Design of the Mneme Persistent Object Store

J. ELIOT B. MOSS
University of Massachusetts, Amherst

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. The project strategy is to implement efficient, distributed, persistent programming languages. We report here on the Mneme persistent object store, a fundamental component of the project, discussing its design and initial prototype. Mneme stores *objects* in a simple and general format, preserving object identity and object interrelationships. Specific goals for the store include portability, extensibility (especially with respect to object management policies), and performance. The model of memory that the store aims at is a single, cooperatively-shared heap, distributed across a collection of networked computers. The initial prototype is intended mainly to explore performance issues and to support object-oriented persistent programming languages. We include performance measurements from the prototype as well as more qualitative results.

1. INTRODUCTION

The Mneme[1] 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. The

[1] Mneme is the Greek word for *memory*; we pronounce it NEE-mee.

project strategy is to implement efficient, distributed, persistent programming languages. We report here on the Mneme persistent object store, a fundamental component of the project. We discuss, in turn, our goals for the long-term Mneme store effort; the conceptual design of the store; a working prototype of a subset of the full design; related work; conclusions and future directions.

1.1 Goals for the Store

The Mneme project's overall goal is to provide better support for cooperative, information-intensive tasks, including computer-aided design (CAD)—ranging from VLSI design through electrical, mechanical, and architectural design—as well as computer-aided software engineering (CASE). The tasks also include document preparation, publishing, and office automation applications, as well as hypertext and other advanced information systems and tools to support group work. We use *cooperation* to imply concurrent, distributed work and a need for nonstandard (i.e., not necessarily serializable) concurrency control and integrity management, and use *information intensive* to imply a need for reliable and efficient storage, retrieval, and manipulation of potentially long-lived data having considerable internal structure.

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 a good foundation; the store project is an attempt to produce prototypes that allow the hypothesis to be tested, by us 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 *object*, such that identity of, and relationships among, objects are preserved.

—The store should incur very low overhead to manipulate resident objects and have high performance for retrieval and storage.

—The store software 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 operate in a distributed, heterogeneous, client/server environment, with as much transparency as possible, while substantially respecting autonomy of resources.

—The store should support demarcated, somewhat independent, subcollections 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.

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 commonly associated with 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 contiguous block of fields with an associated 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, rather than, for example, automatically managed binary or higher arity relationships. More sophisticated data models can be built in terms of the simple Mneme objects. This keeps Mneme lightweight and general, and somewhat "low level," compared with database management systems.

Performance. One of the motivations of the overall Mneme project in exploring language-database integration is the poor performance of approaches that do not offer a seamless combination of programming language and database functionality. Having to make calls on a separate database manager, frequently residing in a different process, and having to convert data between programming language and database formats are examples of performance obstacles that integration can address. (Another motivation for integration is the relatively poorer functionality and less desirable semantics of nonintegrated systems; this has been called the *impedance mismatch* problem [13].) Further, many cooperative, information-intensive applications, especially CAD, are quite demanding. Unpublished estimates for performance requirements include being able to do at least 100,000 references per second to fields of objects, to support "dragging" collections of items on a workstation display screen,[2] and being able to retrieve 10,000 "useful," "typical size" objects per second from external storage into memory.[3] "Typical size" for languages such as CLU [28, 29], Smalltalk-80® [17], and Trellis® [38] appears to be in the range of 30 to 50 bytes; but this is rather informally collected evidence. In any case, the desired retrieval rate is at least a substantial fraction of the bandwidth of a magnetic disk on a typical workstation. The underlying challenge is interesting: designing the store as a separate component *without* incurring the overheads that arise from separate, self-contained database managers.

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 approach, the store should be usable on as many systems

---

[2] This number comes from informal discussion with a Smalltalk CAD tool developer at a workshop.
[3] This goal came from CAD tool developers of a large computer manufacturer.
® Smalltalk-80 is a registered trademark of PARC Place Systems, Inc.
® Trellis is a registered trademark of Digital Equipment Corporation. The programming language component of the Trellis system was formerly called Owl.

as possible, and from a reasonable collection of existing programming languages. While our intent is to support persistent and database programming languages (the integrated approach), it is cost effective and reasonable to require that one be able to write applications that use the store via subroutine calls (noninte-grated), provided this does not compromise support for integrated use.

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, and concurrency and versioning semantics, to be tuned, controlled, and extended. Such policies include clustering, prefetch, caching, buffering, concurrency control, recovery, and versioning.

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 concur-rent users) and autonomy (individual control of resources, ranging from the physical (servers and disks) to the conceptual (subcollections of objects)). Note that the trade-off must not be fixed, since different organizations and different applications have different relative needs for sharing versus autonomy.

Subcollections. 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 from one store (space of objects) into another, so as to exchange data between different organizations, groups not connected to the same distributed system, for backup, and so on. Subcollections can also support autonomy (e.g., if subcollections can be separately "owned" and controlled) and can 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, though, specific implications deriving from the project direc-tion. The store should be suited to integration with some programming lan-guages (current efforts include Smalltalk-80 and Modula-3 [9]), it should allow experimentation with techniques for sharing data cooperatively, and should be a suitable basis for an architecture examining issues of distributed execution as well as distributed storage.

2. CONCEPTS OF THE MNEME STORE DESIGN

We now introduce the architecture, concepts, and semantics of the Mneme store design. We treat the design as an ideal to be approached through a series of prototypes, to be adjusted as experience is gained from those prototypes.

2.1 Architecture

Primarily in order to support the performance goal, we determined that much of the functionality of the Mneme store would be embodied in a subroutine library linked with the application, or, in the case of a persistent programming language, included as part of the language run-time system. If the store software were run in a separate process or on a separate machine, the communication, task switching, and general operating system overhead would substantially interfere with performance. Thus, we decided that much of the detailed object semantics would be implemented within the user's process and address space, though storage facilities are ultimately provided by the operating system or via servers run in separate processes, possibly on different machines. This architecture is illustrated in Figure 1.
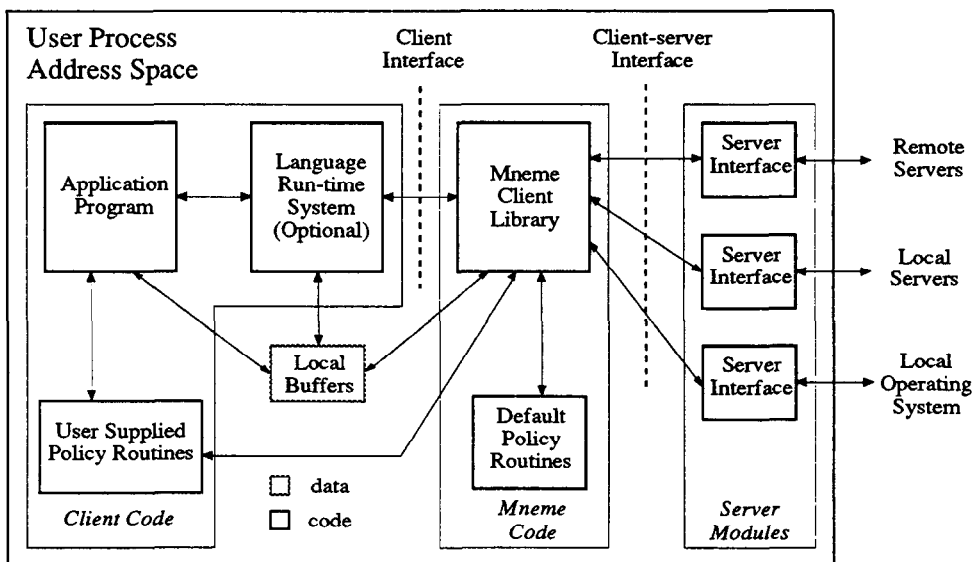
Fig. 1.   The Mneme store system architecture.

Putting the detailed object semantics in the user address space has advantages beyond performance. It also makes it easier for users to extend Mneme's policies; they can simply provide routines for Mneme to call at appropriate points. Servers are likewise accessed via subroutine calls to server interface routines. The initial prototype uses a simple local disk "server," implemented within the client process using standard operating system calls.

The primary abstraction supported by the client interface is that of an *object*, whereas the primary abstraction of the client-server interface is a *physical segment*, which is a vector of bytes accessed using a server-specified identifier. While one could store one object per physical segment, this would tend to be inefficient. Physical segments generally contain many objects, and thus offer means to cluster objects for storage and retrieval, providing hope of attaining our

performance goals. We are not committed to client-server interaction at a single level of granularity, and there is some evidence that multiple granularities need to be offered [14]. We are also not committed to any specific servers or modes of interactions with them; the architecture is designed to be independent of that issue.

We now discuss three issues that arise from this architectural approach: safety, concurrency control, and buffering. Since user code can misbehave in arbitrary ways, putting substantial parts of Mneme in the client space jeopardizes the safety of the Mneme data structures as well as user data. The only way to obtain a high degree of safety, as well as enforced security and authorization controls, is to erect protection barriers, which, on most hardware platforms, implies using a separate process or a separate machine. Thus, servers are the source of true safety, though the compiler and run-time system for a persistent program language can greatly enhance safety by eliminating or reducing the possibility of various errors. Safety is not a primary focus of our work. On the other hand, it is clear that safety and performance may be at odds, since safety demands protection boundaries that generally reduce performance. However, as is the case with retrieval performance, safety performance probably hinges on granularity, and, as we will see, Mneme offers multiple choices for places to associate safety and security controls. In particular, if controls are implemented mostly on physical segments or larger granules, then the performance penalty will be much smaller than if controls are placed on individual objects.

