Addressing Large Distributed Collections of Persistent Objects: The Mneme Project's Approach*

J. Eliot B. Moss †

COINS Technical Report 89-68
June 1989

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

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

†Address: Department of Computer and Information Science, Lederle Graduate Research Center, University of Massachuetts, Amherst, MA 01003; telephone: (413) 545-4206; Internet address: Moss@CS.UMass.Edu.

Abstract

The Mneme persistent object store project has as one of its primary goals to support cooperative, concurrent, and reliable use of large, distributed collections of objects. In our case, distribution is intended to mean not only physical dispersion, but also some autonomy or independence of subcollections of objects—that is, the object space is under some degree of decentralized management. Providing independence of subcollections of objects has interesting implications as to how to address and retrieve objects efficiently. Here we describe the need for and benefits of independent subcollections of objects, examine the addressing implications, and show how these implications have affected the design of Mneme. Most particularly, we argue against the appropriateness of a large flat store of bytes or even of object identifiers, and in favor of richer, more flexible, structures. We also contend that comparable performance can be achieved by careful implementation of the richer structures, with considerably more functionality and flexibility than flat structures provide. The cost is increased complexity of the supporting software.

This paper will appear in the proceedings of the Second International Workshop on Database Programming Languages, Gleneden Beach, OR, June 1989, being published by Morgan-Kaufmann. This version has the same text, but is formatted differently.

1 The problem

We wish to consider the problem of addressing and manipulating objects in an essentially unbounded distributed space of persistent objects. The objective is to understand how to implement a large distributed virtual object space, on top of which an object-oriented database programming language (DBPL) might be built. The general style of object we have in mind is roughly that of Smalltalk—a self-contained vector of fields, many of which are references to other objects. The problem is difficult because all of these conditions hold simultaneously:

- We desire good performance, in both space and time.
- There are very many objects.
- The objects are spread around a decentralized distributed system.
- Most of the objects are small.
- Objects may persist between program executions.

The first two conditions give rise to a series of arguments about addressing within a large space of objects. As will be seen, we come to the possibly controversial conclusion that rather than using long, globally unique ids or a large "flat" virtual address space, we should use short addresses, assigned and interpreted in context. The third condition raises issues of autonomy and independence, which reinforce part of our proposed design approach, namely that the object space should be structured into disjoint localities of objects under somewhat separate management. The fourth condition motivates some of the details of translation, specifically, managing objects and object addresses in groups to reduce per-object overheads. The last condition explicitly makes the point that we must manage permanent memory resources as well as those of running processes. Further, since persistence implies that at least some objects will be long-lived, persistence demands flexibility to reorganize and reallocate resources as patterns of use shift over time.

1.1 The system model

For clarity, we spell out our model of the distributed system context in which the object store is to reside. The system consists of a number of *client processes*, running on a distributed collection of computers, supported by a number of *server machines*, loosely confederated to provide the large distributed persistent object store. We are not directly concerned with the means and cost of communication, but wish the scheme to work well across a local area network. New clients and new servers can be added freely, and we desire the object naming and access mechanisms to scale as the store grows to quite large sizes.

1.2 Mneme

Mneme (the Greek word for *memory*) is the name of the persistent object store system under development at the University of Massachusetts. Its goals subsume the goals of the system discussed in this paper. Mneme is additionally concerned with issues such as supporting multiple languages, making use of more than one kind of back end storage server, and providing support for multiple object management strategies and policy extension. A single user, non-distributed prototype has been running since September 1988. Further information on Mneme, its goals and concepts, and implementation strategies taken in the first prototype can be found in [Moss and Sinofsky, 1988].

1.3 The opposing view

Our position is that relatively short, contextual addresses for objects will work best, on cost and functionality grounds. The opposing view is that long, non-contextual (global) addresses are better. In this view, every object would have a globally unique object identifier. Such an identifier is most easily envisioned as a fixed length bit string, i.e., a logical (but not physical) pointer to the object it identifies. An id does not directly encode the object's location (though it may contain a hint), so there must be additional mapping information. Such names are thus *location independent* [Khoshafian and Copeland, 1986]. If we are willing to give up some location independence, we can reduce the mapping overhead by tying the name (id) to the location of the object within a large shared virtual address space. Since there have been serious proposals in recent years to build databases within large virtual address spaces, we consider the issues in detail below. Many of our arguments have to do with the size of the addresses. These arguments mostly apply to flat spaces of unique identifiers as well, since the sizes of global unique ids and global virtual addresses are similar. We consider global ids after global virtual addresses. After presenting our arguments against large flat address spaces, we present the relevant aspects of the Mneme design, indicating how it meets our goals.

1.4 Size of the address space required

Before considering the arguments for and against long addresses, let us make some (necessarily rough) estimates as to just how long they may need to be. Clearly, the application environment and needs of the organization using the system have a lot to say. Since we desire a system that scales well, we consider moderately large collections of data, though perhaps not the largest collections that could be envisioned. We should also be generous, since memories and address spaces tend to grow with time as technology makes it feasible to have more memory within a system.

First, how many *objects* might we wish to address? Within a complex engineering design (e.g., the space shuttle or a jumbo jet) there will be millions of components, many drawings,

revisions, notes, documents, etc. Consider the problem of designing and maintaining a fleet of jumbo jets. In general we need a record of all the components in *each* individual plane, for maintenance records, and all of this must be kept available (possibly in archival storage), as well as the complete history of repairs, revisions, etc. With a few hundred planes, each with on the order of a million parts, and hundreds of hours of maintenance per year, it is easy to run into something like billions of objects online and billions, perhaps trillions, of archival records over the life time of the product. There are likely into the millions of pages of documents produced, some of them textual, many with graphics. Clearly we will need well over the currently typical 32 bits.

