl. However, consider that languages implementable without tags presumably know types well enough to distinguish pointers from non-pointers at compile-time. If that is true, then non-pointer types can be stored in the bytes area with no tagging overhead, and pointer types stored as ids in the slots area (incurring some tag setting and checking overhead).

Another alternative object format to consider is something along the lines of records in Pascal, which would match up better with C++ [Stroustrup, 1986] objects, for example, than our segregated slots and bytes parts. Such a format would mix slots and bytes, and, since we need to know where the ids are, would require descriptors. We did not feel the complexity of descriptors was justified.

Finally, we could have chosen more of an object and relationship model, rather than just objects and pointers (ids). This would not have matched up well with existing programming languages. It would also have been much more difficult to implement and might have introduced performance problems. We also felt that such models could be built on top of the Mneme model—that Mneme provides a low level abstraction of storage, and advanced features such as relationships and arbitrary properties and attributes can be built in terms of this structure. Of course an "object" at the higher level of semantics would probably not correspond directly to a single Mneme object. Our choice is consistent with providing a low overhead, simple, and general base on which to build, without imposing very specific semantics.

To summarize, the format we chose for objects embodies the desired structure and minimal object semantics—every object has an id, and the object's slots describe its pointer relationships to other objects. This structure is simple, general, and efficient in storage and access.

2.2 Handles

A *handle* is a data structure providing efficient access to the internals of an object. To manipulate an object in the Mneme store, one first acquires a handle, by presenting the id of the object to be accessed. When access is no longer desired, the handle may be destroyed. The actions of creating and destroying handles serve several purposes. First of all, those actions delimit a period of time during which an object is actively being accessed and must be available to the application. This has implications for concurrency control and consistency management, discussed in more detail later. Creating a handle is the time at which an object may be fetched from the store, causing what we call an *object fault*. Of course, the object may already be resident in main memory. Once a handle is acquired, though, the object is *known* to be available, and further checks are not required.

Even when an object is resident, to create a handle the object must be located given only the id of the object. The handle data structure can embed direct pointers to the parts of the object, avoiding object id lookup on every slot or byte access. Furthermore, the object format can be decoded once and the relevant information stored in the handle data structure in a less compact but computationally more efficient form.

The main drawbacks of handles is their size and the imposition of an additional level of management between the application and the object. This level of management is also a level of abstraction, though, and improves the robustness of the system. Handles appear to be adequate for direct use of the store by applications, but they may not be the best approach for tight integration with a programming language. This point is examined more closely in our critique of the prototype.

2.3 Files and Locality of Ids

Mneme groups objects together into units called *files*. A file of objects can be separately named and located within the overall distributed store. A typical implementation of the concept would associate individual files with servers, though the association could change with time, and this implementation strategy is not dictated by the Mneme store design. Every Mneme object resides in a file, and there are no provisions in the interface for moving objects from one file to another.

Files are a convenient unit for storage, and provide modularity of the object space, one of the stated goals. We intended that Mneme files, or groups of related files, be reasonable units of backup, recovery, garbage collection, and transfer between different Mneme stores. Transfer in particular, and backup and recovery to some extent, depend also on the style of use by applications, since a file's objects can refer to objects in other files, and hence such references might occur out of context if a file is transferred or restored without regard for its references to/from other files.

In addition to modularity, files allow us to take advantage of locality of reference, and to provide a substantial degree of autonomy, as follows. Object ids, as stored in objects within the store, always name objects within the *same file* as the object containing the id. This allows ids to be relatively short—on the order of 30 bits. References to objects in other files are made by referring to *forwarder* objects within the same file. A forwarder is an ordinary object with the exception that it has a particular attribute bit set. Because a forwarder is an ordinary object (except for the setting of the one bit), it can contain arbitrary information to name and locate the intended target object. Typically a forwarder would indicate the file and the name of the object within the file. Because of the very general nature of the forwarder mechanism, we can support a variety of cross file reference techniques, including ones where the binding is interpreted contextually, similar to UNIX[4] environment variables or VAX/VMS[5] logical names. Forwarders provide a substantial opportunity for extension of the basic Mneme store functionality.

To provide autonomy, as well as to support garbage collection, the actual ids of objects in a file are not used to name the objects from "outside" of the file. Rather, each file has a table mapping external names to internal object ids. The external names, together with some unspecified means for naming files, provide the only form of (potentially) immutable persistent object identifier in the design. Thus, such a name should be assigned to any object that an application or user may need to name explicitly at a later time. We do assume, though, that relatively few objects will need such names. Autonomy is supported in that the mapping table "shields" the objects inside a file from external manipulation. The mapping is also a place where extensions can be provided, such as for access control or pre-fetch of a collection of objects when a particular object is first accessed.

In addition to the table that maps external names to internal ids, each file also has a distinguished *root*, a slot that can be set at will to indicate a starting place for retrieving objects in the file. Note that the root and the mapping table are essential, since ids cannot simply be synthesized and presented to the Mneme store functions with any reliability. This is because the ids of objects within a file can be reassigned, allowing reclustering, garbage collection, reuse of the limited space of ids, and even explicit object deletion.

There is considerable synergy between the modularity, autonomy, extensibility, and com-

[4]UNIX is a registered trademark of AT&T Bell Laboratories.

[5]VAX and VMS are registered trademarks of Digital Equipment Corporation.

pactness (short ids) provided by the Mneme file concept. The concept also maps well onto existing file systems as well as the client/server model of distributed data storage and access. In addition, the model is a relatively simple one. It is useful, though, to compare it with alternative approaches. We have presented more detailed arguments in [Moss, 1989].

One possibility would be a large virtual memory, where the addresses name bytes or words. Problems with that approach include lack of object semantics, difficulty in garbage collection, the need for rather long addresses, and poor support for autonomy, modularity, and extensibility. The main advantages are simplicity and familiarity, though the simplicity may be only apparent, not real, when we consider administration of a large distributed store. Most of the problems mentioned occur for any byte/word addressed store, regardless of whether the virtual memory is more structured, e.g., segmented as on Multics [Organick, 1972] or the Intel 432 system [Intel Corporation, 1981; Organick, 1983].

Another alternative to the Mneme addressing scheme is immutable object identifiers. Sometimes provision of such identifiers is seen as equivalent to supporting object identity, though we argue that object identity and persistence of immutable *names* for objects are distinct ideas. For further discussion of the concept of identity, see [Khoshafian and Copeland, 1986]. Note that Mneme supports identity, in the sense of preserving the graph structure defined by object references, without the most frequently used kind of name (object ids) being immutable. While forwarders and mapping tables can support other semantics as well, we presume that they provide at least the capability to continue to refer to precisely the same object, so long as the object is not explicitly deleted. Hence, our design does provide immutable persistent names when they are required, but avoids their overhead in the many cases where they are not required.

Immutable object identifiers present two major problems, both related to performance. First, in a large system, they will have to be long. This might be alleviated if objects are grouped in a manner similar to our files, but if there is substantial movement of objects from file to file, performance problems of space (a mapping table) or time (forwarding addresses) results. The other performance problem is retrieval time. If object ids are immutable, as objects are reclustered over time the ids lose any power they might originally have had to provide a hint as to where an object is located. In a large store it is likely that two or more secondary storage accesses will be required to fetch an object: one to determine its location, and a second to retrieve it. The Mneme store (as we will see) can use object ids as substantial location hints and eliminate at least one secondary storage access on object retrieval, assuming a per-object id-to-location map would be too large to keep in main memory.

Gehringer has proposed variable length capabilities [Gehringer, 1979] and variable length names [Gehringer, 1989]. While his schemes obtain many of the advantages of our approach, they require special support hardware, and it is not clear that, even given the hardware, his approach would perform as well as ours.

In summary, the Mneme concept of a file as a modular collection of objects, as a local space of object identifiers, and as a unit of autonomy, supports several of our goals better than the alternative designs considered.

2.4 Pools and Strategies

While Mneme files are physical and logical units of grouping and naming objects, Mneme *pools* are logical, and not directly physical, groups of objects within files, and pools are not directly involved in object naming. Each Mneme object is associated with (stored in) exactly one pool, and that pool determines the policy under which the object is managed. A management *strategy* is a vector of routines for making individual policy decisions. A strategy is associated with a pool when the pool is created. Strategies can depend on pool specific variables, called *pool attributes.* Thus a strategy can be generic with specific parameters given by the pool's attributes.

Let us consider two examples of these concepts in action. First, when an object is to be created, one specifies the desired number of slots and bytes and the initial attribute settings. One also indicates the pool in which to store the object, and can provide an additional parameter to be interpreted by the pool's object creation policy routine. After preliminary argument checking, the Mneme object creation routine calls the pool strategy's object creation routine, which will choose where to place the new object (which also partially determines the new object id).