Granularity is also relevant to concurrency control, and again it is necessary to support more than one granularity. In addition to supporting multiple granularities of locks (e.g., lock hierarchies as described in [19]) or some other concurrency control technique such as timestamps or optimistic concurrency control [26], concurrency control may be needed at multiple levels of *abstraction* [32]. For example, objects, which might sometimes move from segment to segment, are at a higher level of abstraction than physical segments or ranges of bytes within segments. Another interesting problem related to concurrency control is that, in a persistent programming language, because we desire transparency, the concurrency control requirements will usually be implicit in the manipulations of objects (reads and writes). A related problem is that different servers may provide different concurrency control features, yet the programmer should be isolated from server idiosyncrasies.

Our approach to these problems is to assume that serializability [16] of reads and writes is adequate in most cases, and that the programmer will become explicitly involved when more subtle concurrency control is required. Section 2.2.5 describes our approach to extensible concurrency control within the object store design. A key challenge is to devise a client-server interface that is independent of server implementation yet offers adequate flexibility. It is clear that concurrency control is ultimately performed by the servers, but the Mneme client software and the layer between Mneme and the underlying servers both perform significant transformations between the (mostly implied) concurrency control requirements of the user and the concurrency features supplied by the servers.

With respect to buffering, Mneme uses client virtual memory to hold objects in use by the client. These buffers are private to the individual client, and the

client typically requests a physical segment at a time, with concurrency control used to obtain the desired degree of coherence and consistency. While the client's buffers are private to the client, it is possible for a server to use a shared buffer pool, either within client memory or in a separate process or machine. It might be possible, although more problematic, to allow a client to access shared buffers directly, via a shared virtual memory area; but this has negative impacts on safety and possibly on transparency (if, for example, locking were required upon each individual manipulation of an object in the shared space).

In sum, putting most of the Mneme object semantics into the client space is necessary for performance, although it introduces other problems. However, these problems can be addressed by varying the split of server functionality between the client space and other local or remote processes, and by varying granularity. Moreover, some of the problems, most notably transparency of concurrency control and the possible need to copy from shared buffers into client space, also arise if Mneme is implemented in a separate address space.

2.2 Concepts

Below we describe the concepts available to Mneme clients. In presenting these concepts and their semantics, we will refer back to our goals for the store to help explain the rationale for the features and choices. The major concepts are objects, files, and pools. Additional concepts include handles (which are used to access objects) and transactions.

2.2.1 *Objects*. A Mneme *object* is a byte vector, with some embellishments, as follows. First, an object has associated with it a unique *object identifier* (id), which allows the object to be located and accessed. Object identifiers are logically equivalent to addresses or pointers; each object has an "address," the object can contain ids that "point to" other objects (or even the same object). A Mneme id is about the same size as a pointer—a 32-bit word, less a few bits to allow for tagging in those languages that need it. Second, the ids stored within an object can be *enumerated*, which helps support garbage collection (among other things). Third, each object has a few associated *attribute bits* (on the order of eight) intended for indicating such properties as whether the object is read-only. The design does not specify their use—attributes are more of a hook for extensions— though some specific potential uses are mentioned. Finally, each object has a (current) *size*. Additionally, one can change the size of an object, replace it by another object, and delete it.

We developed this specific object concept for several reasons, most easily discussed by considering alternatives. Perhaps the simplest notion of object, used in some well-known designs (e.g., [10, 20, 39]), is a vector of bytes alone. Our addition of attribute bits is not very profound; requiring that ids inside objects be enumerable is more interesting. We do it partly so that we can garbage collect the store, but it also allows us to use different forms for identifiers in client memory versus external memory, which is described further in Section 2.2.3.

Our approach to enumerating ids in objects is for Mneme to rely on a small number of client-provided routines to tell Mneme where the ids are within an object when Mneme needs to know. Thus, the client is free to use any appropriate

descriptor or format mechanism. For example, Smalltalk and Trellis segregate pointer and nonpointer data and have only a small number of simple formats, which require run-time support anyway. Languages that make more distinctions at compile time might need to produce tables and label objects with their type/format.

Even though we go beyond simple byte vectors, our object concept has no notion of "class" or "type." This is not a problem, since one can simply establish a convention for languages needing class/type information. For example, the first few bytes of each object might contain the class/type of the object.

We allow for tagging of ids. The main reason for this is to fit better with Smalltalk and similar languages. If a language does not require tagging, then it will not incur additional overhead in supplying or manipulating tags. The main implication is that ids are limited to 28 bits. (How we support an effectively unbounded space of objects is explained in Section 2.2.3.)

Just as we could have chosen richer object semantics than we did, we could have chosen a richer object relationship model, e.g., binary or higher order relationships, rather than just objects and pointers (ids). This would not have matched up well with existing programming languages. It would also have required more implementation effort and might have introduced performance problems. The difficulty is that relationships have many alternative representations, a number of which must be offered in order to obtain good performance over a range of cases. Further, as more sophisticated data model features are added, one requires more of a data manipulation language, which encroaches upon the programming language's responsibilities. We feel that such data models can and should 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 probably would not correspond directly to a single Mneme object; that is, higher level relationship models probably imply higher level object models as well. 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, the object can contain ids to describe its pointer relationships to other objects, the ids can be easily enumerated (for garbage collection, etc.), and the object can contain any non-id data desired. This structure is simple, general, and efficient in storage and access.

2.2.2 *Handles.* While Mneme could support all object manipulation by having the client present ids, this would require looking the ids up in a resident object table prior to each access. To allow the client to amortize the lookup cost over a series of manipulations of the same object, we introduce *handles*. A *handle* provides efficient access to the internals of a resident object. Specifically, a handle contains the object's id, a pointer to the data part of an object, the size of the object, and so on. Not only does using a handle obviate repeated lookup and avoid object residency checks, it also permits tight encodings to be used in object headers without sacrificing performance, since the header can be decoded once

and the information stored in the handle for repeated use. Thus actual manipulation of object contents is done via presenting a handle rather than an id.

Clients provide space for handles, which are "created" (filled in) by giving Mneme the object id and a pointer to the handle space. When access is no longer desired, the handle may be "destroyed," meaning it will no longer be used to access the object in question. The actions of creating and destroying handles serve several purposes. First of all, these actions delimit a period of time during which an object is actively being accessed and must be available to the application. Thus, handle creation and destruction can be used to indicate the times at which locks should be acquired and released (at least conceptually). Creating a handle requires that the object be located, which involves searching a table of resident objects, and, if the object is not resident, the object must then be fetched from external storage, causing what we call an *object fault*. Thus, handle creation and destruction provide information essential to buffer management.

The main drawbacks of handles are 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. On the other hand, while handles may be adequate for direct use of the store by some applications, they may not be the best approach for tight integration with a programming language. Therefore, in addition to access via handles, we allow applications to obtain pointers to objects in the buffers and manipulate objects directly. This avoids the subroutine call overhead and the indirections and checks of the handle interface, but requires more care on the part of the programmer. We have explored the performance aspects of handles versus direct pointers in more depth elsewhere [31]. In brief, direct pointer access substantially boosts performance of object manipulation compared with the handle interface. In order to delimit a span of use of an object, after obtaining a pointer and using it, one must *release* the pointer, informing Mneme that the particular pointer is being discarded and will not be used further. One may obtain the pointer again, given the object id, and begin another span of use, as often as desired. Handles allow updates to be noted by the handle routine performing the updates. When pointers are used, the application must inform Mneme if the object is modified. This may be done when the pointer is acquired, when it is released, or any time in between.

2.2.3 *Files and Locality of Ids.* Mneme groups objects together into units called *files*. A file of objects can be separately named and located within an overall distributed store. A typical implementation of the concept associates individual files with specific server machines, although 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. We are exploring whether and how to allow objects to move from file to file, which may be important for global garbage collection (cf. [8]).

Files are a convenient unit for storage, and provide modularity of the object space, one of the stated goals. We intend 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 the 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 objects containing the id. This allows ids to be relatively short—less than 32 bits, as previously mentioned. 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. By convention, a field in a forwarder indicates the *forwarding protocol* to be used; the protocol numbers are then mapped to forwarding protocol routines via a simple table lookup, and the forwarding routine is called to interpret the rest of the contents of the forwarder. This scheme allows us to add new forwarding protocols at will.[4] Forwarders also allow application mediated versioning schemes to be implemented easily and transparently.

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® environment variables or VAX/VMS® 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. However, we do assume that relatively few objects will need such names. Autonomy is supported in that the mapping table can have procedural hooks, analogous to forwarding protocol routines, to indicate actions to be taken when an external name is looked up. Since the mapping routines can involve arbitrary code, they can do authorization checks, supply different objects to different requesters, synthesize data on demand, initiate prefetch, and so on.

Note that the mapping tables are essential, since ids cannot simply be synthesized and presented to the Mneme store functions with any reliability. This is because Mneme may reassign ids of objects within a file, to support reclustering, garbage collection, reuse of the limited space of ids, and even explicit object deletion. The tables themselves are not expected to take much space relative to the objects, since only a small percentage of objects will have external names and thus be entered in the tables. For example, a CASE tool might store program

---

[4] Whether such code can be loaded dynamically is an implementation issue we have not explored.
® UNIX is a registered trademark of AT&T Bell Laboratories.
® VAX and VMS are registered trademarks of Digital Equipment Corporation.

parse trees in Mneme. In this case only the root of the tree needs an external name, whereas the parse tree likely contains hundreds to thousands of objects.