Assuming that address lengths, like word sizes, are most convenient when they are powers of two, we should use 64 bit addresses, or possibly even 128 bits. For concreteness, we will frame our arguments as 32 versus 64 bits, but the principles are the same for other sizes. We also note that 64 bits is probably adequate as either the virtual address size or the size of globally unique ids.

At this point, a reasonable question to ask is: If we favor 32 bits over 64, would not the same arguments have led us to retaining 16 bits rather than 32 in the struggle not so many years ago that led to broad abandonment of 16 bit address spaces as being too small? The answer is yes, the cost argument by themselves would lead to the smaller address space. The balancing factor is how many objects a program needs to name and access directly for the program to run efficiently (i.e., not spend too much effort managing its name space). It is clear that 2^{16} is simply too small, and we admit that for some applications 2^{32} is a bit restrictive, especially since multiple gigabyte main memories will shortly be feasible. Still, as we hope will become clear, it may be reasonable to use short addresses from within programs, even if they turn into longer addresses for actual memory accesses.

2 Why not use a large flat virtual address space?

First, let us review the purported advantages of a large flat address space. They are easy to enumerate:

- System software is relatively simple and well understood, being essentially a virtual memory paging mechanism, perhaps augmented with locking, logging, and special fetch and replacement policies for some parts of the address space.
- User software is simplified in that it deals only with ordinary virtual memory pointers. This speeds and simplifies object and search structure traversal, avoids format conversions, and reduces the amount of object copying required, as noted in [Copeland *et al.*, 1988].
- Hardware support is well understood and easy to justify.

Systems that have taken the large virtual memory approach include the Intel 432 [Organick, 1983; Intel Corporation, 1981], the IBM RT [Chang and Mergen, 1988], the

Bubba project [Copeland *et al.*, 1988], CPOMS and related systems [Brown and Cockshott, 1986; Cockshott, 1987; Connor *et al.*, 1989], and MONADS [Keedy and Rosenberg, 1989]. [Chang and Mergen, 1988] and [Copeland *et al.*, 1988] deal more especially and completely with problems of concurrent access and reliability (transaction support). In particular, past objections to flat stores that have been addressed with some success include concurrency control, recovery, and buffer management.

While a flat address space is undoubtedly appealing because of its simplicity and apparent efficiency, the approach has a number of problems, which we feel are severe enough to justify abandoning the approach for decentralized systems of the scale we envision. Here is a list of the difficulties we find with large flat address spaces:

- The large number of address bits required have a negative impact on the price-performance of a system, in its cpu as well as its memory hierarchy.
- Managing a large persistent virtual address space presents complications that operating system virtual memory managers have not usually addressed.
- The flat structure is inappropriate for a decentralized system because it interferes with autonomy. It also interferes with efficient, decentralized resource management (e.g., garbage collection).

2.1 The price-performance argument

Address size has a variety of impacts on the price-performance of a computer, which we enumerate as completely as we can here. Note that we are concerned with *virtual* address size, not physical address size. Hence we are not concerned with the width of the physical memory address bus, the number of physical memory address pins on chips, and the like. Let us assume that we are comparing two address sizes, *wide* and *narrow* addresses. For concreteness we can think of these as 64 bits and 32 bits respectively, but the same principle applies regardless of the absolute size of the addresses.

A global effect that the virtual address size tends to have is that the machine's natural word size will be at least as big as the virtual address size, so that pointers (virtual addresses) will fit in a word. This will almost certainly be true of a RISC machine. Thus, while the virtual address size need not have a direct impact on the physical memory address bus width, it will tend to impact the physical memory *data* bus width.

2.1.1 Impacts on the CPU

Wide virtual addresses require wide registers to hold them. These wide registers not only require chip area proportional to the width of the registers, they also require wider buses and more pins on the chip to move data to and from memory, etc. This impact is largely proportional to the width, and mainly requires the chip be bigger, giving lower

yield and higher power consumption for the same technology and number registers. Thus the chip will cost more but not offer increased functionality beyond the wider values.

Worse, the ALU, shifters, and other manipulative components will not only be bigger, but some of them may be slower. For example, addition will take longer with larger numbers given the same technology and degree of carry look ahead logic. For a rough estimate, we could model addition times as a fixed time plus a constant times the logarithm of the word length (i.e., one extra level of carry look ahead logic for each doubling in the word length), i.e., $c_1 + c_2 \log_2 w$ where w is the width in bits. The worst case would be small c_1 and large c_2, and doubling from 32 to 64 bits would give a ratio of addition times equal to $\frac{6}{5} = 1.2$. Thus the slow down (under this simplistic model) is no worse than 20%. Still, even a 5% or 10% slow down cannot be justified unless one really needs the wider words.

At this point we should mention that an independent justification for wide words would be a need to handle many large integers or to offer higher precision floating point numbers. This is perhaps why most supercomputers have relatively wide words (60 or more bits are typical). We note, though, that the community of concern appears largely satisfied with the properties of 32 bit machines as far as arithmetic goes—the concern is about offering better object and database technology, not higher precision numbers.

How much bigger will the wide address chip be than the narrow address one? A simplified model of the area of the cpu chip might be $a_1 + a_2 \log_2 w + a_3 w$, namely, there is a constant part (e.g., the control section), a part that grows logarithmically with width (perhaps a shifter control, carry look ahead, etc.), and a part that grows directly with the width (registers, buses, ALU, and so forth). In the worst case, the wide address chip would be nearly twice the size of the narrow address chip. Let us estimate the ratio of sizes as 1.5.