Another example of policy involvement occurs when a handle is requested for an object (given the object id), and the object is determined not to be resident. In this case, the object's pool (strategy) object fault policy routine is called. Clearly the object itself must be retrieved, but the policy routine can make decisions such as additional data to request (pre-fetch), how the object should be locked, and the buffer replacement policy to be used. If the requested object is a forwarder, then further policy and forwarding/mapping implementation routines would become involved.

In addition to providing a default strategy, the Mneme store design allows new strategies to be developed, and specifies means by which strategy routine vectors can be filled in by an application, so that policies need not be built in to the Mneme store code. If an application attempts to use an object whose pool strategy vector has not been set up, an error code is returned.

Pools support policy/mechanism separation and policy extensibility. The design leaves open as a research question the detailed design of the strategy routine interfaces. We hope that the pool concept will allow flexible approaches to object clustering, storage allocation, pre-fetch, concurrency, consistency, buffering/caching, and perhaps even security, versioning, and other issues not yet considered in detail in the Mneme project.

2.5 Transactions

Since the Mneme store is intended to support exploration of techniques for cooperative sharing, some kind of concurrency control and resiliency mechanism is required. A question we are still wrestling with is whether the features to be discussed here form an adequate basis for such exploration, or whether a more radical approach is needed. In any case it is useful to consider the design as it stands.

A Mneme store *session* is a period of interaction with the store, analogous to a login session with an operating system. A session establishes a context of use, including open Mneme files and ids of objects in those files. A *transaction* is a unit of work, concurrency control, and consistency, within a session. The intent is that so-called "long transactions" or "design transactions" that may span sessions are implemented at higher levels of abstraction, using Mneme store transactions to implement their various atomic steps. Sessions allow considerable caching of names, objects, and other file related information, increasing performance and providing a more convenient context of work for applications. Notably, the design guarantees that object ids (as used by a client) retain their meaning from transaction to transaction within a session.

The design includes a basic transaction model as well as facilities to build application-specific, non-serializable, models. The basic transaction model is quite straightforward. The set of committed transactions is guaranteed to be serializable, and individual transactions are guaranteed to be all-or-nothing (all effects installed on commit, none on abort). Serialization is done in terms of individual objects and whether they have been read or written. This specification allows considerable flexibility. For example, one can use read-write locking or optimistic concurrency control. With care, different pools can use different strategies.

An important point is that our design defines the meaning of abort and commit, but makes no guarantees concerning which transactions can or will commit. Thus, while read-write locking on a per-object basis is correct and allows high concurrency, one can lock in stronger modes (e.g., write lock when an object has only been read, in anticipation of a possible write) or in larger granularities (physical groups of objects, whole pools, or even entire files). This can allow simpler and more efficient techniques to be used to boost performance when the highest concurrency is not required (e.g., when a whole design file is checked out, it should not be necessary to lock each individual object in the file).

Another point to consider is that any locking or concurrency control is performed *im-*

plicitly in our design, as a side-effect of acquiring a handle, updating slots or bytes, etc. It remains to be seen if additional "hooks" are necessary for more precise specification of access to objects, or whether the pool/strategy approach to tailoring concurrency control and recovery is adequate.

The transaction semantics extension facility aims to provide a small number of minimal, general, primitives. These primitives supply three basic pieces of functionality: mutual exclusion, logging, and notification. This functionality is intended to be used to provide concurrency control, resiliency, consistency, and synchronization as needed. The logging and notification services have not actually been designed, so the facility is incomplete. There do not appear to be any fundamental problems in designing those services, though there are a number of choices to be made among competing approaches. The mutual exclusion facility is still of interest, though, because it uses other features of Mneme in novel ways.

Mutual exclusion is supported via *volatile pools*. When an application process acquires a handle on a volatile object (an object stored in a volatile pool), Mneme gives the process exclusive access to that object until the handle is released. At that time the object may be accessed and/or updated from other processes. Thus, handles provide "short locks" on objects. This primitive is adequate for building any desired concurrency control semantics, though there are interesting performance questions, such as how best to cache volatile objects. There are also remaining semantic questions related to resilience. (If there is a crash, what state should be restored after the crash?)

In sum, the Mneme store design includes a basic transaction facility and a sketch of transaction extension facilities. In addition to transactions, which have "traditional" database system semantics (but are specified so as to allow a variety and mixture of strategies in implementation), the design also includes the notion of a session. Sessions provide a scope for naming objects and allow for caching across transactions.

3 The Initial Prototype

We have built a working prototype Mneme store based on the design presented above. The prototype is written in C [Kernighan and Ritchie, 1978] and has been run on the VAX/VMS, VAX/Ultrix,[6] and SunOS[7] operating systems, and should run under other systems supporting C, if they have adequate memory and disk capacity. The system is designed to be highly portable, and the choice of C was made largely because of the wide availability of implementations of C, especially within the research community. C also supports the

[6]Ultrix is a registered trademark of Digital Equipment Corporation.
[7]SunOS is a registered trademark of Sun Microsystems.

type-unsafe operations necessary for such things as imposing object structure on raw bytes retrieved from a server.

To describe the prototype, we start by presenting the specific interface that we designed—the C routines available to Mneme store clients and the functionality provided by those routines. We then describe the internal design of the store, starting with limitations purposely imposed for this prototype, and continuing with a discussion of the internal concepts, major data structures, algorithms, and internal interfaces. We close with some brief preliminary performance results obtained using the prototype.

3.1 The Client Interface

The interface is designed with several programming conventions in mind:

- Every routine returns a *result code*, chosen from a standard list of codes, with negative codes indicating errors, zero indicating a normal success (when only a success-failure distinction is needed), and positive codes indicating additional successful conditions. Hence we will omit mentioning the results of routines unless we wish to point out particularly interesting cases.

- Any routine that must return data of unknown size allocates that data itself, setting a client-supplied pointer variable; it is the client's responsibility to free the storage when it is no longer needed. Fixed size result items are always allocated by the client, to allow efficient static or stack allocation. The client passes the address of these items to the Mneme routines as required.

- While C does not support data abstraction, several types are considered abstract in the Mneme interface. That is, client code should not make assumptions about how those types are represented. These types include FILEID, POOLID, STRATID, STRAT_EVENT, ID, SLOT, HANDLE, and FILENAME.

In the presentation below we will label the routine arguments to make their role more clear, using IN to indicate a read-only value, INOUT to indicate a variable that may be read and/or updated, and OUT to indicate a variable that is written but not read (i.e., will contain a result if the routine completes successfully). In the INOUT and OUT cases C requires that an *address* be passed, so an integer variable appears as int *, etc. For those routines that return informational values in their result codes, we use the notation = to indicate the kind of information provided. We omit descriptions of a few non-essential routines in the interest of brevity.

Object and Handle Operations

There are a few operations related to objects that use object ids, but most use handles. We first consider the ones oriented towards ids.

```
MnObjectCreate   (IN FILEID f, POOLID p,
                  int numSlots, int numBytes, int attrs, ID nearId,
                  OUT HANDLE *handle, ID *id)
MnObjectDestroy  (IN ID id)
MnObjectExists   (IN ID id) = boolean
MnObjectCompare  (IN ID id1, ID id2) = boolean
```

MnObjectCreate creates a new object, returning both the id and a handle on the new object. It turned out to be convenient to require that the file as well as the pool be specified, to eliminate the need for a global pool table. One also specifies the desired number of slots and bytes, and the initial value for the attributes. The nearId argument gives a location hint, suggesting that the new object be allocated "close to" the object named by nearId. The pool policy routine can take this hint into account in determining where physically to place the new object. nearId may be the null object id, indicating no hint. One may also pass a null pointer for the address of either the handle or the id to be returned, indicating that the corresponding item is not to be returned.

MnObjectDestroy allows explicit destruction of objects. If an object is destroyed and a later attempt is made to obtain a handle on the object, an error code is returned, provided that the id has not yet been "recycled" and used for another object. Thus, it is the client's responsibility not to use any dangling references. This decision allows efficient, direct, object reclamation in those cases where it is safe. This feature might be used by a garbage collector. Since objects can be destroyed, we provide MnObjectExists to check if the object corresponding to a given id still exists. Because forwarders allow objects to be aliased (to have more than one distinct name), we provide MnObjectCompare for checking if two objects are the same after following any forwarders.