We believe the Mneme file concept is good because of the way it simultaneously enhances modularity, autonomy, extensibility, and compactness (short ids). Most notably, the scheme keeps ids short (less than 32 bits), but supports an object space of unbounded size. The file 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 present more detailed arguments in [30].)

One alternative approach is a very large shared distributed virtual memory, where object ids are addresses naming bytes or words. Problems with this approach include lack of object semantics, difficulty in garbage collection, the need for long addresses and large pointers, and poor support for autonomy, modularity, and extensibility. Of course, object semantics can be provided at a higher level; and that is what Mneme does. Mneme memory management is simplified and made more flexible by not having object ids linked to precise physical locations. This is because object ids with some degree of location independence allow some movement without id reassignment. Thus, we can do some compaction and reuse of space without garbage collecting the entire store. Garbage collection is more of a problem in a shared address space, since the whole space may have to be searched to determine reachability. The segmentation introduced by files, similar to the areas of [8], makes garbage collection more realistic.

The main advantages of the virtual memory approach are simplicity and familiarity, although the simplicity may be only apparent, not real, when we consider administration of a very 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 [34] or the Intel 432 system [22, 35].

Another alternative to the Mneme addressing scheme is immutable object identifiers. Sometimes provision of such identifiers is seen as equivalent to supporting object identity, although we argue that object identity and persistence of immutable *names* for objects are distinct ideas. For further discussion of the concept of identity, see [24]. Note that Mneme supports identity in the sense of preserving the graph structure defined by object references, but 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.

Immutable object identifiers present three major problems, all related to performance. First, in a large system, they will have to be long. This might be ameliorated 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) result. (If objects move a lot, Mneme will also incur time overhead chasing through forwarders.) The second 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. Prefetch and clustering will reduce the impact of this problem, but cannot entirely eliminate it. Mneme (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. The third problem is internal and external object table size. If object ids are entirely independent of object location and clustering, then a per-object mapping table is needed for finding objects in external storage, and a per-object table is also needed for locating objects currently resident in main memory. In both cases the tables must be able to handle sparse key sets. Mneme uses smaller external mapping tables, and faster and probably more compact internal mapping tables, deriving both speed and space benefits compared to immutable object id approaches.

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.

2.2.4 *Pools and Strategies.* Mneme files are physical and logical units of grouping and naming objects: physical in that files physically contain objects, and logical in that files are visible to Mneme users and are involved in the naming of objects, although cross-file references can be followed transparently, unless access is denied or some other difficulty arises. Mneme *pools*, on the other hand, are logical, and not directly physical, groups of objects *within* files, and pools are not directly involved in object naming. Pools do not span files because each file is an autonomous collection of objects, bound to other files only via the cross linking of the forwarder mechanism.

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 pointers to 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. In addition to determining object management policy, pools support scanning the objects they contain, and thus are useful for grouping objects for later query processing.

Let us consider two examples of the pool concept in action. The routine for creating an object has the following inputs: the desired size for the object, the initial attribute settings, the pool in which to create the object (which implies the file as well since pools are units within files), and 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 chooses the physical segment in which to place the new object.

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 determines

whether to prefetch additional data, how the object should be locked, the buffer replacement policy to be used, and so on. If the requested object is a forwarder, then further policy and forwarding/mapping implementation routines become involved.

Recall that policy routines reside in the client's address space. For the strategy mechanism to work correctly, strategies are assigned numbers by an appropriate authority (analogous to forwarding protocol numbers), and standards are developed as to what each strategy should do. Pools indicate their associated strategy by strategy number, and the client tells Mneme the location of the routines implementing each strategy, so that Mneme can fill in the strategy routine tables. Setting up the strategy tables is frequently done just after initialization, but routines can be added while the system is running. Dynamic loading of strategies, if provided, is up to the client.

Thus, 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 into the Mneme store code. If an application attempts to use an object whose pool strategy vector has not been set up, an error handler is called.

Pools support policy/mechanism separation and policy extensibility. The current design leaves open as a research question the detailed design of the strategy routine interfaces. The pool concept supports flexible approaches to object clustering, storage allocation, prefetch, concurrency, consistency, buffering/caching, and perhaps even security, versioning, and other issues.

2.2.5 *Transactions.* Since the Mneme store is intended to support exploration of techniques for cooperative sharing, some kind of concurrency control mechanism is required. Serializability is a clean, unifying correctness condition that has enabled the development and evaluation of many concurrency control protocols that prevent interference. Unfortunately, there does not yet exist such a simple and unifying correctness condition that allows all the forms of cooperation that appear to be desirable. The approach we have taken in Mneme is to support serializability, in the belief that most of the time it is appropriate, and to provide additional nonserializable primitives for those situations where serializability is too limiting. The key here is to provide primitives with reasonably simple, yet fully general, semantics, so that application-specific concurrency control is not overly difficult to build, yet virtually any protocol can be implemented.

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. All Mneme object manipulations occur within transactions. The intent is that so-called "long transactions," "design transactions," or "cooperative transactions" that 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.

As indicated above, the design includes a basic transaction model that supports serializability, augmented with facilities to build application-specific, nonserializable 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 modified. 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 (they must produce consistent serialization orders).

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 more restrictive 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, as in [19]). This allows use of simpler and more efficient techniques to boost performance when the highest concurrency is not required. For example, when an entire design file is checked out, it is not necessary to lock each individual object within the file.

Another point to consider is that any locking or concurrency control is a side-effect of acquiring a handle, updating slots or bytes, and so on, similar to the automatic locking typical of database systems. That is, there are no explicit *lock* or *unlock* calls performed by clients, no additional lock modes, and so on. 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 two basic pieces of functionality: *mutual exclusion* and *notification*. This functionality is intended to provide concurrency control and synchronization, as needed.

Mutual exclusion is supported via the introduction of a new concept: *volatile pools*, which work as follows. When a client acquires a handle on an object stored in a volatile pool, Mneme gives the client exclusive access to that object until the handle is released. Similarly, mutual exclusion is guaranteed from the time a direct access pointer is acquired until it is released. When the handle (or pointer) is released, the object may be accessed and/or updated by other processes. Thus, volatile pools provide "short locks" on objects, with the result that the object contents are volatile, in the sense that they can appear to change spontaneously.[5] The mutual exclusion provided by volatile pools is adequate for building any desired concurrency control semantics. The short exclusive-mode lock is used for locking the data structure containing lock and scheduling information for the user-defined locks, similar to the techniques of [44]. While this approach is fully general, interesting performance questions remain, such as how best to cache volatile objects, which can be treated differently by different volatile pool strategies.

---

[5] This notion of volatility is not related to hardware crashes or other forms of memory loss.

Notification is required to support continuation after waiting for a transaction to change state (abort, commit, etc.). Mutual exclusion plus notification are adequate to implement application-specific locking. Notification has two aspects: registering interest in particular events and receiving notices of the events as they occur. Registering interest in events is straightforward. Receiving notices in a general way is a bit more subtle, since Mneme is not aware of any multi-threading that might or might not be present in the application. Our solution is to have the application designate a subroutine to be called when the event occurs. This subroutine may take action directly, or may simply register the event occurrence so that the application run-time system will process it later. The Mneme design provides for notification of transaction state change and for time expiration.

For a number of reasons, crash resiliency is a more difficult problem to handle well: it can noticeably affect performance, it tends to require a single global strategy, it has pervasive effects on the software, and it is a major source of system complexity. In Mneme, we chose to adapt the recovery methods of the POSTGRES storage manager [41]. The idea is to locate (insofar as possible) log information pertaining to a particular segment of data in that same segment, with the only global log being the record of committed transactions.

This approach is appealing for Mneme because it allows each pool to take a different strategy as to the details of logging and recovery for the segments of objects in that pool, and minimizes the common facilities to the transaction commit log. Localizing log information with the affected objects also makes sense for volatile pools: when the objects are recorded, their log records will be also, with no additional effort. Further, any connection made between the log information in the segments and the transaction log is up to the pool strategy routines. The POSTGRES approach is also appealing because it leads quite naturally in the direction of support for temporal or historical data as well as simple versioning.

The extent to which the POSTGRES approach suffers from performance difficulties in practice remains to be seen, given that we are not assuming the availability of stable main memory. We expect that logically storing the log information with the segments, but recording changes physically using techniques such as the database cache [15, 33], will help overcome the potential performance problems.

In summary, the Mneme store design includes a basic transaction facility, a transaction extension facility, and support for crash resiliency. 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.

2.3 Summary of Design Concepts

Now that we have introduced the Mneme design, it is perhaps helpful to review the main contribution and value of each concept of the design:

—*Objects.* Clearly, we must offer objects of some kind; the issue is exactly what kind. We provide a simple model intended to offer good performance, to be a

good base on which to build more sophisticated semantics, and to be a good match for programming language implementation.

—*Files.* Files, combined with the forwarding mechanism, address many of the significant issues of distribution, especially autonomy, while encouraging transparency (automatic following of forwarders) and allowing extensions (new forwarding protocols, etc.). Files also provide modularity of the data in the store, making it easier to garbage collect (among other things).

—*Pools.* The primary advantage of pools is that they allow different collections of objects to be managed under different policies, even within the same file. This is ultimately important for performance. Adding or modifying pool policies (strategies) is also the most significant way in which Mneme can be extended.