What is the ratio of the costs? There will be a fixed part and a part proportional to the chip area, to produce a chip. This must be reduced according to the number of faulty chips produced—that is, we need to estimate the difference in yield. Again, taking a very simple model in which the number of flaws is proportional to the area, we would like to estimate the yield of the wide address chip, y_w, given the yield of the smaller one, y_n. Given the average number of faults per unit area, f, and the total area A, we can estimate the yield (the probability that a chip of area A will have no faults) as $\exp(-fA)$. Thus, with this very simple model, we estimate y_w as $\exp(-fA_w)$, which is $\exp -fA_n(1.5)$, so $y_w = y_n^{1.5}$. For high yields this may not be very bad. For lower ones, it could be quite substantial.

2.1.2 Impacts on cache memory

Doubling the size of the addresses will double the cost of data part of a cache to hold the same of addressable units (words), since the address size doubles, and, as argued above, the data size will double, too. If we reduce the size of the cache to compensate for the increased

cost, we will reduce overall performance since we will take more cache misses. On the other hand, since we can probably encode instructions into about the same number of bits in either the wide or narrow scheme, we do not necessarily need to double the cache size in bits to get the same performance in wide scheme as in the narrow scheme. The desire for simple RISC architectures may indicate against these savings, i.e., to maintain instruction execution performance we might need to waste a lot of the additional instruction bits. Assuming that about half the cache cycles are for instructions, the wide scheme would use between 75% and 100% as many cache cycles as the narrow one. It seems unlikely that there would be a net price-performance increase with the wide architecture, though, since it entails more complex logic to unpack instructions, saves no data references, and requires at least 50% more cache bits to hold the same volume of useful data (assuming the cache is half instructions and half data). It would be interesting to do more detailed studies of these issues.

2.1.3 Impacts on main memory

Except perhaps for instructions, character strings, and bit maps, the wide architecture doubles the size of data, both pointers and numbers. Even code, strings, and bit maps incur some penalty because they are generally padded out to the next complete word. The padding overhead can be estimated as half a word per entity (code sequence, string, etc.). Without measurements of object sizes and distributions, we cannot be precise about these effects. In a system that is dominated by code, strings, and the like, the wide architecture would incur little space overhead. In a system dominated by pointers and numbers, the space required would approximately double. In the absence of more detailed data, which is clearly application specific anyway, we conclude that we need on the order of 50% more bits of main memory for the wide architecture. We take this as an estimate also of the average increased size of objects, in bits, under the wide scheme.

Even if somehow it is only addresses (pointers) that are widened there is still likely to be a considerable increase in size. We believe this because of our experience with various heap-oriented programming languages, such as Clu [Liskov *et al.*, 1977; Liskov *et al.*, 1981], LISP [Steele Jr., 1984; McCarthy *et al.*, 1984], Trellis/Owl [Schaffert *et al.*, 1986], and Smalltalk [Goldberg and Robson, 1983]. We feel these languages are more representative of future programming concerning object size and use of pointers than are traditional (e.g., relational) databases, where pointers are rare. In heap oriented languages, pointers seem to comprise 30% to 50% or more of the data in many programs. We note as an aside that a database might actually shrink significantly if recast into a heap oriented language, since many string keys would be replaced by shorter object ids or addresses, referring directly to their target objects. The frequency of occurrence of pointers, etc., cannot be resolved until representative applications are built and measured.

2.1.4 Impacts on secondary storage

Since objects are estimated to the 50% (between a few percent and 100%) bigger, secondary storage must grow proportionally. The size has a direct impact on cost, but it also has an impact on performance. 50% more bits must be transferred for each object. Worse, objects will be more spread out, incurring increased secondary storage positioning time (in typical modern devices, such magnetic and video disks and CDs).

2.1.5 Impacts on distributed system performance

Since objects are larger and words longer in the wide architecture, more bits need to be transferred per object transmitted. Given fixed bandwidth (e.g., same number of systems on an Ethernet), there will be more average delay before transmission can start, too.

2.1.6 Conclusions about costs

We have seen that doubling the word size of an architecture while holding other aspects fixed has notable impacts on the cost and performance of the system, and virtually all these impacts are negative. Thus, we do not have a time-space tradeoff, *unless* the additional address space really makes programs more efficient by obviating their need to multiplex their address space, and the performance increase of avoiding application address space management outweighs the loss of price-performance incurred by the wider word size.

2.2 Complexity and cost of a large virtual address space

Extending virtual memory management techniques suitable for a 32 bit address space to a 64 bit address space may have some problems. The sheer size of the space makes it harder and more complex to manage in a number of ways. Here are some of the problems:

- How do we allocate and free virtual memory? The data structures necessary for the bookkeeping can get to be quite large. Maintaining them efficiently and robustly will not be simple. There is an analogous problem in allocating and freeing secondary storage, except perhaps that the problem is worse since it has more direct and substantial impacts on performance.

- How do we page efficiently? The allocation algorithms presumably builds and maintain map information about where virtual memory resides on secondary storage. The sheer size of the mapping information itself is a problem. We will need to find ways to cache it intelligently or else page faults will tend to require two or more independent secondary storage accesses: one or more to access the mapping information, and then one to fetch the page.

The point we are making is that any system supporting a large address space must deal with the issues of scale that have not been tackled by most operating systems to date. It

is glib to claim that going from 32 to 64 bits is trivial. We feel that some of the mapping and address space management techniques we have developed for Mneme could apply to flat address spaces, too. this does not reduce the force of our other arguments against flat space, though.