Here are the interfaces to the handle oriented object routines:

```
MnObjectFile       (IN HANDLE h, OUT FILEID *f)
MnObjectPool       (IN HANDLE h, OUT POOLID *p)
MnObjectNumSlots   (IN HANDLE h, OUT int *ns)
MnObjectNumBytes   (IN HANDLE h, OUT int *nb)
MnObjectVolatile   (IN HANDLE h) = boolean
MnObjectModified   (IN HANDLE h) = boolean

MnObjectGetSlots   (IN HANDLE h, int first, int count, SLOT *slotPtr)
MnObjectGetSlot    (IN HANDLE h, int which, OUT SLOT *s)
MnObjectSetSlot    (IN HANDLE h, int which, SLOT s)
MnObjectSetSlots   (IN HANDLE h, int first, int count, SLOT *slotPtr)
MnObjectGetBytes   (IN HANDLE h, int first, int count, BYTE *bytePtr)
MnObjectSetBytes   (IN HANDLE h, int first, int count, BYTE *bytePtr)
MnObjectAttrs      (IN HANDLE h, int andMask, int xorMask, OUT int *attr)
```

The first group of routines are simple inquiries. The second group allows access to the three parts of an object. There are operations to get and set single slots as well as a series of slots, because we felt that might commonly be done. The byte operations always operate on a series of bytes (but one can always specify a count of 1). Note that the operations on multiple bytes or slots take a pointer to a client specified buffer area.

The MnObjectAttrs operation needs a little further comment. If the current value of the attributes of the object is v, the result (which is also stored in the object as the new value for its attributes) is[8] $(v \wedge \text{andMask}) \oplus \text{xorMask}$. This formulation allows any of the four boolean operations on a single bit value (set, clear, invert, nothing) to be performed individually to each attribute bit, all at once, requiring only one attribute manipulation routine.

Recall from the design that a slot can be empty, contain an id, or contain immediate data. The three operations below allow these cases to be distinguished. Since a slot has a known fixed size, there is no use (in C anyway) in providing coercion operations between slots and the three kinds of items they contain. We did provide C macros that reduce to type casts (not illustrated here).

```
MnSlotIsEmpty (IN SLOT s) = boolean
MnSlotIsData  (IN SLOT s) = boolean
MnSlotIsId    (IN SLOT s) = boolean
```

Finally, there are three operations on handles themselves: to create them, destroy them, and obtain the id of the object to which they are bound. Note that destroying a handle does not affect the *object* to which the handle is bound, except perhaps to allow other access in the case of volatile objects. .

```
MnHandleCreate  (IN ID id, OUT HANDLE *h)
MnHandleDestroy (INOUT HANDLE *h)
MnHandleId      (IN HANDLE h, OUT ID *id)
```

Pool and Strategy Operations

Recall that pools (and files) have associated *attributes*, allowing them to have arbitrary client-specified (and strategy-specific) parameters. In the initial prototype, these attributes are realized as name-value pairs where both the name and the value are strings. One can get, set, and remove individual attributes, and get all the attributes of a pool at once. Here we used STRING for the representation of an attribute, which is actually char * in the prototype. MnPoolGetAllAttrs allocates and returns two parallel arrays, one for the attribute names and one for the values; count indicates the number of elements in these arrays.

[8]$\oplus$ is the exclusive-or operator.

```
MnPoolCreate       (IN FILEID f, STRATID s, OUT POOLID *p)
MnPoolDestroy      (IN FILEID f, POOLID p)
MnPoolGetAllAttrs  (IN POOLID p, FILEID f,
                    OUT STRING (*attrs)[], STRING (*vals)[], int *count)
MnPoolGetAttr      (IN POOLID p, FILEID f, STRING attr, OUT STRING *val)
MnPoolSetAttr      (IN POOLID p, FILEID f, STRING attr, STRING val)
MnPoolRemAttr      (IN POOLID p, FILEID f, STRING attr)
```

There is only one routine related to strategies; it allows the client to indicate the routine to be executed when a particular *strategy event* occurs for a given strategy. A strategy event is a particular occurrence, such as creation of an object. The set of events is not considered part of the Mneme client interface *per se*; it is part of the strategy interface, which we will discuss later.

```
MnStrategySetRoutine (IN STRATID s, STRAT_EVENT e, int (*routine()))
```

File and General Operations

To support the extension of options related to file handling, such as the "mode" in which to open a file, the prototype provides a session-wide collection of *options*. These are name-value pairs, where the name and value are both strings. While options are similar to the attributes attached to pools (and files), options are not associated with data structures, but with the current session. They are similar to Unix environment variables. Each file operation (or, for that matter, any server or policy routine) can examine options relevant to it. Here are the various routines, analogous to the pool attribute routines:

```
MnOptionGet     (IN STRING option, OUT STRING *value)
MnOptionSet     (IN STRING option, STRING value)
MnOptionRem     (IN STRING option)
MnOptionGetAll  (OUT STRING (*options)[], STRING (*values)[], int *count)
MnOptionSetAll  (IN STRING (*options)[], STRING (*values)[], int count)
```

The basic file manipulation operations, and the file attribute routines, are also straightforward. Macros are provided (not illustrated here) to set up and examine variables of type FILENAME.

```
MnFileExists     (IN FILENAME fn) = boolean
MnFileCreate     (IN FILENAME fn, OUT FILEID *f)
MnFileDestroy    (IN FILENAME fn)
MnFileRename     (IN FILENAME oldname, FILENAME newname)
MnFileOpen       (IN FILENAME fn, OUT FILEID *f)
MnFileClose      (INOUT FILEID *f)
```

```
MnFileGetAllAttrs (IN FILEID f,
                   OUT STRING (*attrs)[], STRING (*vals)[], int *count)
MnFileGetAttr     (IN FILEID f, STRING attr, OUT STRING *val)
MnFileSetAttr     (IN FILEID f, STRING attr, STRING val)
MnFileRemAttr     (IN FILEID f, STRING attr)
```

A number of additional file inquiries are provided, whose use should also be fairly obvious. MnFileGetRoot returns a handle and/or an id, similarly to MnCreateHandle. MnFileSetRoot allows the root object to be set from a handle, or, if the handle pointer is null, an object id.

```
MnFileGetRoot  (IN FILEID f, OUT HANDLE *h, ID *id)
MnFileSetRoot  (IN FILEID f, HANDLE *h, ID id)
MnFileName     (IN FILEID f, OUT FILENAME *fn)
MnFileId       (IN FILENAME fn, OUT FILEID *fid)
MnFileGetPools (IN FILEID f, OUT POOLID (*pids)[], int *count)
```

Finally there are a two global operations that manipulate all the open files, and the initialization routine (called once only, at the beginning of a Mneme session).

```
MnFileGetAllOpen (OUT FILEID (*fids)[], int *count)
MnFileCloseAll   ()
MnInit           ()
```

Transaction Operations

The collection of routines provided is simple and straightforward. In addition to begin, commit, and abort routines, there are routines to check if the session is currently in a transaction (MnTxnExists; note that no object operations are allowed outside of transactions), to check if the current transaction appears to be committable (MnTxnOk; may involve interaction with a server), and to commit the current transaction and immediately start a new one (MnTxnAgain; provided to give a hint that objects manipulated by the committing transaction are likely to be used by the new transaction as well).

```
MnTxnBegin  ()
MnTxnCommit ()
MnTxnAbort  ()
MnTxnAgain  ()
MnTxnExists () = boolean
MnTxnOk     ()
```

To give some support for associating session specific temporary information with Mneme objects (e.g., heap versions of objects for direct use by a programming language), Mneme supports the association of an integer with an object id. These associations are flushed at every transaction commit (since the objects may be acquired by other users and the associated information such be reconstructed). Since efficient language integration demands that the programming language be more tightly integrated with Mneme (i.e., the language run-time system should usually access Mneme objects in the Mneme buffers rather than making a copy of them), these routines are likely to be reworked or dropped.

```
MnTxnGetInfo (IN ID id, HANDLE *h, OUT int *info)
MnTxnSetInfo (IN ID id, HANDLE *h, int info)
```

Profiling Operations

The interface includes several operations for acquiring statistics and counts. Two data types are provided: FILEDATA is a structure giving various size aspects of a Mneme file, and PROFDATA is a structure containing counts of various operations and times. These are not client extensible, but are designed to be easily extended in the future. For each structure operations are provided to print the structure on the standard output (whatever the C printf library routine does). The Copy routines sample file or profile data into a client-supplied structure, and the Reset routine resets the profiling counters and timers.

```
MnFileInfoPrint (IN FILEID fid)
MnFileInfoCopy (IN FILEID fid, OUT FILEDATA *fd)

MnProfileReset  ()
MnProfilePrint  ()
MnProfileCopy   (OUT PROFDATA *pd)
```

3.2 Implementation

As previously mentioned, the initial prototype does not implement the complete client interface. We first discuss the limitations intentionally imposed on the initial implementation, and then consider relevant aspects of the detailed design of the prototype (concepts, data structures, algorithms, and internal interfaces).