—*Physical segments.* By grouping multiple objects together into a single chunk for transfer and storage, segments improve performance. Also, since segments are very simple (vectors of bytes) and at a lower level of abstraction than objects, they are a good basis for the interface between Mneme and external storage servers.

—*Transactions.* Transactions and sessions provide the concurrency control and synchronization crucial in a system allowing concurrent work. Volatile pools allow new concurrency control techniques to be devised and implemented without fundamental changes to Mneme.

3. THE INITIAL PROTOTYPE

We have built a working prototype Mneme store. We had two main goals for the prototype: to explore certain aspects of the design and to provide support for our persistent programming language implementation efforts. In terms of the design, we were most concerned about the performance of the interface to objects, especially when accessing and manipulating objects resident in buffers. We were also concerned about I/O performance. The prototype is thus, in a sense, a feasibility study to see if adequate performance can be achieved. With respect to persistent language support, we are developing implementations of Persistent Smalltalk and Persistent Modula-3, which pose different problems since they are at opposite ends of the interpreted versus compiled and run-time versus compile-time type checking spectra. Efficient persistent languages impose two primary demands: rapid access to resident objects, and compact objects and object identifiers. If these demands are not met, then the persistent language (when manipulating resident objects) will compare unfavorably with nonpersistent languages.

Our goals for the first prototype do not require all features of the design, so we implemented only those features needed. In some cases we provided limited implementations of features as well. For example, the prototype supports at most one million (2^{20}) objects per file, stored in at most 1024 segments. We emphasize that the design is itself a valuable contribution, without a complete implementation, because important strategies and approaches (such as the file concept) have been articulated and have influenced the prototype. Further, the implementation is also worthwhile because it has given useful experience with the design,

allowed interesting performance measurements which will guide further design, and provided support for persistent programming language implementation.

3.1 Differences from the Design

We now describe the primary ways in which the prototype is different from the design presented earlier. First, there is no explicit support for distribution, multiple users, or resiliency. Clearly, the decision to omit these features from the initial prototype greatly simplified implementation, but since we were most concerned with testing the object interface performance with resident objects and raw retrieval speed, we did not need multiuser or distributed systems support. It is also clear that adding such support to the system is not likely to increase performance. We do believe, however, that it will not dramatically *decrease* performance either. This is because, as previously argued, we expect concurrency control and retrieval to be done mostly in terms of medium to large granules, meaning the per-object overhead will be small. Further, we are most concerned with applications that manipulate a considerable volume of data, so that most transactions generate more than few updates; this suggests that the incremental cost of transaction commit records, for example, is small. In any case, we wanted to see if the object interface could *possibly* reach our performance goals, and we have purposely separated the question of whether or not transaction and distribution support has dramatic or only incremental effects. In fact, having the nontransaction system as a benchmark will enable us more easily to determine the costs of the transaction features when they are added.

Another difference is that the prototype does not support extremely large collections of objects. It allows (as mentioned above) up to about one million objects per file and about one thousand segments per file. While multiple files are supported, cross-file references are not. Again, files with up to a million objects were adequate for us to get started. Further, we have sketched out in some detail the mechanisms necessary to support larger files, and firmly believe that the crucial performance measures will not change much as we move from current mechanisms to more capable ones. Some of the necessary changes are discussed in more depth later. Our main reason in not supporting larger files in the first prototype is that it is a bit more complicated, and we desired to have the prototype operational as soon as possible.

A third difference is that the prototype supplies a small number of built-in object formats within Mneme, rather than relying on the client to deal with object formats and id enumeration. In particular, the prototype segregates ids and bytes, and uses a particular tagging scheme. Ids are stored in *slots*. A slot can contain an id (which can be null, represented by 0) or an immediate value; the two cases are discriminated using the sign bit of the 32-bit word. Since garbage collection is not yet implemented, the slot/byte segregation and built-in tagging scheme are not all that important a difference from the design. Also related to objects, the prototype does not support changing an object's size or replacing it with another object.

Finally, the prototype does no buffer management in the sense of automatically choosing and purging buffers of objects. Because of the direct pointer interface, this could be problematic. There is also an interesting tension between associating

buffer management with Mneme pool strategies—which is natural in the Mneme framework—and the obvious point that buffer management is necessarily global, since it involves managing memory as a global resource. Our view of the performance-critical aspects of applications is that they must be able to retrieve a design file rapidly, manipulate the design file efficiently, and propagate updates rapidly when done. If we view this model of persistent programming applications as primary, then buffer management is not the most important issue, since an application proceeds in distinct phases of loading, working, and saving. For example, in a CAD tool, one is likely to load a design file (retrieve all the relevant objects in the file), work on that file for a period of time, and then store back any changes at the end of the work session. We were mostly interested in whether the Mneme object interface could possibly be efficient enough for this style of work. Hence, advanced buffer management was not important in the prototype, and was not provided. We simply allocate virtual memory at will and write modified segments back only when their file is closed.

3.2 The Client Interface

We now summarize the C routines available to clients of the Mneme prototype. We begin by describing the style of the interface. It is relevant because good style in interface design makes a considerable difference in usability and also affects how frequently users of the interface make mistakes. Here are the principles we arrived at, some from the start and others after draft interfaces had seen some use:

—A number of uniformity conventions were followed, to make the interface easier to understand and remember. For example, every routine returns a result code with a uniform interpretation.

—Since users are notoriously slack about checking for error result codes, we provided an exception reporting and handling feature. Mneme maintains a stack of error-handling routines and the user may push and pop routines on that stack at will. When an error occurs, the top routine is presented with information about the error and can handle the error, cause Mneme to return the same or a different error code, or pass the error on to the next routine down the stack. This feature has proved to be of considerable assistance in finding and resolving bugs, in applications as well as in Mneme itself.

—All dynamic (as opposed to persistent), visible (as opposed to internal) structures are allocated and freed by the user. This improves performance by allowing local stack space to be used wherever possible, avoiding calls to manage heap-allocated memory.

We now summarize the functions available, in related groups. Where relevant, we describe the actual interfaces of routines.

3.2.1 *Object, Handle, and Pointer Operations.* There are a few functions related to object ids alone (object deletion, identity comparison, and existence checking), but most object manipulations are performed via handles or direct pointers. It is

useful to consider the interfaces of the object-creation routines:

```
MnObjectCreate      (poolid, nslots, nbytes, attrs, near, hnd, id)
MnObjectCreatePtr   (poolid, nslots, nbytes, attrs, near, ptr, id)
```

They show that creating an object requires identifying the pool in which to create the object (that pool's strategy routine does the actual placement decision), which implies the file as well, since pools do not span files. We must also specify the size of the object (number of slots, number of bytes) and the initial attributes. The near parameter is passed to the strategy routine and is generally used to indicate the identity of an existing object near which the new object should be placed. Finally, hnd, ptr, and id are the handle on, direct pointer to, and id of the new object, respectively.

It is also instructive to consider the interfaces to the routines for obtaining and releasing handles and pointers:

```
MnHandleCreate     (id, hnd)
MnHandleDestroy    (hnd)
MnIdPtr            (id, mod, ptr)
MnIdRelease        (id, mod)
MnIdMod            (id)
```

MnHandleCreate and MnIdPtr are the routines that can cause object faults; they take an object id as input and return a handle or direct pointer. The pointer routine also allows us to indicate that the object will be modified, should we happen to know that. MnHandleDestroy and MnIdRelease are the routines that mark the end of a period of use of an object. MnIdRelease allows us to note that the object has been modified, and MnIdMod allows such notation to occur at any time, since it may not always be conveniently known when MnIdPtr or MnIdRelease are called. Recall that in the case of handles, all manipulation is done via additional Mneme routines, which can automatically record if an object is modified.

Additional handle routines allow us to determine information about the object (its file, pool, number of slots, number of bytes, and whether it has been modified) and to access and update its slots, bytes, and attributes. We also provide the ability to copy ranges of slots or bytes from one object to another and to fill a range of slots or bytes with a user-specified value.

Since a direct pointer allows most of these operations to be performed without Mneme calls, there are fewer operations related to these pointers, but one can determine the number of slots and bytes in an object as well as access and update the attribute bits. An interesting aspect of the direct pointer interface is that clients can observe the *external* (on-disk) format of the Mneme ids, rather than the usual client form. Recall that with the on-disk form, an id is relative to the file containing the object. To simplify and speed up manipulation of ids in external format, a number of the id-related operations have corresponding forms that accept a *file identifier* (uniquely identifies a currently open file) and a *persistent id* (the external identifier of an object in the file). There are also routines to map between the internal and external id format. The additional routines are needed as a consequence of our decision to support direct access to

objects in the Mneme buffers, as well as an implementation decision that the buffer contents would be in the on-disk format. (Clients may change the internal format of objects in the buffers, so long as the external format is restored by the time the buffer is written back to disk.)

3.2.2 *Pool and Strategy Operations.* Pools can be created and destroyed, and they have associated attributes, allowing them to have arbitrary client-specified (and strategy-specific) parameters. In the 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 at once. The only interesting interface to consider is the one for pool creation:

    MnPoolCreate   (fileid, stratid, poolid)

Here fileid and stratid are inputs, indicating the file to contain the pool and the strategy to be used. The output is poolid. A pool id is used in object creation, as we have seen; one can also determine a pool's file from the pool's id. While we have not done so to date, it would be straightforward to provide routines to iterate through all objects in a pool, all currently resident objects of a pool, and so on.