2.3 Problems of autonomy, flexibility, and efficiency

In fairness, most of the proposals for large virtual address spaces have been for *centralized* systems. We are concerned with large distributed systems and must allow for considerable *autonomy*, that is, independent management of resources (processors, memories, disks, etc.) in the system. The size of the address space is not the problem here. Rather, it is the lack of structure—the fact that the space is flat. At the very least we would need to introduce some kind of protection mechanism and check access to pages according to client and application. In addition to access to resources, autonomy influences their allocation and reclamation.

The flat space is also likely to be overly *automatic*, to have its access policy built considerably into the hardware of the system. Thus, it is likely to be inflexible. The flat address space is not conducive to hardware *heterogeneity*. It is also not conducive to heterogeneity of object format, and, more important, to variety in object management policies. We believe that it is important to be able to identify interesting subcollections of objects and to specify the policy to be used in managing a subcollection. This need has been recognized by database system implementors, presented as a complaint about the inappropriateness of common operating system virtual memory management policies for database page management. More modern systems support some policy variation, for example, [Copeland *et al.*, 1988] describes a system with two policies.

2.3.1 Advantages of a structured address space

The easiest way to add to this argument is to point out the advantages of dividing the address space into separately managed *localities*, where each locality has considerable independence regarding the allocation and reclamation of objects in its locality, the details of locating and buffering objects, of object formats, and of concurrency control and resiliency. Here are some of these advantages:

- Localities can use management policies suited to the objects they contain. The only restriction is that policies cannot vary so much as to cause unresolvable conflicts between localities, since we assume that clients can use objects from more than one locality at the same time. For example, *arbitrary* mixture of concurrency control policies cannot be supported, since timestamping and locking may give different serialization orders, but optimistic and pessimistic techniques can be mixed. Not only is policy variation nice to have, it is *essential* to be able to add and tune policies to applications in order to achieve acceptable performance.

- Localities help organize the data into independently useful pieces for the users and for applications. Thus, localities may allow simple and reasonable techniques to extract meaningful subsets of data to share with other users, to copy, to replicate, etc. A flat space makes it difficult to locate and extract a meaningful subset of data, insert it into a similar space elsewhere, and then use it meaningfully.

- As argued in [Bishop, 1977], localities are *essential* for effective garbage collection in very large address spaces[1] The alternative is to impose ad hoc, unenforced, and probably unenforceable rules (i.e., programming conventions) about where to insert levels of indirection, to restrict the patterns of inter-object references (e.g., the data *must* form a tree in the system described by [Copeland *et al.*, 1988]), or to forego garbage collection—itself a problem since compacting garbage collection and the resulting reclustering and compaction of objects have significant performance advantages over time. We expand on this particular advantage of structured address spaces below.

- Localities make it easier to support a degree of autonomy for data, and concomitant access control. This can be thought of as part of "object management policy", but is distinct from the performance and correctness aspects previously mentioned.

- Localities more readily support heterogeneity of hardware, operating systems, storage formats, and communications protocols. A system incorporating the appropriate support for localities thus provides a more plausible migration path from current systems, and into existing organizations. While it is not a subject we are pursuing, a locality might manage to hide beneath itself a traditional database system and to provide object access to the data managed by that system.

2.3.2 Efficient resource management

Let us sum up why we believe a flat space is not conducive to efficient resource management. This is mainly because it tends to force too much commonality of policy and algorithm. For example, with a flat space, we probably need to decide in advance whether and how garbage collection will be performed, and state (and if possible, enforce) a number of conventions to make garbage collection possible and practical. In a similar vein, there are problems in choosing a page size to use throughout a large heterogeneous system, not just because of hardware variations, but even more because of differing characteristics of different objects. Because of the lack of policy variability, it is harder to take advantage of user or application knowledge to improve performance.

A flat space may also encourage centralized rather than decentralized management, e.g., of allocation of pieces of virtual address space, leading to bottlenecks and critical failure points in a decentralized system. Localities lead naturally to decentralized resource management and autonomy.

---

[1] Bishop called localities *areas*.

2.4 A detailed example: garbage collection

In order to make some of the above arguments in favor of structured rather than flat address spaces more concrete, we now discuss the issue of garbage collection in more detail. First, consider what is necessary in garbage collecting a large flat space. We may be able to locate pointers and distinguish them from non-pointer data, or we may not. If we cannot distinguish pointers from non-pointers, then we must use a *conservative* garbage collector: one that assumes every quantity is a pointer and thus retains all storage *apparently* referenced. Note that a conservative collector must not move objects directly referenced. This inability to compact could be a severe problem, since compaction and reorganization affects clustering and improves run-time performance, sometimes dramatically.

The worst problem we face, though, is the need to examine the *entire* address space to establish that an object is no longer referenced and can thus be reclaimed. Hence, all garbage collection is essentially global, a daunting and unrealistic approach in a large distributed environment. An explicit deletion scheme may be the only viable alternative, but it raises serious software reliability concerns (dangling references) and complicates software design. Explicit deletion has hidden performance penalties as well, particularly when compared with the better compacting garbage collection schemes. A compacting collector can use linear allocation, rather than a free list or other strategy, making allocation very efficient (quite readily done inline in a few instructions). We have already noted that compaction will tend to improve performance by improving clustering and locality. Finally, there is the overhead in design, the distortion of program structure, and, in the case of reference counting storage reclamation, higher run-time cost.