Limitations of the Implementation

We were most concerned with the performance of the Mneme id and object manipulation routines. Hence, in the first prototype we ignored transaction issues, and a number of more advanced features not relevant to basic retrieval and access performance. Specifically, options, pool and file attributes, multiple files, and cross-file references were not implemented. We also imposed a limit on the total number of objects within a file to 1024 × 1024. This limit simplified the implementation of some internal data structures.

Design Concepts: Logical Segments

A *logical segment* in the Mneme internal design is a set of object ids that have the same high-order bits and vary only in the low order bits. In the initial prototype the logical segment size is 1024 object ids. Thus, any two object ids that differ only within their 10 least significant bits are in the same logical segment. A logical segment of ids is the analog (in object id space) of a virtual memory page (in a virtual address space). We require that objects in the same logical segment be located "in the same place"; what this means will be made more clear in a moment. The key feature of logical segments is that they allow us to locate objects using a table or index that is much smaller than if we allowed each object to be placed independently. In the initial prototype, the logical segment map is 1024 times smaller than a per-object map would be. This smaller size will allow the map to fit in main memory when a per-object map would not, and thus the logical segment grouping allows an object fault to be satisfied with one disk retrieval. We also require that objects in the same logical segment be co-resident in main memory, so logical segments are a unit of clustering, too.

Given the various sizes mentioned, we can now indicate the format of an object id as it is considered by Mneme; see Figure 1. The file number part was reserved for implementing multiple files. In the prototype type that field is set to zero.

F = File # (10 bits)	L = Logical Segment # (10 bits)	O = Object # (10 bits)

Figure 1: Mneme Object Id Format

As stored within a file, the file number part of the id is always zero. When an id is fetched from a slot, the file number is filled in with the number of the file containing the object from which the id is being fetched. This number is simply the index of the file in a (per session) open file table. When storing ids into slots, the file number part is zeroed (cross file references are not implemented).

Each file has a *logical segment table*. The permanently stored version indicates (indirectly) where each logical segment is on disk. When the file is open, the session maintains a map of the resident logical segments. Thus, finding an object from its id proceeds by indexing the session's file table with the id's file number, and then the file's logical segment table with the id's logical segment number. A logical segment begins with an array of pointers to its objects (a kind of object table), so the last step in locating a resident object is to index that array with the object number from the id. This *two-level direct map* scheme for locating objects is illustrated in Figure 2. We contrast it with other schemes later. Of course, the lookup procedure can fail if the logical segment is not resident, in which case the logical segment's disk location is known, the logical segment is retrieved, the tables updated, and finally the object is available as if it had been resident all along.

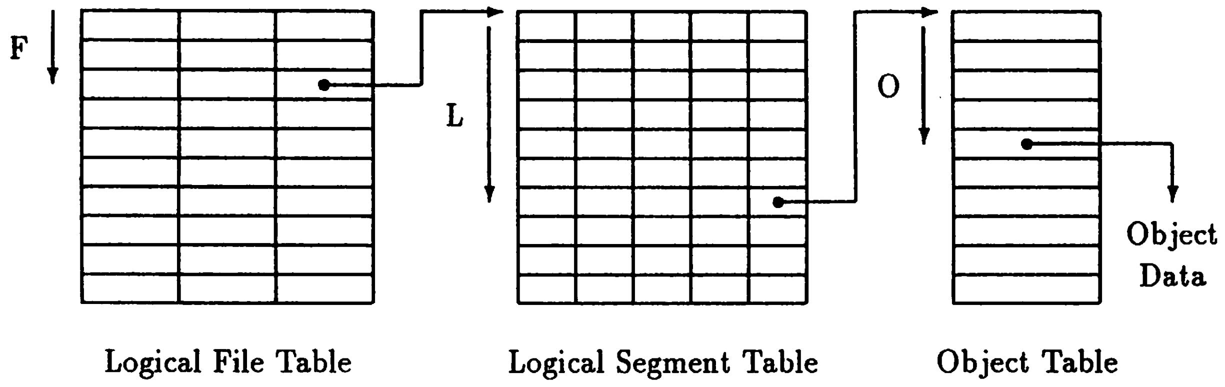


Figure 2: Two Level Direct Map Object Lookup Scheme

Design Concepts: Physical Segments

A Mneme *physical segment* is a byte string that contains one or more logical segments of objects. Physical segments are the items of discourse with servers, and it is generally assumed that a physical segment is stored contiguously, allowing efficient retrieval and storage of the segment as a unit. Our physical segments are analogous to the pages of database systems, except that physical segments do not have any fixed size. Reasonable sizes for physical segments probably range from a substantial fraction of a disk track up to a disk cylinder.

When an object fault occurs, we always retrieve a complete physical segment (a policy routine could request additional segments). Thus the physical segment is the true unit of clustering in Mneme. A file's logical segment map merely indicates for each logical segment the physical segment containing it. There is also a physical segment map that gives the server id for each physical segment. This id is presented to the server interface to fetch or store bytes of the segment. In the simplest server implementation, a server id is merely the offset of the start of the segment within the operating system file that implements the Mneme file containing the segment. Within Mneme, physical segments are identified by their index within their file's physical segment table. The internal version of the physical segment table includes the address of the physical segment's data when the physical segment is resident.

Objects are associated with Mneme pools via the physical (and hence logical) segments that contain the objects. What this means is that each physical segment belongs to exactly one pool, and that a pool thus manages a collection of physical and logical segments. This makes considerable sense since a pool needs unambiguous control over the placement and retrieval of its objects, and the management of the physical resources used to implement the objects.

In sum, a physical segment implements a set of logical segments along with their objects, providing a unit of clustering, transfer, update, and interchange with back end servers. In contrast, a logical segment is a group of object ids (and by implication a group of objects) that are always located, stored, and retrieved as a unit. A logical segment can be viewed abstractly as a piece of address space (actually, object id space), or concretely as a piece of an object table (which is never assembled as a single contiguous table).

Data Structures

While our discussion of the logical and physical segment concepts has introduced important parts of the Mneme prototype data structures, it is helpful to review the entire data structure now that those concepts have been introduced. We will distinguish between the *session* structures or forms and the *disk* structures or forms. The session structures are built and used in main memory during a session. They access and update the disk structures, which persist from session to session.

File tables. The *logical file table* (LFT) is a session structure. Mneme FILEID values are integers that index the LFT. When a file is opened, its *file header* is copied into main memory, and a field of the LFT entry for the file refers to the file header. The file header gives the physical segment ids of the file's physical and logical segment tables, etc. These tables are retrieved from the server and copies kept in main memory as long as the file is open. For simplicity, the in memory versions of the tables are always allocated at the maximum size, though the disk resident version uses only as much space as necessary for the used entries. The file header also indicates the file's root object, gives the heads of various free lists, and contains a variety of counters (e.g., how many objects are stored in the file).

Physical segment tables. There are in fact two segment tables associated with each Mneme file in the prototype. The *system physical segment table* (SPST) contains entries for the physical segments that contain internal table information (including the physical segment tables themselves). These physical segments are exceptional in that they do not contain logical segments or objects, but only specialized system data. This data is implemented in terms of physical segments in order to maintain a uniform interface to the servers and to allow the tables to change in size more conveniently (physical segments may be re-allocated to accommodate size changes). The *user physical segment table* (UPST) contains entries for the physical segments that contain client objects. The reasons for separating the physical segment table into two tables are obscure and uninteresting.

The entries in the SPST and UPST are rather large in the prototype. They contain the following information: the server id for the physical segment; the size of the segment, its block size, number of blocks, and number of new blocks that have been added to the segment since it was read in; the number of objects (actually, number of object table entries among the logical segments) in the segment; several free pointers for managing storage in the segment; the pool to which the segment belongs, that pool's strategy, and a field used to link together the segments of the pool; *referenced* and *modified* flags; and a pointer to the data of the segment (if the data is resident).