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 the 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. This interface was not developed much in the prototype, and relatively fixed policies were used, with the result that the currently defined strategy events are not very interesting. The interesting aspect of this feature is its design and future extensibility rather than the current implementation.

3.2.3 *File 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. This idea is important because it simplifies the interfaces of the file operations and helps to prevent operating system peculiarities from creeping into them.

The basic file manipulation operations, and the file attribute routines, are straightforward. We provide routines to create, delete, rename, open, and close files; to test whether a file of a given name exists; and to get, set, and remove file attributes (name-value pairs stored in the file). File attributes are intended to help support things such as tagging files with the name of the application that built them.

A number of additional inquiries are provided related to open files (named by *file identifiers* returned by the file open routine). The most interesting routines are those that get and set the *root object* of a file. Since the prototype does not

support automatic cross-file references, the incoming reference table mechanism was not needed in its full generality. Instead, we provided one starting point in each file: the *root object*. When a file is created it has no root object, but at any later time, any object in the file can be designated as the root. This has proved adequate for the simple applications we have dealt with to date. It is also a good starting point for eventual incoming reference table functionality: we can hang the more advanced structure off the root object. To do so we need only develop routines for managing the necessary dictionary data structure and offer them for general use. The root object approach currently allows each application to use whatever approach is appropriate. The eventual implementation of cross-file references will require future standardization of these lookup mechanisms, or of aspects of them, at least. The root object idea may turn out to be useful *in addition to* the incoming reference table: objects reachable from the root object in some sense "belong" in the file, whereas those that are reachable only from the incoming reference table are candidates to move to other files. (This idea comes from [8].)

3.2.4 *Miscellaneous Operations.* The interface includes several operations for acquiring statistics and counts. These are clearly very important, considering our goals of measuring performance. Several routines are related to tracking and reporting information about the contents of files (number of objects, number of segments, space consumed, etc.) and several with dynamic statistics (number of faults, volume of data moved, number of calls to certain routines, etc.). There are also a number of other minor functions to handle matters such as initialization, shutdown, the error handler stack, and so forth.

3.3 Implementation

The prototype is written in C [23], and has been run on the VAX/VMS, Ultrix®, and SunOS® 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 type-unsafe operations necessary for such things as imposing object structure on raw bytes retrieved from a server.

While most aspects of the prototype implementation (table design, algorithms, etc.) are straightforward, the implementations of object identifiers and segments are deserving of more detailed discussion, since these implementations are crucial to good performance.

3.3.1 *Client and Persistent Identifiers.* As previously mentioned, the on-disk and client forms of object identifiers are different. While perhaps not obvious, the reason is straightforward. Since each file autonomously maintains its own space of object identifiers naming objects within the file, two files can end up using the same identifier, producing a clash if we attempt to use both files at the same time without some sort of mapping. Now we *could* add a file identifier to

---

® Ultrix is a registered trademark of Digital Equipment Corporation.
® SunOS is a registered trademark of Sun Microsystems.

the object identifiers to disambiguate, but we desire to keep object identifiers short: they should fit in a single machine word, with room left for a few tag bits. This is important in obtaining good performance: multiword identifiers not only take more space within objects (making objects larger) in a general sense, they also take more space at each level of the memory hierarchy, and thus increase cache misses, virtual memory page faults, buffer replacements, and so on. The minimal reasonable expansion from one machine word is a doubling to two words. Further, since we are talking about persistent objects, larger ids mean more disk space, and possibly longer seek times (because things will be more spread out). Larger ids will also consume more disk and network bandwidth for transmission. Further, manipulating multiword identifiers will take more instructions. In short, identifier length has a global effect on performance. This issue is explored in more detail in [30].

Using short ids does mean that to disambiguate object ids from different files requires some sort of mapping between the form in the files (which we call *persistent ids*) and the form ordinarily used by clients (*client ids*). If we are not to lose the performance benefits of short ids, the mappings between persistent and client ids must be time and space efficient. In the prototype, because the number of objects per file and the number of open files were both limited, we could actually concatenate a file number and an object number within a single machine word. However, we have designed techniques that do not cheat in this way, so as to convince ourselves (and others) that the mapping is efficient enough.

First, for files that are within specified size limits in terms of the number of objects they contain, we can still append the file identifier to the object identifier. This is accomplished by having a boundary in the middle of the id field such that the file number is to the left and the object id (persistent id) within that file is to the right. Having a fixed boundary is limiting, so we use two id bits to distinguish between any of four boundaries. To make this more concrete, suppose ids are 28 bits long. We use one bit to choose between this scheme and another to be discussed in a moment, two bits to choose the boundary, and have 25 bits remaining to split between file numbers and object ids. While the boundaries may need tuning as we gain experience, here is a reasonable initial choice for the four boundaries:

Tag	File bits	Number of files	Object id bits	Objects per file
00	11	2048	14	16384
01	8	256	17	131072
10	5	32	20	1048576
11	2	4	23	8388608

The point is to allow many files, to allow large files, and to allow both to be mixed. Note that this mapping approach always includes *all* objects of a file, even if not all of them are being referred to by the application. On the positive side, this mapping is quite fast and requires no tables if file ids are assigned appropriately. Note that if a file grows while it is in use, such that it has more objects than permitted by its current mapping, we can institute a new mapping in the next larger category (maintaining the ability to map previously converted client

ids by permitting aliases). The details of the scheme are not crucial; its speed and flexibility in terms of numbers and sizes of files are its important features.

What the variable boundary scheme does not handle is a large number of large files. Of course, we have already imposed a firm limit of 2^{28} client ids in use at a time by restricting ids to 28 bits, and a large number of large files would tend to violate this hard limit. The interesting case is a large number of large files where we are naming only a portion of the objects in each file. To handle this situation, we introduce table-supported id mapping. Clearly, it would be expensive in space to map individual ids; we choose to map them in "chunks" of 2^{14} identifiers. Each open file being mapped this way has a table indicating the upper 13 bits to use for each group of 2^{14} identifiers in the file that have an assigned mapping. This supports mapping from persistent ids to client ids. The reverse mapping, as well as allocation of the "chunks," is handled by a single global table in the client with an entry for each of the 2^{13} patterns in the upper bits. While slower than the variable boundary technique, this scheme is still fast; the choice between the two schemes is controlled by a single bit in file identifiers and client identifiers.

3.3.2 *Logical Segments.* In addition to mapping between persistent and client identifier forms, we must be able to locate objects efficiently, on disk and in memory. The concept of *logical segments* plays an important role in locating objects with high performance. A logical segment is a set of object ids that have the same high-order bits and vary only in the low-order bits. In the 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 *physical segment*; recall that a physical segment is the unit of communication between Mneme and servers. Thus, rather than mapping individual objects to physical segments, we need only map logical segments to physical segments. Since there are 1024 objects per logical segment, the mapping tables can be up to 1024 times smaller than a per-object table. The smaller size allows the map to fit in main memory when a per-object map may not, and thus the logical segment grouping allows most object faults to be satisfied with one disk retrieval. Logical segments were introduced to improve performance and are not required by the Mneme design.

Since each pool has an associated object management strategy, each physical segment is contained in precisely one pool, so that the physical segment is under the control of a unique strategy. Further, since each logical segment is contained in precisely one physical segment, each logical segment is contained in a single pool. Thus, at a physical level, each pool manages a set of physical segments, and at a logical level, a set of logical segments.

In the prototype, files are limited to 2^{20} objects, leading to the following object location mechanisms. Given an object id, the file number is in some upper bits; this is used to index the file table. An entry in the file table includes a pointer to the file's logical segment table, which is brought into main memory when the file is opened. The lower bits of an object id can be broken into a logical segment

number (10 bits) and the number of an object within that segment (10 bits). The logical segment number is used to index the file's logical segment table; the logical segment table entry indicates whether the logical segment is resident, and if so, where it is. A logical segment has a header containing self-relative pointers to the (up to) 1024 objects within the segment, which can be thought of as a fragment of an object table. The object number bits of an object id are used to index this table fragment and find the actual object. If the object is *not* resident, the logical segment table's pointer to the resident copy of the segment will be null. The logical segment table entry also indicates the physical segment containing the logical segment. This physical segment can be requested from the file's server and loaded into a buffer. This two-level direct lookup scheme is illustrated in Figure 2.

Since the client id format will change in the future, the object-location mechanism will also change; and it is important to convince ourselves that it can be about as fast as the two-level technique. First, we note that the crucial code path is locating resident objects, since a few additional instructions to deal with a nonresident object will not affect fault time noticeably. In the future we will substitute a hash table for the two-level scheme. The hash procedure will take the upper 20 bits of an id, probe the hash table, and either fail (the logical segment is not resident) or produce the address of the logical segment's object table fragment. Given a simple hash function (e.g., extract some bits) and sufficiently low load factor on the table, this should perform approximately the same as the two-level lookup, on average.

Locating logical segments on disk can still be done via direct lookup, or, if we are concerned about the space consumed by tables when not all of a file is used, we can substitute a multiple-level lookup scheme (e.g., a B-tree), at the cost of threatening the one-disk access property that the logical segment concept attempts to provide.

3.3.3 *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.