Let us turn to a space structured as a collection of independently managed localities. Following Bishop [Bishop, 1977], we distinguish between *intra-locality* and *inter-locality* references. Inter-locality references require special interpretation, which may be done by the referenced locality. If each locality maintains some sort of *incoming reference table* (IRT) then external references to objects in a locality can be controlled and filtered, and are insulated from rearrangement of the objects within the locality. That is, the IRT contains some kind of pointer to the object within the locality, and when the object moves, we simply update the reference in the IRT.

We can accomplish significant garbage collection independently, too. In this case, the IRT is used as a set of roots. A locality may also have a set of "natural" roots—objects that semantically "belong" in the locality. If an object is not reachable from either set of roots, it can be reclaimed. If an object is reachable only from the IRT (i.e., not from the natural roots), then it needs to be retained, but it is also a good candidate to move into another locality. Entries in the IRT can be marked with a count of the number of localities that have references to the object in question. As we garbage collect an area we can detect that it no longer refers to objects in other areas, and decrement the reference count in the IRT entries for those objects in the other areas. This work can almost certainly be

done offline, if some care is taken about the order in which things are processed. It is even possible to take unilateral action and reclaim an object, leaving behind a specially marked IRT (or somehow preventing use of the same object id again) so that a dangling reference can be detected and reported. If objects reachable only from the IRT and not the natural roots of a locality are moved to localities that refer to them, then global clustering will be improved. Further, cycles of garbage among localities will, over time, collapse until they are within a single locality, and then will be reclaimed at the next garbage collection of that locality.

The point is that we gain considerable autonomy: each locality has significant latitude in its management policies and algorithms. We also gain considerably in performance of garbage collection. This is because we can garbage collect pieces independently, we can garbage collect them more frequently, we can use policies appropriate to the locality, and we can get more machines operating in parallel. We will also have more smooth, incremental operation, rather than having long periods during which the system cannot be used. (Actually, it should be clear that stop-and-collect is completely unreasonable for systems of the scale we are considering.)

2.5 Conclusions about large flat virtual memory

We have presented rather detailed arguments as to why, on the grounds of price-performance, a narrow address space should be preferred over a wider one. We have also argued that a flat space, such as that offered by virtual memory, is less desirable than a structured space, on grounds of autonomy, flexibility, and efficiency, the efficiency coming from allowing more tailoring of policy to the situation at hand, and possibly from more decentralized management of resources.

3 Why not a large flat object id space?

The width and the autonomy/flexibility arguments largely apply to a large flat object id space as well as they do to a large flat virtual address space. An id space is different in that it offers more flexibility through increased location independence. We may be able to move objects, resize them, even garbage collect them, much better than we can manage a virtual address space. Still, the autonomy and flexibility arguments, as well as the price-performance arguments, apply. In fact, an object id space introduces the additional problem of locating objects within the physical resources of the system. If objects are typically smaller than pages, the overhead may be higher than in the virtual memory system. Because objects can move, change size, and so on, there may be an additional level of indirection, incurring both time and space costs. These costs it may be better to bear, though, because the flexibility is valuable. This is an instance where more detailed measurements of the overheads of the indirection will be necessary to resolve the issue.

4 What about long but structured addresses?

Suppose we attempt to gain the flexibility and autonomy we desire by introducing the concept of localities and structuring addresses to indicate a locality and a location within that locality. There are several possible problems with this approach. First, in the case of virtual memory (as opposed to object ids) autonomy is undermined by the export of addresses within a locality and ability to access within a locality directly (i.e., unmediated by the locality's mechanisms). This is solved by using object ids or a similar level of indirection. Second, the names embed the locality directly, likely making it a bit more difficult to move objects from one locality to another. But the biggest problem is that the addresses are still *long*, and the price-performance argument applies.

5 Narrow client address spaces

The above arguments lead us to conclude that we need to avoid long addresses, that we need to structure the object space into localities, and that there must some indirection in object reference to obtain the desired degree of flexibility. We now consider techniques for using narrow addresses, and as a side effect describe many aspects of Mneme's approach. There are actually two related but distinct issues: use of narrow addresses within running client, and use of narrow addresses in the persistent storage format of objects. In this section we consider narrow client addresses. The following section considers narrow addresses in secondary storage.

5.1 LOOM

Accessing a large space of objects from a machine with a narrower address space has certainly been done before. One well known system, LOOM, is described in [Kaehler and Krasner, 1983] and [Kaehler, 1986]. LOOM stands for "large object oriented memory", and it provided access from a 16 bit wide machine to objects residing in a 32 bit addressed store on disk. The only goal of that system was to expand the effective virtual memory; we have additional goals. Further, LOOM was invented to get around inadequate address space, while we are seeking to keep addresses shorter to boost performance and fit well with existing 32 bit architectures. That is, the degraded performance of widths larger than the currently typical 32 bits, such as 64 or 128 bits, does not imply that going narrower than 32 bits would introduce savings. In fact, as we narrow, eventually we reach a point where we cannot maintain addressability of an application's active collection of objects, and we will induce something akin to thrashing of the mapping of the vast object space into the smaller client address space.

LOOM was found to be effective in two ways. First, it did expand the space of objects that could be used in a Smalltalk system, allowing one to go beyond the 32K object limit[2] LOOM was also judged to make effective use of very scarce main memory, by "paging" on an object rather than page basis, and building a working set of objects rather than pages. With the highly constrained memory available, this seemed to outweigh the cost of retrieving objects individually from secondary storage. Given adequate main memory, clustering objects would likely have been more effective, at least we are assuming so in the design of Mneme. It is not at all clear that, given adequate main memory, 32K objects would represent the "working set" of a substantial application. The point we are making is that LOOM was dealing with severely constrained main memory, whereas we are willing to assume that main memory is adequate to prevent address space "thrashing" (and probably physical memory thrashing as well).