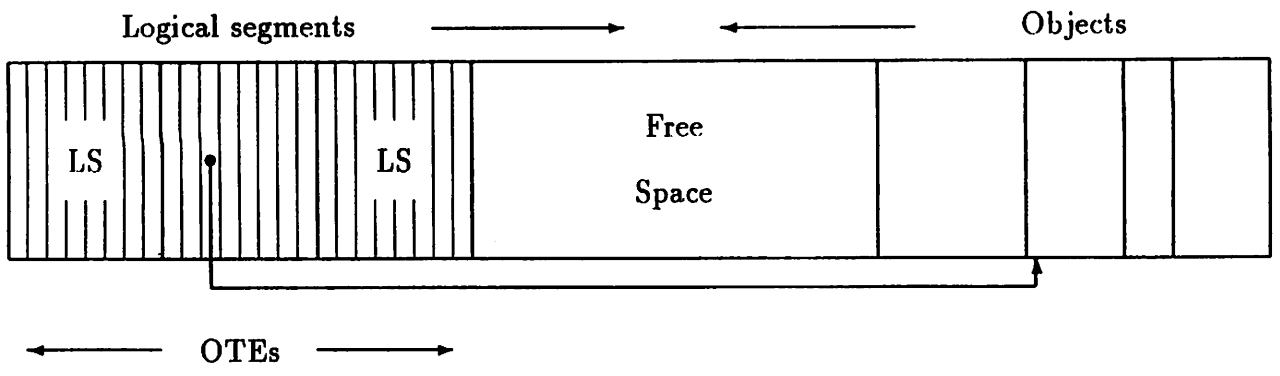
Logical segment tables. There is a logical segment table (LST) for each file, and its entries contain: the index (in the UPST) of the physical segment that contains the logical segment, the index of the logical segment within the physical segment (recall that a physical segment may contain many logical segments); a pointer to the object table entries of the segment (if the segment is resident); and a pointer to the TxnInfo data for objects in this logical segment (if there is any TxnInfo).

Pool allocation tables. Every file has a pool allocation table (PAT) whose purpose is to record information about the pools in the file. A Mneme POOLID is simply an index into the PAT. The entries in the PAT indicate the pool's strategy, the physical segment containing the pool attributes (if any; this is currently not implemented), and the head of the list of physical segments of the pool. The PAT is located on disk via the file header.

Physical segments. A physical segment is laid out with the object table parts of all of its logical segments at the beginning. These grow upwards as logical segments are added to the physical segment. The objects are allocated from the high end of the segment, with the object region growing downwards toward the object table region. An *object table entry* (OTE) is mostly a self-relative pointer to the corresponding object. The OTE also contains *referenced*, *modified*, and *deleted* flags. The physical segment's PSTE (PST entry) contains the various storage management pointers for memory allocation within the segment. Free OTEs are chained through the OTEs themselves, with the head being in the PSTE. The physical segment structure is illustrated in Figure 3.

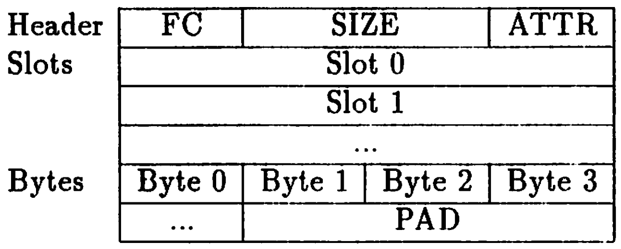
Object formats. Objects themselves consist of a 32 bit *object header*, followed immediately by the slots (if any), and then the bytes (if any), and padded to the next 32 bit boundary. The object and header formats are shown in Figure 4.

The interpretation of the size field depends on the format code. This allows the best use



LS	=	Logical Segment
OTE	=	Object Table Entry

Figure 3: Physical Segment Data Structure



FC	=	Format code, 4 bits
SIZE	=	Size information, 20 bits
ATTR	=	Attributes, 8 bits
PAD	=	Padding to word boundary

Figure 4: Mneme Object Format

of the size bits in a range of circumstances. Five formats are implemented, as described in Figure 5.

Format 3 allows large objects by using two additional words for the size information. Format 4 is a special case; it allows a language to use one slot for tagging objects with their class/type and still use the short header form for objects that otherwise contain only bytes. The set of formats can be extended in the future as necessary.

Handles. Handles are straightforward. They cache various information about an object and speed access to it. The fields of a handle are: a pointer to the object header, a pointer to the object's OTE, a pointer to the object's first slot, a pointer to the object's first byte, the object's id and pool, and the number of slots and bytes in the object. As previously noted, handles obviate locating an object and decoding its header on each use.

Tag	# of Slots	# of Bytes
0	10 bits from size	10 bits from size
1	20 bits from size	none
2	none	20 bits from size
3	32 bits from additional word	32 bits from additional word
4	1 (fixed)	20 bits from size

Figure 5: Mneme Object Formats

Strategy table. The strategy table is simply a two dimensional array indexed by strategy number and strategy event. Initially, only the default policy is entered; all other entries are zero.

Algorithms

Given the data structures, the algorithms are fairly obvious. One interesting case, perhaps, is the writing back of data into the store. Since transactions are not implemented, this currently happens when a file is closed. Starting from the file's LFT entry, we go to the PAT and iterate through the file's pools. For each pool, we write back its physical segments that have been modified. These are found by following the link field in each PSTE and by checking the *modified* flag in the PSTEs. When a segment is to be written back, if it has grown, a new server segment is allocated and the old one deallocated. After all the pools have been processed, the system segments are updated as necessary and the server is directed to close the file.

Internal Interfaces: Strategy Routines

The currently defined strategy events are: GetRoot (used when retrieving the root object of a file), NewObjectFormat (used to determine the format of a new object), NewObject-Segment (used to determine the placement of a new object), and GrowSegment (used to determine the new size of a segment that needs to grow). This aspect of the design clearly needs more work. For example, there should be a strategy routine related to MnHandleCreate and object faulting, and NewObjectFormat does not appear to be necessary at this point.

Internal Interfaces: Server

The server interface is much better developed. First, there are a number of file handling routines, mostly used by their Mneme counterparts. (The server interface follows conventions similar to those of the Mneme client interface, including returning result codes, etc.) The type FID is the server's id for a file (e.g., the number of the operating system channel to the file, or the address of an internal file information block).

```
SvFileExists  (IN STRING fileName)
SvFileCreate  (IN STRING fileName)
SvFileDestroy (IN STRING fileName)
SvFileRename  (IN STRING oldName, STRING newName)
SvFileOpen    (IN STRING fileName, OUT FID *fid)
SvFileClose   (INOUT FID *fid)
```

There are two routines to fetch and store the Mneme file header of a file, which (as far as the server is concerned) is a 512 byte block stored at some conventional place in the file (the place is chosen by the server implementation).

```
SvFileGetHdr (IN FID fid, INOUT char *buffer)
SvFilePutHdr (IN FID fid, IN char *buffer)
```

There are five routines for manipulating (physical) segments, which are identified by their server ids (type SVRID). The get and put routines are actually more general than previously indicated. They can access an arbitrary range of bytes within the segment, given by a length and a starting offset. This extra capability is currently unused.

```
SvSegCreate (IN FID fid, int len, char *buffer, OUT SVRID *sid)
SvSegDelete (IN FID fid, SVRID sid)
SvSegGet    (IN FID fid, SVRID sid, INOUT char *buffer, IN int len, int start)
SvSegPut    (IN FID fid, SVRID sid, IN char *buffer, int len, int start)
SvSegGrow   (IN FID fid, SVRID sid, int increment)
```

There are also five (unimplemented) transaction routines and an initialization routine (not illustrated).

3.3 Preliminary Performance Results

We have thorough performance measurement and analysis studies underway, but report some preliminary raw results here. We have implemented a revised version of the simple object-oriented database benchmark reported in [Rubinstein *et al.*, 1987]. This revised version is described briefly in [Cattell, 1988], with a more complete presentation in preparation.[9] We will call this the RC (revised Cattell) benchmark.

[9]Private communication with Cattell.

Description of the RC Benchmark

The database consists of *part* objects and *connection* objects. Each part has a *type* (a string of up to 10 characters), an application chosen *part number* (32 bit integer), x and y coordinates (32 bits each), and a *date* (32 bits suffice). Each connection has a *type* (a string of up to 10 characters), a *length* (32 bit integer), and (directionally) connects two parts.

The *small* database contains 20,000 parts and 60,000 connections, with each part having exactly 3 outgoing connections to randomly chosen parts. Larger databases contain 200,000 and 2,000,000 parts. Because of the 1024K object and 1024 physical segment limits of the Mneme prototype, we could test only with the small database.

The benchmark measures are the performance of three operations:

- Lookup: Retrieve 1000 randomly chosen parts given their part numbers and call a null procedure passing the x and y coordinates of the part information.
- Traversal: For a randomly chosen part, follow its connections forward seven levels deep (from each part we will visit 3230 parts, with duplicates possible). A null procedure is to be called for each part traversed. *Reverse* traversal is also to be measured and compared with forward traversal to ensure comparable results.
- Insert: Add 100 parts with 3 connections from each part to randomly chosen other parts.

These tests are to be performed "cold" (with empty buffers) and "warm" (with buffers already filled from previous database operations). The buffer space is allowed to be large enough to contain the entire small database in main memory, given that the database size is not overly inflated. (Our implementation used approximately 4.6 megabytes.)