While the details of physical segment layout may not be crucial, for concreteness we describe the format used in the prototype. 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 entry in the file's physical segment table contains the various storage management pointers for
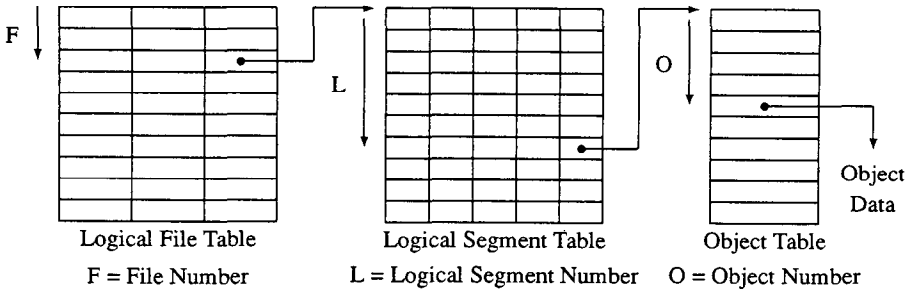
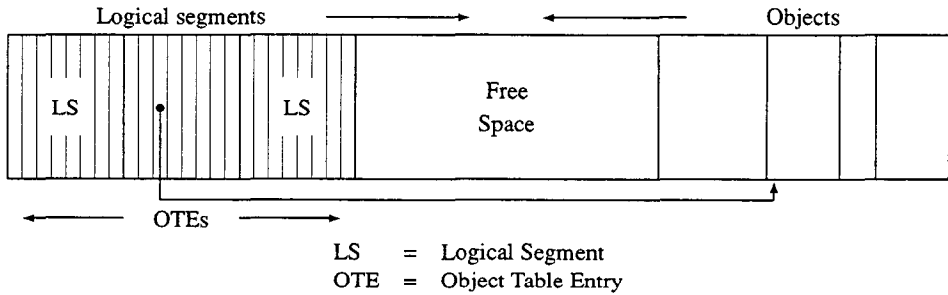Fig. 2.   Two-level direct map object lookup scheme.

Fig. 3.   Physical segment data structure.

memory allocation within the segment. Free OTEs are chained through the OTEs themselves, with the head being in the physical segment table entry. The physical segment structure is illustrated in Figure 3.

Objects are associated with Mneme pools via the physical (and hence logical) segments that contain the objects. This means 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).

3.3.4 *Server Interface.* The server interface is worthy of brief comment, though there are no surprises. We expect this interface to evolve from prototype to prototype, especially since there are currently no provisions for distribution or concurrency. The interface includes the notion of a *file*, which corresponds to a file at the client interface. There are operations to create, delete, rename, open,

and close files, and a file existence test. Files also have a *header* associated with them and stored inside them. Mneme can fetch and store the header explicitly. Since servers assign physical segment ids, the header gives Mneme a place to get started without a segment id, just as the root object of a file gives Mneme clients a place to start without an object id. The only abstraction supported other than files is physical segments. They can be created and deleted; contiguous regions of a segment can be fetched and stored; and a segment can be grown (this feature is questionable, since it can always be accomplished by creating a new segment and copying over and deleting the old one).

4. PERFORMANCE RESULTS

We have measured various aspects of the performance of the prototype and present some relevant results here. The primary comparison we make is with nonpersistent forms of similar manipulations. In addition, there are absolute measures, and some comparison with the Exodus storage manager (ESM) [10]. The comparison with ESM is necessarily rough, because it provides different abstractions and different functionality. Thus, though Mneme performs better than ESM on most of the tests reported here, those tests are not necessarily representative of what ESM was designed to do. Most of the performance results also appear in [31], although in the context of a somewhat different research question.

4.1 Measurement Approach

We measured the prototype, for the most part, with an instrumented program that constructs and traverses trees. In these trees, we can control the branching factor, the number of additional slots of internal nodes, the number of slots of leaves, the number of bytes of internal nodes, and the number of bytes of leaves. To allow the smoothest variation in the total number of nodes, we used binary trees in the measurements. The program can construct and traverse nonpersistent trees, as well as those stored in Mneme. It is written in C and runs on a DECStation® 3100 system; further details appear in [31].

We consider the performance of various user-level operations, examining in turn object creation, writing objects to disk, reading objects from disk, and locating and manipulating resident objects.

4.2 Creation

The table below presents measured per-object creation times (except the malloc line, which is an estimate). These are CPU times, but the elapsed times are almost exactly the same, since creation does not involve I/O (objects are created in main memory buffers; saving them to disk was measured separately). The "custom" allocation method is linear allocation (bumping a single pointer followed

---

by a limit test), with no special provision for reclamation other than reusing the entire heap space. The malloc cost was estimated by comparing the cost of a series of custom allocations versus a series of C library malloc calls. The other two columns are the measured cost of creating a Mneme object and returning a handle or a direct pointer, respectively.

Software	C, custom	C, malloc	Mneme, handle	Mneme, direct
Time, μsec/object	9.0	11.4	37.8	33.1

We see that creating persistent objects in Mneme buffers costs three to four times as much as allocating nonpersistent objects in a main memory heap. In all cases, allocation cost was independent of object size within the range of sizes tested (about 10 to 1000 bytes total), regardless of whether the object was mostly slots or mostly bytes. Some of the performance difference between C and Mneme results from the calculation of Mneme object size from number of slots and bytes, the initialization of the object header, the formation of the object id, and so on. While it may be possible to reduce the factor of three or four somewhat, allocating persistent objects does involve more work, if only to allow strategy variation and more sophisticated choice of object placement. ESM was about four times slower than the Mneme direct access interface, so we feel that Mneme's object-creation performance is good, even though it is substantially slower than C. We should point out, too, that persistent programming language implementations will create most objects in a transient heap, and propagate them to the persistent store only when they really need to persist (i.e., when they are found to be reachable from a persistent root, e.g., the root object of a Mneme file). Thus, creation performance is not the most crucial measure.

4.3 Saving Objects

Using data sets ranging in size up to 8 megabytes, we measured elapsed time to save objects to disk. It does not matter whether the objects are new or old, and the number of objects does not affect the time noticeably; the procedure is I/O bound and depends only on the total number of bytes written. The tests were performed using a 667 megabyte SCSI drive (DEC RZ56). There was some variation, as can be seen in Figure 4, but the average write speed was approximately 321 kilobytes per second,[6] which is nearly 90% of what the disk can accept (measured as 360 to 370 kilobytes per second). ESM achieved some higher rates in our tests (343 kilobytes per second on large files), but, because its object ids are 12 bytes long, compared to Mneme's 4 bytes, Mneme was faster in terms of objects per second. Since Mneme used most of the available bandwidth, its performance is good. If the "typical" object size is 40 bytes, then Mneme can save about 8,000 objects per second. Similar objects (ones containing the same number of pointers) would be larger in ESM—60 to 80 bytes—so ESM can save 4,000 to 6,000 equivalent objects per second.

---

[6] We use *kilo* to mean 1000 and *mega* to mean 1,000,000, reserving K for 1024 and M for 1024 × 1024.

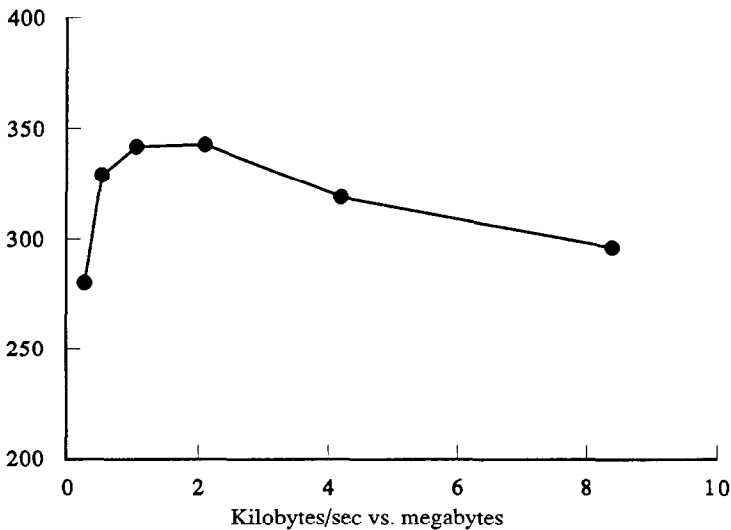Fig. 4.   Writing time.

4.4 Fetching Objects

To test retrieval time, we performed simple traversal of the trees, faulting objects as necessary but doing minimal work on the objects. Here the results varied a little, depending on whether we used the pointer interface or the handle (call) interface, but when fetching 2 megabytes or more Mneme is I/O bound and retrieves 279 kilobytes per second, as shown in Figure 5. For a "typical" object size of 40 bytes, Mneme fetches almost 7,000 objects per second. This is close to our original performance goal, but could stand improvement. ESM achieved between 220 and 230 kilobytes per second.

   Tests of sequential reads of large files indicate the disk can deliver 600 to 700 kilobytes per second, more than double what we actually achieve, and three times ESM. Clearly, Mneme is not always doing sequential reads. The rate at which the disk delivers data when the blocks of a large file are read in *reverse order* (end of the file to the beginning of the file) is 289 kilobytes per second, indicating that disk read order may be what is causing the difference in performance, rather than CPU overhead. Mneme achieves 97% of the reverse read rate. Knowing which segments to prefetch, clustering them together, and reading them all sequentially might boost the object read speed, but to do much better than we already have probably depends strongly on the application; but this certainly deserves deeper investigation.