5.2 Temporal locality

So, we are assuming that the virtual address space of the hardware supporting clients is adequate for the programs and data *actually used at one time*. This assumption relies on a principle of *temporal locality*: that an application accesses only a small fraction of the object space at any given time. There is a further assumption that the hardware's virtual address space is adequate for the applications' temporal localities. The principle of temporal locality seems intuitively true, and is related to the well established principle of locality relied upon in designing and using memory hierarchies. The difference is that we are talking about what an application needs to *name* or *address*, rather than what it is actually *manipulating*. Still, we believe the principle holds and hope to verify it in practice. (It cannot be proven in any formal sense, being an empirical principle.)

On the other hand, whether or not 32 bits in particular is large enough for applications' temporal localities clearly depends on the application. We believe that it will be adequate for many, perhaps most applications, but that that there may be some applications where 32 bits is inadequate. Also, as processing speeds and memory size increase, the address size may need to increase as well. If our arguments and principles hold up, though, it should not be necessary to go beyond 64 bits for a very long time.

To be more explicit, we could define the temporal locality of an application at time t for period Δt as the set of object ids used by the application from $t - \Delta t$ to t. Note that this differs from the working set, which is the set of objects actually *manipulated* (accessed) from $t - \Delta t$ through t. The chief difference is that an object id can be passed around without accessing the object to which it refers. Note that given a the speed of the processor and a value Δt we can bound the size of the temporal locality as well as the working set. Suppose, for example, we decided that it was reasonable to make adjustments

[2]While the object ids were 16 bits, one bit is a pointer vs. non-pointer tag bit, giving only 15 bits of object id.

to the address space on the order of seconds. Then we need the *virtual address space* to be large enough that not every address can be manipulated within a second. While this is not the only consideration (we need also to understand the cost of adjusting the addressable set of objects), it does tend to support the notion that 32 bits is likely to be adequate until we have must faster machines.

5.3 Mapping between the client and object spaces

It is clearly necessary to maintain a mapping of client space to the object space if we are to use narrow pointers. That is, we must be able to allocate client address space dynamically to objects, and to reclaim no longer used address space, too, so as to handle long running client programs. The maintenance of client address space is analogous to the maintenance of real memory and virtual memory mapping tables in a virtual memory system. The two tasks are not *precisely* the same for two reasons. First, we are managing client *address space*, not the client's real memory, which can be managed by traditional virtual memory or buffer management techniques, or a combination of them. Second, the address space need not be a space of bytes or words, but may be a space of objects, named by object ids. In Mneme, the client space is an id space.

We will use the term *client id* (cid) for an object id as used by the client, and *global id* (gid) for a form that uniquely identifies an individual object within the entire object space. It will turn out that in Mneme gids are never explicitly built, but they are conceptually present as a locality and an id within that locality.

5.3.1 Mapping cids to gids and virtual addresses

We must be able to map cids to main memory addresses in order to manage virtual (and real) memory. that is we must be able to locate an object in memory, if it is currently resident. Likewise, for non-resident objects, we must be able to locate them in the store and fault them in. For that we need the object's gid. Thus each cid has an associated gid, and possibly a virtual memory address as well. Note that the cid to gid mapping must be available for all *named* objects, whereas the cid to virtual memory address mapping is only for the *resident* objects.

5.3.2 Mapping gids to cids

Since objects as they arrive from secondary storage will have some form of gid in them (this issue is considered in detail in the next section), we will need to be able to convert from gids to cids, so as to determine the name the client is using for the objects mentioned in any object that is fetched.

5.3.3 Implementing the maps: logical segments

The space consumed by the maps and the time required to consult them have critical impacts on system performance. One of the key ideas in the implementation of Mneme is to avoid having a map entry for *each object.* We group the ids together into logical "pages", where all the ids on the same "page" have the same upper bits and differ only in some number of lower bits. We call such a "page" a *logical segment.* Note that the *objects* named by the ids in a logical segment can vary in size and consume no predetermined amount of space. The *ids* all map to the same "page" of client id space, though, and further, the objects will either be all resident or all absent (they are stored and retrieved as a unit). Naturally, the objects in a logical segment should be stored contiguously on disk. We return to the issue of physical object storage below.

We believe a good size for logical segments is between 128 and 1024 object ids. Our current prototype uses 1024, but we expect to switch to 128, to give more flexibility in object placement. Still, it will require experience and measurements to determine the best size in practice.

The benefit of logical segments is smaller maps. Since we are talking about thousands, millions, or more objects, the benefit is significant in that it may mean the difference between fitting the map in real memory and not. If the map does not fit in real memory, then an object fault involves at least two secondary storage accesses, one for the map and one for the object. Note that we are giving up some location independence (all objects in the same logical segment are mapped, fetched, and stored together), but possibly cutting secondary storage access costs in half.

The Mneme scheme does have the drawback of preventing objects from being redistributed among logical segments (reclustered) without changing their ids. We expect reclustering to be done periodically within localities, and that localities will be maintained in such a way that their objects can be renumbered without affecting other localities (i.e., inter-locality references should use an incoming reference table).