Benchmark Implementation

The basic part and connection information is straightforward, resulting in 28 bytes for parts (4 bytes each for part number, x, y, and date, 11 bytes for the null terminated *type* string, and 1 byte padding) and 16 bytes for connections (4 bytes for length, 11 for the *type* string, and 1 byte padding). We implemented the interconnections by chaining together each parts incoming connections, and (on a separate chain) its outgoing connections. This results in two slots in each part object for the chain heads and two slots in each connection object for the chain links. The total size of parts and connections, including the 4 byte object header, was 40 and 28 bytes, respectively. Thus, adding in 4 bytes of OTE for each object, 1024 parts consume 44K bytes and 1024 connections consume 32K bytes. The physical segment size used for parts was 40K[10] and 32K for connections.

[10]This should have been 44K and the tests will be re-run in the future with 44K.

Locating parts by part number was done using a B+ tree. The representation was relatively space efficient, using for each B+ tree node and leaf one Mneme object occupying a physical segment all by itself. The leaves hold approximately 1000 keys, so a two level tree suffices.

The tests were performed on a VaxStation 3500 running VMS. The system has 32 megabytes of main memory. We performed the test on a local RA70 disk (a 270 megabyte 5.25 inch hard disk) and on a remote RD54 disk (a 160 megabyte 5.25 inch hard disk) served by a VaxStation 2000. For rough comparison with other popular equipment, a VaxStation 3500 is generally rated at 2.5–3 VAX MIPS (comparable to middle to high end Sun 3 systems and about 3 times a VAX 11/780) and the VaxStation 2000 is rated at .8–.9 times the speed of a VAX 11/780.

Benchmark Results

The results obtained are presented in Table 1. The numbers given are seconds of elapsed time (*not* cpu time), since the interesting performance measure is responsiveness, not system load. The column labels use the following abbreviations: R = remote, L = local, C = cold, and W = warm. The local, warm case number are not available, but since both the warm cases do no I/O, there is little difference between them (both consist almost purely of cpu time).

Operation	R/C	L/C	R/W
Lookup	14.45	9.97	2.56
Traversal	32.04	12.67	1.14
Insert	15.43	9.77	3.70

Table 1: Raw Performance Numbers (seconds elapsed)

It is interesting to consider the number of objects manipulated in the given cases, and to calculate the number of objects per second. If we concern ourselves only with the "user" level objects (parts and connections), we obtain the results in Table 2. The lookup object count should be obvious. Insert creates 100 parts and 300 connections, and traversal manipulates 3280 parts and a connection leading to each except the first.

It is also useful to take into account the B-tree objects accessed. Given the high branching factor, lookup examines 2 B-tree nodes per part object. Insert actually inserts the parts into *two* B-trees (the second one is used for additional benchmarks not reported here), giving 4 B-tree nodes accessed per part node. The adjusted object totals and object access rates are shown in Table 3.

Operation	Objects	R/C	L/C	R/W
Lookup	1000	69	100	391
Traversal	6559	205	518	5754
Insert	400	26	41	108

Table 2: User level objects accessed (per second elapsed)

Operation	Objects	R/C	L/C	R/W
Lookup	3000	208	301	1172
Traversal	6559	205	518	5754
Insert	800	52	82	216

Table 3: Total objects accessed (per second elapsed)

Finally, we wish to consider how many references are made to *components* of objects. This involves some estimation, especially for the B-trees. In lookup, we perform about 5 comparisons in binary search at the top level, and about 10 comparisons at the leaf level, each involving one field access. In addition we access at least 2 fields of each B-tree node to determine sizes. We then access 2 fields of the part object. We should also add at least 2 field accesses per object touched to account for establishing access to the object. The total number of fields accessed per lookup is then approximately 27. Traversal accesses one field in each of 1093 objects, and 2 fields in each of 3279 connection objects, plus the estimated 2 field accesses per object touched. Insertion must fill in all fields of the parts and connections added, which we count as 7 and 4 respectively, plus a charge of about 2 accesses to set up the objects. We must also account for lookup and changes to the B-trees. Finding the insertion point (comparing with lookup) costs about 25 accesses. Insertion is expensive, though, since on average half of the 2000 fields in the leaf nodes must be moved. We estimate each insertion performs approximately 2000 field accesses. However, we should discount this since most of the field accesses are done via block transfers. We used 1000 field accesses per insertion as our best estimate. The results of this estimation exercise are shown in Table 4.

In examining the various tables, it is useful to consider that lookup and insertion do considerable work on just a few objects, whereas traversal does a small amount of work on a large number of objects. Thus, the lookup and insertion results are more reflective of field access cost (the "warm" column makes the most sense) and traversal is most representative of object access. It is also important to take into account that the B-tree package is not as carefully tuned as is Mneme itself. Still, it is safe to say that Mneme supports accessing

Operation	Fields	R/C	L/C	R/W
Lookup	27000	1.87	2.71	10.55
Traversal	21000	.66	1.66	18.42
Insert	100000	6.48	10.24	27.03

Table 4: Estimated fields accessed (thousands per second elapsed)

thousands of objects and fields per second.

We find these results very encouraging. The traversal performance is especially interesting because it concentrates on Mneme's object location and faulting mechanisms. Each object visited seems to have required not more than roughly 500 instructions in this benchmark, including the benchmark's code and all field accesses. Further, the object location code can be improved by a factor of more than 2 and possibly as much as 5 or 10 with more careful coding.

Caveats

We must stress that the results are preliminary and require further analysis and checking before they can be considered definitive. Another important point is that the benchmark was designed to test object oriented and engineering database performance. Mneme is an object store and does not have a data definition or data manipulation language. Hence, we chose the implementation for parts, connections, and their inter-connection "by hand". Finally, Mneme does not support concurrency control and crash recovery at this point, which are likely to affect performance at least a bit. On the more encouraging side, the results above *do* include writing thousands of bytes of changes back to disk, and we would likely lock on a physical segment basis, so concurrency control and recovery overhead should not be drastic. We also feel confident that the path length in MnHandleCreate and related operations can be substantially shortened with more careful coding. Finally, it is not clear how meaningful this particular benchmark is, since it tests mainly "micro" operations (small steps) rather than "macro" ones (complete applications tasks). Our goal here, though, is not to evaluate the benchmark for its predictive or comparative value, but only to use it to give some sense of Mneme's absolute performance.

4 Shortcomings of the Prototype

Considering our goals, the most significant shortcomings of the initial Mneme prototype are probably the lack of multi-user and multi-file facilities (concurrency control, recovery, transactions, cross file references, etc.). There are other problems and limitations that need to be addressed, though. We distinguish between problems with the conceptual design, problems with the client interface, and problems with the detailed design and implementation.

4.1 Problems with the Conceptual Design

The segregation of slots and bytes presents significant problems in achieving a tight integration of the Mneme store with a programming language such as Modula-3, which intermingles pointer and non-pointer fields within objects. We need to come up with an appropriate descriptor mechanism, but without sacrificing the compactness of the current object format. Likewise, handles are not appropriate for tight language integration. Rather, we need to export a direct pointer to object information in the buffers, with appropriate routines provided for converting pointers to/from Mneme's disk format. It would also appear that there are places where language specific "call backs" should be invoked, etc.

In a similar vein, the details of the slot tagging mechanism probably need to be reworked at the same time a descriptor mechanism is introduced. The principal reason is that different languages may do the tagging in different ways for their own reasons, or may not need tags at all. A suitably extensible descriptor mechanism would solve this problem, too. Another issue related to object formats is the use of attribute bits. This needs to be refined, with some bits reserved for Mneme use or having standard meanings, and other bits being applications or language specific. Finally, the current scheme makes no provision for heterogeneous hardware. A descriptor mechanism can help with that as well, since it can be used to locate and indicate integer, floating point, string, etc., formats and to support conversion as required for each hardware platform.

The conceptual design is not suited to the manipulation of large amounts of relatively unstructured data, such as digitized audio and video or images. These probably require a distinct kind of object accessed in a rather different way. For example, rather than faulting in an entire object, it may be appropriate to have a "window" into the object. We do believe we have a sound approach to small to medium size structured objects, though.

No provisions have been made for multi-threaded clients. It is not clear how this affects the conceptual design; the impact may be more on the client interface, which must somehow provide for multiple contexts and must handle concurrent invocation of Mneme operations, with internal serialization as necessary.

The whole area of naming and locating files is too vague. We need to define an interface to a name server, and implement a very simple one that can be replaced with better ones depending on what is available in the host environment. Similarly, the cross file reference mechanisms, and the incoming reference tables for files, need to be defined.

In implementing the B+ tree, and in contemplating the implementation of additional search structures layered on top of Mneme, we have noticed a need for more direct access to some internal structures. For example, there are times when it would be useful to be able to iterate through all physical or logical segments in a pool and through the objects in them. There is also a need to name physical segments when placing objects. In building search structures, there is also frequently a need to move objects (e.g., to split a B tree node). How to present the internal concepts appropriately is an interesting problem in the conceptual design.