4.5 Locating and Manipulating Resident Objects

To measure the cost of manipulating objects, we had to devise some manipulations to use as a basis for comparison. We came up with six manipulations: reading and updating slots, bytes, and integers. By *integer* we mean an aligned 4-byte quantity in the bytes part of an object, treated as an integer. Each of the
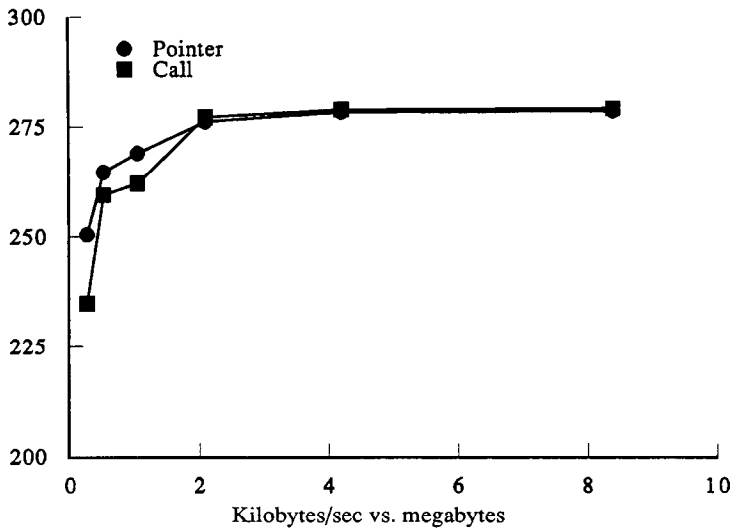
Fig. 5. Reading time.

manipulations performed a calculation, as indicated in this table:

Operation	Item	Calculation
Reading	Slot	count null pointers
	Byte	sum of bytes
	Integer	sum of integers
Updating	Slot	swap even-odd pairs of slots
	Byte	negate each byte
	Integer	negate each integer

Note that repetition over multiple objects in these tests is accomplished by recursive tree traversal, so the overhead of the traversal is included in all tests. Here are the measured per-object times (μsec) for traversal with *no* calculation, for comparison:

Software	Time
Nonpersistent C	11.7
Mneme, pointer	20.2
Mneme, handle	22.2

From this table we can see that resident object location using the MnIdPtr operation takes about 8.5 μsec, and constructing a handle in MnHandleCreate takes another 2.0 μsec. Our original goal was to support at least 100,000 object field accesses per second. We can see that Mneme can just barely do that if each field access requires an object lookup (8.5 μsec, plus a little time to get the field), but if multiple fields are accessed per lookup, as seems likely, then it will not be a problem. ESM took about 66 μsec per object for traversal. That its pointers (ids) are three times larger than Mneme's probably has something to do with the cost of this operation.

Now let us consider actual calculations, as opposed to simple traversal. The table below indicates incremental per-item costs for each calculation, in μsecs:

Software	Reading			Updating		
	Slot	Byte	Integer	Slot	Byte	Integer
Nonpersistent C	0.41	0.20	0.32	0.44	0.51	0.52
Mneme, pointer	0.42	0.20	0.33	0.44	0.51	0.50
Mneme, handle	0.83	0.25	0.51	1.06	0.62	0.72

It is natural that the Mneme pointer interface gives performance equivalent to C, since once one has acquired a pointer to the object in the buffer, the manipulations are exactly the same. The handle interface is slower for several reasons: there must always be a procedure call; the request is checked to make sure it does not exceed the bounds of the object; ids in slots are always converted from external to internal format; and updates require reading information into a scratch buffer, modifying data in the buffer, and then copying the buffer back into the object. ESM allows direct access for reads, so its read performance is also equivalent to C. However, ESM updates require that data be prepared in a separate buffer and block-transferred into the object, so updates are more costly, and in fact noticeably more expensive than Mneme handle updates.

4.6 Summary of Results

Mneme offers good performance: the pointer interface allows object manipulation at the speed of the application language; reading and writing utilize most of the available disk I/O bandwidth; and performance is better than the Exodus storage manager. Object creation and resident object location and manipulation appear fast enough to satisfy our original performance goals; reading and writing appear a little slower than desired, but are close to hardware limits, though since the disk can deliver considerably more bytes per second when read sequentially, techniques to obtain more of that bandwidth should be explored.

5. 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), persistent/database programming languages, and even object stores and database extensions of virtual memory. In the interests of brevity and incisiveness, we relate the Mneme design (not the prototype) to well-known exemplars of the various approaches rather than attempting exhaustive enumeration of all related work.

5.1 Extensible Databases

POSTGRES [42], Exodus [10, 18, 37], and Genesis [7] 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 fundamentally that of a relational database system.

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, but usually general routines composed together do not perform as well as a hand-coded implementation.

Exodus takes more of a language approach to extensibility. Its architecture includes the E language, an extension of C++ [43]. Of the extensible database systems, Exodus is the one most similar to Mneme, but the fairest comparison is 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. The Exodus storage manager prototype used in our tests supports buffer management but not concurrency, crash, recovery, or distribution.

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, and its large object identifiers and higher overhead interface make it less efficient as a platform for our persistent programming languages.

5.2 Object-Oriented Databases

Some relevant object-oriented database systems are Orion [25], GemStone [36], and VBase [3]. 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. More importantly, Orion and GemStone are not designed to be as easily extended as Mneme is through its pool strategies, forwarding protocols, and so on. Also, these other systems are all attempts at complete database management systems, and end up being "heavyweight" compared to Mneme.

5.3 Persistent and Database Programming Languages

PS-Algol [4, 5] is representative of the persistent programming languages, and E (already mentioned), O_2 [6, 27], O++ [1, 2], 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 supporting a large, distributed, unbounded object space.

5.4 Object Stores and Persistent Memories

Related work in this area could be stretched to include Multics [34] and a variety of capability architectures. More narrowly construed, some interesting examples are Camelot [40], 801 [11], FAD [12], ObServer [20, 39], and CACTIS [21]. Camelot and 801 offer different versions 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. The 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), and so on. FAD has been similarly implemented, although it does include some limited object semantics. The 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 object spaces. The Mneme design has placed more emphasis on performance and on scaling to large distributed systems. While the developers of other systems might rightly claim that they could extend them to large distributed systems, the Mneme design addresses important issues that they have not addressed. In particular, our file concept, and the related ideas of forwarders and of mapping between persistent and client identifiers for objects, tackle the problems of scale head on. It will be easier for us to extend our prototype, since we have a design that addresses scale, autonomy, extensibility, and other issues of 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 oriented towards "active" data, and on how to perform derivation most efficiently when slots are updated. Mneme might provide an interesting substrate on which to build CACTIS or other derived/active data semantics.

5.5 Summary of Related Work

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

—supporting multiple programming languages;

—providing (storage) objects with structure and identity, but "no frills";

—emphasizing performance;

—addressing issues of autonomy and large distributed systems; and

—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 performance results indicate our emphasis on efficiency; and we have already used the policy extensibility to vary the cluster (segment) sizes in some of our tests. The goals met distinguish Mneme from other systems. Further, the current work will

extend readily to distributed systems, since we have been designing for distribution from the start.

6. CONCLUSIONS

Our approach to providing object semantics and high performance for CAD and related information-intensive applications is to build a simple, "lightweight" 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 aims: 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. The principal achievement is the combination of object functionality (identity and structure) and performance.

To gain perspective on the design and implementation ideas that have been developed in the project so far, it is instructive to consider how the original goals are supported and which features of the design (or implementation strategy) address which individual goals. We consider a series of goals in turn:

—*Persistent programming.* Efficient manipulation of resident objects is supported by direct pointer access to objects in Mneme buffers, by having object identifiers about the same size as a machine pointer, by the fast object lookup mechanism, by allowance for tagging in objects, and by provision for garbage collection and automatic storage management. The short object ids, tagging, and orientation towards automatic storage management also make it easier to support persistence transparently in a programming language.

—*Distribution.* The Mneme file concept, with forwarders, the incoming reference table, and root objects address a number of distributed systems issues, particularly autonomy and security, but also issues of scale. These same features also address the goal of supporting identifiable and separable subcollections of objects.

—*Extensibility.* The Mneme pool concept is present primarily for policy extensibility (although it can also be used just to group related objects). Forwarders and incoming reference tables also provide important support for semantic and policy extensibility, as do the file, pool, and object attributes. The idea of specifying a server interface also supports extensibility. Finally, providing simple and general semantics rather than building in more sophisticated features (e.g., data modeling primitives) supports extensibility by not preempting higher level approaches.

—*Performance.* Overall simplicity appears to aid performance, although one can probably always find *some* higher level feature that must be built in if one is to achieve the best possible performance. More specifically, though, Mneme attacks performance problems by addressing the issue of per-object costs, with the philosophy that any work that must be done for each object manipulated will noticeably affect performance. Hence, Mneme applies grouping and clustering in a number of ways to improve performance, most notably in the physical segment (clustering) and logical segment concepts.

—*Portability and usability.* This has been addressed by careful interface design applying principles such as uniformity, and by using a readily available and widely implemented language (C) in which to build the prototype. We have also taken care to avoid hardware-, operating system-, and compiler-specific features.

Taking a broad view, the conceptual trade-offs in design and implementation are interesting in that Mneme's performance, portability, and genericity come as much from what is *not* provided as from what is. Applications may indeed require additional features and semantics, but Mneme supports exploration of a wide range of tasks by 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 promises to provide good support for many emerging applications. Mneme's portability and comparative simplicity will make it attractive to researchers and others who are interested primarily in the applications above, rather than the bytes below. The system has already been welcomed by two software engineering research groups and others are very interested. 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 on distributed-execution models that overcome the limitations of distributed systems which are based only on a shared store.