5.3.4 Implementing the maps: associating cids and gids

To convert from a gid to a cid, we ignore the lower order bits and map the segment. We could have a table with an entry for segment currently mapped into client address space, and probe it by hashing. This would offer the greatest flexibility in assignment of gid segments to cid segments. What we chose to do in our prototype is to map entire localities contiguously. That is, all logical segments of a locality being used by a client are mapped into a contiguously numbered set of client logical segments. Thus, to map from a gid to a cid, we merely add on a base segment number determined by the locality containing the gid being mapped. Since this mapping is generally done only when objects have been fetched into virtual memory, we know the locality and the mapping turns into a simple relocation, adding a known value to each pointer.

We note that converting gids to cids is a relatively frequent operation, and must be done before an id is released for client use. There are two times that it might be reasonable to do the conversion. One is when an object is fetched from secondary storage. At that time, we might convert all gids in the object to cids. If we fetched a group of objects together, we could convert them all at once, or we could convert each of them as they are first used. Which technique is best depends on how many of the objects are used, and the cost of detecting first use. The other two to do the conversion is as an id is fetched from an object (every time it is fetched). Again, the desirability of this method over converting whole objects is related to application properties, in this case how many accesses are made to ids in objects versus how many total ids there are in the objects. This basically boils down to how heavily the application uses its objects before it is done with them: heavy use suggests converting in advance, light use suggests converting on fetches out of objects. The current prototype converts on each fetch, but we may change to whole object conversion since it integrates more efficiently with programming language run-time systems (avoiding the need for a procedure call to access an object field containing an id).

Mapping entire localities into client space contiguously as we do leads to very efficient gid-to-cid mapping, which is why we chose it, but it does have some drawbacks. First, large localities eat up the client id space, so that clients cannot access a large number of large localities, even if they are using only a few objects in each locality. Secondly, we should over-allocate client space to allow the locality to grow while it is used. We run into difficulty if so many objects are created in the locality that the allocated client id space is consumed. The solution is to allocate an additional, larger, chunk of client id space. The original chunk can remain, though we need to take a little care concerning aliases (two client ids can mean the same object). How severe these problems are is one of the issues we hope to discover in developing and using the prototype.

Converting from a cid to a virtual address involves ignoring the low bits (i.e., forming a client segment number) and mapping the client segment to a location in memory. That location contains a table of pointers to each object in the logical segment, indexed by object number (the low bits of the original id). Thus, our scheme has embedded within it a kind of distributed object table. Note that the cid to virtual address table need only be large enough for the number of logical segments actually resident at one time. This suggests implementing it as a hash table, similar to inverted page tables. Because cids form a linear space, we could also use a more traditional linear map, and simply index by client segment number to an entry that indicates whether the segment is present, and if so, where. This is analogous to the page tables implemented on many systems. The hashing scheme has the advantage of being smaller, since it needs entries only for *resident* segments, not for all *addressable* segments. At the moment, we have implemented a variant of the directly indexed scheme, but will be switching to the hashed scheme shortly.

5.3.5 Clustering objects: physical segments

As previously mentioned, we fetch and store all the objects of a logical segment as a unit. In fact, we allow one or more logical segments to be joined together into a single *physical segment*, where physical segments are the unit of transfer between virtual memory and secondary storage. A physical segment has three parts: a header, describing the segment; an object table part, with one slot for each object, arranged so that the slots for each logical segment are contiguous; and an object data part, that contains the actual contents of each object. The object table entries are self-relative pointers to the object data, so that the entries do not need to be adjusted when the physical segment is transferred to or form secondary storage.

Importantly, physical segments do not have any predetermined size, and may even change size through their lifetime. The variable length of physical segments gives Mneme a flexibility advantage over paged systems with fixed size pages. In terms of fetching data from secondary storage effectively, the physical segments make it easier to fetch what is needed, since they group together exactly what is appropriate to fetch at once. A paged system needs either predictive algorithms, or information equivalent to the physical segment information of Mneme, to do as good a job.

5.3.6 Summary of narrow client addresses

We have discussed how a client can use narrow addresses, and given details of Mneme's approach to client addressing of objects. The approach has some similarity to LOOM, except that we map entire logical segments of object ids at a time, rather than individual object ids. We also map whole localities of segments contiguously, to reduce the wide-to-narrow id conversion cost. Having considered addresses from the client's point of view, we now turn to the issue of how addresses are stored within objects.

6 Narrow objects in secondary storage

If we narrow just the client address space and not the objects, we fail to handle a number of the objections to wide addresses, most notably longer per-object secondary storage and network access times and higher secondary storage and buffer space requirements. So, in Mneme we have gone beyond LOOM and narrowed the secondary storage format of objects, too, to 32 bits. Just as client narrowing required a temporal locality assumption, in narrowing secondary storage we assume that *spatial locality* holds—that most pointers relate pairs of objects in the same locality rather than different localities. If this assumption is not true then we have not clustered the data at all effectively and performance will be terrible anyway.

6.1 Inter-locality references and forwarders

In Mneme, the primary locality concept is the *file*. A file provides a space of up to about 2^{30} objects, which refer to each other by short (32 bit) ids. Thus an id in an object, which we call a *file id* (fid), always names an object in the *same file*. When it is necessary to name an object in *another* file (a different locality) one refers to a local *forwarding object*. The forwarding object has a header bit set to indicate that it is a forwarder, and the contents of the forwarder can be used to locate the target object. Note that since objects can be arbitrarily large, an arbitrary amount of information can be made available for doing the forwarding.