4.2 Problems with the Client Interface

The uniform error reporting mechanism is nice, but it is inconvenient for programmers to check results of every call. We need some kind of error and exception reporting mechanism, such as registering a routine to be called on an error. This is mainly the result of expressing the interface in terms of C, a decision that, on balance, we still feel is correct. The lack of iteration facilities in C also gives rise to interfaces that are sometimes awkwardly expressed. We are considering replacing or augmenting the various GetAll operations with routines that will invoke a client-supplied routine for each item, and allow the iteration process to continue or to be stopped. Given the limitations of C, the interface is relatively sound.

4.3 Problems with the Implementation

The two level direct map is not very space efficient, and the limitation on the number of objects in a file should be removed. This will require substantial redesign of the internal tables. A promising approach to locating objects is to hash the upper bits of the object id and probe a hash table of resident logical segments. The more complicated logic of hashing will likely offset the reduction in levels of indexing, so there will be little difference in time performance, but space can be substantially reduced by keeping in main memory only the physical segment number for non-resident logical segments, rather than the bulky logical segment table entries of the current design. Similarly, we should be able to cut down on the main memory information about non-resident physical segments, since all we need to know are the segment's pool, its server id, and its size (for buffer management). Both the PST and LST should be compact for small files, but gracefully expandable for very large files. The translation of ids from their Mneme-internal to their client form will have to be more sophisticated, yet the cost of transforming ids must be kept as small as possible.

The implementation currently performs no "buffer management" in the traditional sense. Rather, it merely allocates virtual memory whenever a non-resident segment is retrieved. Buffer management needs to be integrated into the system architecture so that applications (presumably via pool policies) can control resource consumption. It is not clear whether the buffer manager should be thought of as a layer interposed between Mneme and the servers, thus acting as a segment cache, or if the buffer manager should be considered separate with pools making calls to both the buffer manager and the servers.

We need to implement more and better servers. For example, the current one does not manage free space within a file. We also need to explore garbage collection, and to consider providing more tools to examine and debug the contents of Mneme files.

5 The Next Prototype

Taking into consideration the problems described in the previous section, we are beginning work on the second Mneme prototype. It will provide an extensible descriptor mechanism, application/language specific tagging, etc. The internal tables will be reorganized as suggested above. The simple transaction mechanism will be implemented, and the transaction design reconsidered so that applications will be able to extend the transaction mechanism. Exception handling and iteration features will be added to the interface. Better support will be provided for implementers of search structures (access methods). We will also consider including support for at least linear sequences of versions of objects. We feel that all these features are needed by the current and potential users of Mneme. We may continue to omit features for multiple files and cross references, since a single large file will suffice for many potential users and the other shortcomings seem to have higher priority.

In addition to improving the Mneme store itself, we plan to implement a lightweight file system that provides more efficient support for Mneme than a traditional time-sharing oriented file system. This file system should run on a "raw disk", or within a single operating system file if one does not wish to dedicate a disk partition to Mneme. We will also continue to offer a server that implements each Mneme file within an operating system file.

6 Related Work

There is and has been much activity recently in the areas of database support for "new" applications (applications of the kind we wish to support), of persistent/database programming languages, and even object stores and database extensions of virtual memory. In the interests of brevity and incisiveness we relate Mneme to well known exemplars of the various approaches rather than attempting exhaustive enumeration of all related work.

6.1 Extensible Databases

Postgres [Stonebraker and Rowe, 1986], Exodus [Carey *et al.*, 1986; Richardson and Carey, 1987; Graefe and DeWitt, 1987], and Genesis [Batory *et al.*, 1988] take different approaches to extensibility. Postgres is built on the relational database model. While it has significant extensions, it does not support object identity or tight language integration. It is not a "lightweight" system. Rather, it attempts to provide a full featured database system. It does provide some "side doors" for extensibility and performance enhancement, but its semantic approach is still substantially different from ours.

Genesis takes more of a tool kit and building block approach for the construction of customized databases. These databases can offer significant performance benefits over general purpose database management systems by eliminating unneeded features and inserting carefully chosen application specific enhancements. The Genesis technology might possibly be used to build something like Mneme, though our hand coded implementation might tend to be better.

Exodus takes more of a language approach to extensibility. Its architecture includes the E language, an extension of C++. Of the extensible database systems, Exodus is the one most similar to Mneme, but the fairest comparison would be between Mneme and the Exodus storage manager (as opposed to other components of the Exodus system). Exodus storage manager objects are byte strings, there is no provision for garbage collection, and the design seems tuned to large rather than small objects.

Thus, most of the extensible database systems support objects weakly or not at all. They also happen to be oriented towards fixed size database pages. While the Exodus storage manager is more similar to Mneme in those respects, it, like the other systems, is designed for centralized rather than distributed use.

6.2 Object Oriented Databases

Some relevant object oriented database systems are Orion [Kim *et al.*, 1988], GemStone [Purdy *et al.*, 1987], and VBase [Andrews and Harris, 1987]. These systems are oriented towards specific languages (Lisp, Smalltalk, and C/C++, respectively). Orion and GemStone do support (single) servers, but none of the systems have Mneme's orientation towards a large distributed space of objects. Also, these are all attempts at complete database management systems, and end up being "heavy weight" compared with Mneme.

6.3 Persistent and Database Programming Languages

PS-Algol [Atkinson *et al.*, 1981; Atkinson and Morrison, 1985] is representative of the persistent programming languages, and E (already mentioned), O_2 [Lécluse *et al.*, 1988; Bancilhon *et al.*, 1988; Velez *et al.*, 1989], O++ [Agrawal and Gehani, 1989a; Agrawal and Gehani, 1989b], and Opal (the language in the Gemstone system) of recent database programming languages. These systems are all oriented towards particular programming languages and/or data models, whereas Mneme is attempting to provide a generic substrate (possibly suitable for building some of the other systems). Some of them, O_2 especially, have semantically richer data models, requiring more built-in features such as sets. None of these systems have Mneme's goal of use in a large distributed system.

6.4 Object Stores and Persistent Memories

Related work in this area could be stretched to include Multics [Organick, 1972] and a variety of capability architectures. More narrowly construed, some interesting examples are Camelot [Spector *et al.*, 1986], 801 [Chang and Mergen, 1988], FAD [Copeland *et al.*, 1988], ObServer [Skarra *et al.*, 1987; Hornick and Zdonik, 1987], and CACTIS [Hudson and King, 1986]. Camelot and 801 offer different version of persistent/recoverable virtual memory. While Camelot supports multiple servers in a distributed environment, it does not provide uniform naming throughout a system. It also provides no notion of "object", only a flat virtual (though resilient) memory. 801 is an architecture in which databases can be built in virtual memory, using the paging hardware for database style locking (at a physical level), etc. FAD has been implemented similarly, though it does include some limited object semantics. 801 and FAD are strictly centralized systems.

ObServer provides a network server offering objects that are byte-vectors, and a variety of locking and transaction modes. It supports only a single server and a single space of object ids, as opposed to Mneme's notion of a large collection of spaces of objects. The Mneme design has placed more emphasis on performance and on scaling to large distributed systems.

CACTIS also has a substantially different focus from Mneme. It supports objects and connections where the connections allow the value of a slot of an object to be derived from the value of a slot of another object. CACTIS is thus oriented towards "active" data, and how to do the derivation process most efficiently when slots are updated. Mneme might provide an interesting substrate on which to build CACTIS or other derived/active data semantics.

6.5 Summary of Related Work

Mneme's goals are unique among the systems discussed. Goals that tend to distinguish Mneme from other systems are:

- supporting multiple programming languages
- providing (storage) objects with structure and identity, but "no frills"
- emphasizing performance
- addressing issues of autonomy and large distributed systems
- offering policy extensibility

The current prototype of Mneme is not distributed, but it does meet the other goals, to wit: Mneme has been used with both C and Ada; earlier discussion of the prototype explained Mneme's object concept; the design and initial benchmark results indicate our emphasis on performance; and we have already used the policy extensibility to tune the cluster sizes for the B-tree implementation. The goals met still distinguish Mneme from other systems. Further, the current work will extend readily to distributed systems, since we designed for distribution from the start.

7 Conclusions

Our approach to providing object semantics and high performance for CAD and related information intensive applications is to build a simple, "light weight" persistent object store, as opposed to a full blown database management system. Referring back to the goals stated in the introduction, the design and initial prototype substantially meet the following: object structure semantics, high performance, portability, and modularity. The Mneme object store design appears quite promising, as evidenced by the initial prototype. We have demonstrated that the object location, faulting, and manipulation code can support thousands of object accesses per second, and the next prototype promises noticeable improvement over this performance. The principal achievement is the combination of object functionality (identity and structure) and performance.