REFERENCES

1. AGRAWAL, R., AND GEHANI, N. H.   ODE (Object Database and Environment): The language and the data model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Portland, Ore., May–June 1989). *ACM SIGMOD Rec. 18*, 2 (1989), 36–45.

2. AGRAWAL, R., AND GEHANI, N. H.   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* (Gleneden Beach, Ore., June 1989). Morgan Kaufman, 1989, 25–40.

3. ANDREWS, T., AND HARRIS, C.   Combining language and database advances in an object-oriented development environment. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, Fla., Oct. 1987). *ACM SIGPLAN Not.* (1987), 430–440.

4. ATKINSON, M. P., CHISOLM, K. J., AND COCKSHOTT, W. P.   PS-Algol: An Algol with a persistent heap. *ACM SIGPLAN Not. 17*, 7 (July 1982), 24–31.

5. ATKINSON, M. P., AND MORRISON, R.   Procedures as persistent data objects. *ACM Trans. Programm. Lang. Syst. 7*, 4 (Oct. 1985), 539–559.

6. BANCILHON, F., BARBEDETTE, G., BENZAKEN, V., DELOBEL, C., GAMERMAN, S., LECLUSE, C., PFEFFER, P., RICHARD, P., AND VELEZ, F.   The design and implementation of O_2, an object-oriented database system. In *Advances in Object-Oriented Database Systems* (Sept. 1988), K. R. Dittrich, Ed., *Lecture Notes in Computer Science, 334*, Springer, New York, 1988, 1–22.

7. BATORY, D. S., BARNETT, J. R., GARZA, J. F., SMITH, K. P., TSUKUDA, K., TWITCHELL, B. C., AND WISE, T. E.   GENESIS: An extensible database management system. *IEEE Trans. Softw. Eng.* (Nov. 1988), 1711–1730.

8. BISHOP, P. B.   *Computer systems with a very large address space and garbage collection.* Ph.D. thesis, MIT, Cambridge, Mass., May 1977.

9. CARDELLI, L., DONAHUE, J., GLASSMAN, L., JORDAN, M., KALSOW, B., AND NELSON, G.   Modula-3 report (revised). Tech. Rep. 52, Digital Equipment Corp. Systems Research Center, Palo Alto, Calif., 1989.

10. CAREY, M. J., DEWITT, D. J., RICHARDSON, J. E., AND SHEKITA, E. J.   Object and file management in the EXODUS extensible database system. In *Proceedings of the Twelfth International Conference on Very Large Data Bases* (Kyoto, Sept. 1986). ACM, New York, 1986, 91–100.

11. CHANG, A., AND MERGEN, M. F.   801 storage: Architecture and programming. *ACM Trans. Comput. Syst. 6*, 1 (Feb. 1988), 28–50.

12. COPELAND, G., FRANKLIN, M., AND WEIKUM, G.   Uniform object management. MCC Tech. Rep. ACA-ST-411-88, Microelectronics and Computer Technology Corp., Austin, Tex., Dec. 1988.

13. COPELAND, G., AND MAIER, D.   Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, Mass., June 1984). *ACM SIGMOD Rec. 14*, 2 (1984), 316–325.

14. DEWITT, D. J., FUTTERSACK, P., MAIER, D., AND VELEZ, F.   A study of three alternative workstation-server architectures for object oriented database systems. In *Proceedings of the Sixteenth International Conference of Very Large Data Bases* (Brisbane, Australia, Aug. 1990). Morgan Kaufman, 1990, 107–121.

15. ELHARD, K., AND BAYER, R.   A database cache for high performance and fast restart in database systems. *ACM Trans. Database Syst. 9*, 4 (Dec. 1984), 503–525.

16. ESWAREN, K. P., GRAY, J. N., LORIE, R. A., AND TRAIGER, I. L.   The notion of consistency and predicate locks in a database system. *Commun. ACM 19*, 11 (Nov. 1976), 624–633.

17. GOLDBERG, A., AND ROBSON, D.   *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, Reading, Mass., 1983.

18. GRAEFE, G., AND DEWITT, D. J.   The EXODUS optimizer generator. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, May 1987). *ACM SIGMOD Rec. 16*, 3 (1987), 160–172.

19. GRAY, J. N.   Notes on database operating systems. In *Operating Systems: An Advanced Course*, R. Bayer et al., Eds., *Lecture Notes in Computer Science*. Springer, New York, 1978.

20. HORNICK, M. F., AND ZDONIK, S. B.   A shared, segmented memory system for an object-oriented database. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 70–95.

21. HUDSON, S., AND KING, R. CACTIS: A database system for specifying functionally-defined data. In *Proceedings of the Workshop on Object-Oriented Databases* (Pacific Grove, Calif., Sept. 1986). ACM, New York, 1986, 26–37.

22. INTEL CORP. *Introduction to the iAPX 432 Architecture, Manual 171821-001.* Intel Corp., Santa Clara, Calif., 1981.

23. KERNIGHAN, B. W., AND RITCHIE, D. M. *The C Programming Language.* Prentice-Hall, Englewood Cliffs, N.J., 1978.

24. KHOSHAFIAN, S. N., AND COPELAND, G. P. Object identity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Ore., Sept. 1986). *ACM SIGPLAN Not. 21*, 11 (1986), 406–416.

25. KIM, W., BALLOU, N., BANERJEE, J., CHOU, H., GARZA, J. F., AND WOELK, D. 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, Calif., Nov. 1988). *ACM SIGPLAN Not. 23*, 11 (1988), 142–152.

26. KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Trans. Database Syst. 9*, 4 (June 1981), 213–226.

27. LECLUSE, C., RICHARD, P., AND VELEZ, F. O_2, an object-oriented data model. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Chicago, Ill., Sept. 1988). *ACM SIGMOD Rec. 17*, 3 (1988), 424–433.

28. LISKOV, B., ATKINSON, R., BLOOM, T., MOSS, E., SCHAFFERT, C., SCHEIFLER, R., AND SNYDER, A. *CLU Reference Manual.* Springer, New York, 1981.

29. LISKOV, B., SNYDER, A., ATKINSON, R., AND SCHAFFERT, C. Abstraction mechanisms in CLU. *Commun. ACM 20*, 8 (Aug. 1977), 564–576.

30. MOSS, J. E. B. Addressing large distributed collections of persistent objects: The Mneme project's approach. In *Second International Workshop on Database Programming Languages* (Gleneden Beach, Ore., June 1989). Morgan Kaufman, 1989, 269–285.

31. MOSS, J. E. B. Working with persistent objects: To swizzle or not to swizzle. COINS Tech. Rep. TR 90-38, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, May 1990. Submitted for publication.

32. MOSS, J. E. B., GRIFFETH, N. D., AND GRAHAM, M. H. Abstraction in concurrency control and recovery. COINS Tech. Rep. 86-20, Dept. of Computer and Information Science, Univ. of Massachusetts, Amherst, May 1986.

33. MOSS, J. E. B., LEBAN, B., AND CHRYSANTHIS, P. K. Finer-grained concurrency control for the database cache. In *Proceedings of the Third International Conference on Data Engineering* (Los Angeles, Feb. 1987). IEEE, New York, 1987, 96–103.

34. ORGANICK, E. I. *The Multics System: An Examination of Its Structure.* MIT Press, Cambridge, Mass., 1972.

35. ORGANICK, E. I. *A Programmer's View of the Intel 432.* McGraw-Hill, New York, 1983.

36. PURDY, A., SCHUCHARDT, B., AND MAIER, D. Integrating an object server with other worlds. *ACM Trans. Office Inf. Syst. 5*, 1 (Jan. 1987), 27–47.

37. RICHARDSON, J. E., AND CAREY, M. J. Programming constructs for database system implementations in EXODUS. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (San Francisco, May 1987). *ACM SIGMOD Rec. 16*, 3 (1987), 208–219.

38. SCHAFFERT, C., COOPER, T., BULLIS, B., KILIAN, M., AND WILPOLT, C. An introduction to Trellis/Owl. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, Ore., Sept. 1986). *ACM SIGPLAN Not. 21*, 11 (1986), 9–16.

39. SKARRA, A., ZDONIK, S. B., AND REISS, S. P. An object server for an object-oriented database system. In *Proceedings of International Workshop on Object-Oriented Database Systems* (Pacific Grove, Calif., Sept. 1987). ACM, New York, 1987, 196–204.

40. SPECTOR, A. Z., BLOCH, J. J., DANIELS, D. S., DRAVES, R. P., DUCHAMP, D., EPPINGER, J. L., MENEES, S. G., AND THOMPSON, D. S. The Camelot project. Tech. Rep. CMU-CS-86-166, Dept. of Computer Science, Carnegie-Mellon Univ., Pittsburgh, Pa., 1986.

41. STONEBRAKER, M. The design of the POSTGRES storage system. In *Proceedings of the Thirteenth International Conference on Very Large Data Bases* (Brighton, England, Sept. 1987), Morgan Kaufmann, 289–300.

42. STONEBRAKER, M., AND ROWE, L. A.   The design of POSTGRES. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Washington, D.C., May 1986). *ACM SIGMOD Rec.* (1986), 340–355.

43. STROUSTRUP, B. *The C++ Programming Language.* Addison-Wesley, 1986.

44. WEIHL, W., AND LISKOV, B.   Implementation of resilient, atomic data types. *ACM Trans. Programm. Lang. Syst. 7*, 2 (Apr. 1985), 244–269.