Rather than providing a single, built-in, forwarding scheme, we plan to allow a variety of schemes, as well as to allow systems programmers to add new ones if they wish. The mechanism can be used to provide symbolic, contextual, and a variety of other dynamic kinds of links. This enhances the usefulness and independence of each individual file. For example, suitable stylistic use of forwarders could allow a single file with outgoing references to be shipped to another system and reinterpreted, successfully, in the new local context. To allow files to be garbage collected and reclustered independently, incoming references should go through some kind of table so that the ids in objects need not propagate beyond the locality, as we have pointed out before.

6.2 Policy domains: pools

In addition to the localities offered by files, Mneme also provides the notion of a collection of objects to be managed by a given policy. A Mneme *pool* is a set of objects, within a single file, that have associated with them a set of policy decision routines. Every object is placed in a pool when it is created; in fact, the pool is involved in the allocation and placement of the object as the object is created. Some of the policy decisions the pool can make include the size of physical segments for the pool, which physical segment is to contain a newly created object, which physical segments should be pre-fetched in addition to one being faulted in, the locking policy to apply to objects in the pool, etc. At a physical level, a pool is a collection of physical segments, created and managed by the pool. New object management strategies are formed by writing a routine to handle each of the policy decisions made by pools, and then creating pools using the new routines.

6.3 Summary of narrow objects

Narrow objects are required in order to overcome the various problems with large addresses for objects. Mneme uses 32 bit intra-locality references, and a forwarding mechanism for inter-locality references. Mneme also provides a smaller locality, the pool, for delineating policy for collections of objects. These mechanisms solve the price-performance and autonomy/flexibility problems with large and/or flat addresses in secondary storage.

7 Narrow client ids and narrow objects together

Note that the narrow client id scheme, as well as the narrow object scheme, of Mneme *requires* that some adjustment be made to object contents as seen by a client, since the same file id can be used for two objects in different files, and they must clearly be given distinct client ids to be used by the client at the same time. Since we must be able to change ids, and since we desire the ability to garbage collect and renumber objects in a locality, we need to be able to find and update all the pointers in each object. Currently Mneme does that by segregating the pointers from the non-pointer data, though we are considering the benefits and drawbacks of other techniques.

As previously noted, we are still researching the most effective techniques for transforming objects between the "in locality" format (file ids) and the "in memory" format (client ids), for one can process batches of objects at a time, or a single object at a time (on first use), or not change the contents of the objects but make the id transformation as pointers are fetched and stored.

8 Summary and conclusions

We have argued here as to the inappropriateness of large flat (single level) address space for distributed persistent object stores. There are convenience and functionality arguments in favor of dividing the store in somewhat independent localities. There are also compelling arguments that the long addresses necessary to support a flat store will lead to poorer performance in space and time than the narrower addresses that match nicely with separate localities and more cost effective hardware. In addition to making these arguments in a somewhat general setting, we described relevant aspects of the Mneme persistent object store prototype, designed taking into account the the arguments presented.

Our conclusions are:

- That flat spaces are inappropriate on grounds of functionality, and localities are necessary in decentralized systems.
- That flat spaces, because they require a large number of address bits, use more space and time resources than a well designed system based on localities.
- That Mneme's approach to large decentralized object stores is sound and promising.

Abandoning wide addresses in favor of narrow ones requires conversion of the addresses between their external format (e.g., file ids) and internal format (e.g., client ids). A significant point we have made in this paper is that it will pay to do the id conversion as opposed to using large name spaces—even if doing conversions costs a bit more, the added functionality and flexibility is necessary and worth the additional cost.

References

[Bishop, 1977] Peter B. Bishop. *Computer Systems with a Very Large Address Space and Garbage Collection.* PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, May 1977.

[Brown and Cockshott, 1986] A. L. Brown and W. P. Cockshott. The CPOMS persistent object management system. Tech. Rep. Persistent Programming Research Project 13, University of St. Andrews, Scotland, 1986.

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

[Cockshott, 1987] P. Cockshott. Stable virtual memory. In *Persistent Object Systems: Their Design, Implementation, and Use* (University of St. Andrews, Scotland, Aug. 1987), Department of Computational Science, pp. 470–476.

[Connor *et al.*, 1989] R. Connor, A. Brown, R. Carrick, A. Dearle, and R. Morrison. The persistent abstract machine. In *Persistent Object Systems: Their Design, Implementation, and Use* (University of Newcastle, NSW, Australia, Jan. 1989), Department of Computer Science, pp. 80–95.

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

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

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

[Kaehler and Krasner, 1983] Ted Kaehler and Glenn Krasner. LOOM–large object-oriented memory for Smalltalk-80 systems. In *Smalltalk-80: Bits of History, Words of Advice*, Glenn Krasner, Ed. Addison-Wesley, 1983, ch. 14, pp. 251–270.

[Kaehler, 1986] Ted Kaehler. Virtual memory on a narrow machine for an object-oriented language. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, Sept. 1986), ACM, pp. 87–106.

[Keedy and Rosenberg, 1989] James Leslie Keedy and John Rosenberg. Support for objects in the MONADS architecture. In *Persistent Object Systems: Their Design, Implementation, and Use* (University of Newcastle, NSW, Australia, Jan. 1989), Department of Computer Science, pp. 202–213.

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

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

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

[McCarthy *et al.*, 1984] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*, second ed. MIT Press, Cambridge, MA, 1984.

[Moss and Sinofsky, 1988] J. Eliot B. Moss and Steven Sinofsky. Managing persistent data with Mneme: Designing a reliable, shared object interface. In *Advances in Object-Oriented Database Systems* (Sept. 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–316.

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

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

[Steele Jr., 1984] Guy L. Steele Jr. *Common Lisp: The Language.* Digital Press, Burlington, MA, 1984.