The conceptual design tradeoffs are interesting in that Mneme's performance, portability, and genericity come about as much from what is *not* provided as from what is. Applications may indeed require additional features and semantics, but Mneme supports exploration and wide range of tasks through not supplying too much semantics, and provides an efficient and useful base on which to build applications rather than starting from scratch every time or living with the inferior functionality and performance of available bases.

In sum, the Mneme project is developing a unique blend of programming language, database, and distributed system functionality, that will provide good support for many emerging applications. Mneme's portability and comparative simplicity will make it attractive to researchers and others interested primarily in the applications above, rather than the bytes below. The system has already been welcomed by one software engineering research group and is being sought by others. The experience of these groups will further confirm Mneme's utility.

In addition to building a second prototype and doing more complete performance studies of Mneme, we are engaged in building Persistent Smalltalk and Persistent Modula-3 on top of Mneme, to come to a better understanding of the performance aspects of persistent programming languages based on an object faulting approach. We are also engaged in work on transaction models supporting cooperation, and distributed execution models that overcome the limitations of distributed systems based only on a shared store.

8 Acknowledgments

Steven Sinofsky wrote the first Mneme prototype. Shenze Chen worked on the server component. Charudatt implemented the RC benchmark. All their efforts were essential in this work. Critique from the faculty and students of the Software Development Laboratory at UMass, especially Lori Clarke, Jack Wileden, and Peri Tarr has been especially helpful. Tarr has been largely responsible for implementing some Software Development Laboratory tools on top of Mneme, which has given us valuable feedback on the concepts, interface, design, and performance of the first Mneme prototype.

References

[Agrawal and Gehani, 1989a] R. Agrawal and N. H. Gehani. ODE (Object Database and Environment): The language and the data model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Portland, OR, June 1989), ACM SIGMOD Record, ACM, pp. 36–45. Published as ACM SIGMOD Record Volume 18, Number 2.

[Agrawal and Gehani, 1989b] R. Agrawal and N. H. Gehani. Rationale for the design of persistence and query processing facilities in the database language O++. In *Proceedings of the Second International Workshop on Database Programming Languages* (Oregon Coast, June 1989).

[Andrews and Harris, 1987] Timothy Andrews and Craig Harris. Combining language and database advances in an object-oriented developement environment. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, Oct. 1987), ACM, pp. 430–440.

[Atkinson and Morrison, 1985] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *ACM Trans. Program. Lang. Syst.* *7*, 4 (Oct. 1985), 539–559.

[Atkinson *et al.*, 1981] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a peristent heap. *ACM SIGPLAN Notices 17*, 7 (July 1981).

[Bancilhon *et al.*, 1988] Francois Bancilhon, Gilles Barbedette, Véronique Benzaken, Claude Delobel, Sophie Gamerman, Cristophe Lécluse, Patrick Pfeffer, Philippe Richard, and Fernando Velez. The design and implementation of O_2, and object-oriented database system. In *Advances in Object-Oriented Database Systems* (Sept. 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 1–22.

[Batory *et al.*, 1988] D. S. Batory, J. R. Barnett, J. F. Garza, K. P. Smith, K. Tsukuda, B. C. Twitchell, and T. E. Wise. GENESIS: an extensible database management system. *IEEE Trans. Softw. Eng.* (Nov. 1988), 1711–1730.

[Cardelli *et al.*, 1988] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, and Greg Nelson. Modula-3 report. Tech. Rep. ORC-1, DEC Systems Research Center/Olivetti Research Center, Palo Alto/Menlo Park, CA, 1988.

[Carey *et al.*, 1986] M. J. Carey, D. J. DeWitt, J. E. Richardson, and E. J. Shekita. Object and file management in the EXODUS extensible database system. In *Proceedings of the 12th International Conference on Very Large Databases* (Kyoto, Japan, Sept. 1986), ACM, pp. 91–100.

[Cattell, 1988] R. G. G. Cattell. Object-oriented DBMS performance measurement. In *Advances in Object-Oriented Database Systems* (Sept. 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 364–367.

[Chang and Mergen, 1988] Albert Chang and Mark F. Mergen. 801 storage: Architecture and programming. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 28–50.

[Copeland *et al.*, 1988] George Copeland, Mike Franklin, and Gerhard Weikum. Uniform object management. MCC Technical Report ACA-ST-411-88, Microelectronics and Computer Technology Corporation, Austin, TX, Dec. 1988.

[Gehringer, 1979] Edward F. Gehringer. Variable-length capabilities as a solution to the small-object problem. In *Proceedings of the Seventh ACM Symposium on Operating System Principles* (Pacific Grove, CA, Dec. 1979), ACM, pp. 131–142. Published as an issue of ACM Operating Systems Review.

[Gehringer, 1989] Edward F. Gehringer. Name-based mapping: Addressing support for persistent objects. In *Persistent Object Systems: Their Design, Implementation, and Use* (University of Newcastle, NSW, Australia, Jan. 1989), Department of Computer Science, pp. 139–157.

[Goldberg and Robson, 1983] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[Graefe and DeWitt, 1987] Goetz Graefe and David J. DeWitt. The EXODUS optimizer generator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, CA, May 1987), ACM, pp. 160–172.

[Hornick and Zdonik, 1987] Mark F. Hornick and Stanley B. Zdonik. A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 70–95.

[Hudson and King, 1986] S. Hudson and R. King. CACTIS: A database system for specifying functionally-defined data. In *Proceedings of the Workshop on Object-Oriented Databases* (Pacific Grove, CA, Sept. 1986), ACM, pp. 26–37.

[Intel Corporation, 1981] Intel Corporation. *Introduction to the iAPX 432 Architecture, Manual 171821-001.* Intel Corporation, Santa Clara, CA, 1981.

[Kernighan and Ritchie, 1978] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language.* Prentice-Hall, 1978.

[Khoshafian and Copeland, 1986] Setrag N. Khoshafian and George P. Copeland. Object identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 406–416.

[Kim *et al.*, 1988] Won Kim, Nat Ballou, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Integrating an object-oriented programming system with a database system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Diego, California, Nov. 1988), ACM SIGPLAN Notices, ACM, pp. 142–152. Published as ACM SIGPLAN Notices Volume 23, Number 11.

[Lécluse *et al.*, 1988] Christophe Lécluse, Philippe Richard, and Fernando Velez. o_2, an object-oriented data model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Chicago, Illinois, Sept. 1988), ACM SIGMOD Record, ACM, pp. 424–433. Published as ACM SIGMOD Record Volume 17, Number 3.

[Liskov *et al.*, 1977] B. Liskov, A. Snyder, R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564–576.

[Liskov *et al.*, 1981] B. Liskov, R. Atkinson, T. Bloom, E. Moss, C. Schaffert, R. Scheifler, and A. Snyder. *CLU Reference Manual.* Springer-Verlag, 1981.

[Moss, 1989] J. Eliot B. Moss. Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Second International Workshop on Database Programming Languages* (Gleneden Beach, OR, June 1989), pp. 269–285. In press; also available as University of Masschusetts, Department of Computer and Information Science Technical Report 89–68.

[Organick, 1972] Elliott I. Organick. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, MA, 1972.

[Organick, 1983] Elliott I. Organick. *A Programmer's View of the Intel 432.* McGraw-Hill, New York, 1983.

[Purdy *et al.*, 1987] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 27–47.

[Richardson and Carey, 1987] Joel E. Richardson and Michael J. Carey. Programming constructs for database system implementations in EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, CA, May 1987), ACM, pp. 208–219.

[Rubinstein *et al.*, 1987] W. R. Rubinstein, M. S. Kubicar, and R. G. G. Cattell. Benchmarking simple database operations. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, CA, May 1987), ACM, pp. 387–394.

[Schaffert *et al.*, 1986] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian, and Carrie Wilpolt. An introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), vol. 21(11) of *ACM SIGPLAN Notices*, ACM, pp. 9–16.

[Skarra *et al.*, 1987] Andrea Skarra, Stanley B. Zdonik, and Stephen P. Reiss. An object server for an object oriented database system. In *Proceedings of International Workshop on Object-Oriented Database Systems* (Pacific Grove, CA, Sept. 1987), ACM, pp. 196–204.

[Spector *et al.*, 1986] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, and Dean S. Thompson. The Camelot project. Tech. Rep. CMU-CS-86-166, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1986.

[Stonebraker and Rowe, 1986] M. Stonebraker and L. A. Rowe. The design of Postgres. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 1986), ACM, pp. 340–355.

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

[Velez *et al.*, 1989] Fernando Velez, Guy Bernard, and Vineeta Darnis. The O_2 object manager, an overview. In *Second International Workshop on Database Programming Languages* (Gleneden Beach, OR, June 1989), pp. 331–350. In press.