Design of an Object Faulting Persistent Smalltalk*

Antony L. Hosking *J. Eliot B. Moss* *Cynthia Bliss*

COINS Technical Report 90-45
May 1990

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, GTE Laboratories, and the Eastman Kodak Company.

Abstract

We present an approach to supporting persistence in heap-based programming languages, called *object faulting*. By modifying the language run-time system, we provide the illusion of a large heap of objects, only some of which are actually resident in memory. When the run-time system detects a reference to the contents of a non-resident object, an object fault occurs, causing the object to be made resident. We discuss an implementation of these techniques for Smalltalk that uses the Mneme persistent object store as the underlying storage manager.

1 Introduction

This work is motivated by the need for exploration of the boundary between programming languages and databases. Computer-aided design, computer-aided software engineering, document preparation, and office automation are examples of data-intensive applications that must store and retrieve large amounts of highly structured information, and be able to share that information among multiple cooperating users. Environments that can successfully integrate features from programming languages and databases will enable us to build and maintain such applications more easily. Successful integration has been described as overcoming the *impedance mismatch* [Copeland and Maier, 1984] between programming language data models and database data models. Traditional database systems require users to cast their problems first in one model and then the other. Applications programmers would benefit enormously from being able to manipulate persistent data (data that outlive the execution of the program) just as they do non-persistent data. *Persistent programming languages* such as PS-Algol [Atkinson and Morrison, 1985] have shown that persistence can, and ought to, be an *orthogonal* property of data: any data item can potentially persist, independently of its other properties, including its type.

There are significant gains to be made from using object-orientation as a means to language-database integration. Object-orientation is one meeting ground of the programming language and database sub-cultures. A number of object-oriented database systems have been or are being developed, examples of which are GemStone [Maier *et al*., 1986], Orion [Banerjee *et al*., 1987], and Iris [Fishman *et al*., 1987]. For programming languages there are Smalltalk [Goldberg and Robson, 1983], Trellis[1] [Schaffert *et al*., 1986] and CLOS [Bobrow *et al*., 1988], among many others. Integration efforts can build on the experiences of both the database and programming language worlds. More significantly, object-orientation provides encapsulation of data and operations in such a way as to enhance the reusability, maintainability, and extensibility of systems. These are particularly desirable features in an experimental context where exploratory methods are used to evaluate different research ideas as quickly as possible.

Our choice of Smalltalk as a vehicle for exploring the interface between languages and databases was made for the following reasons. First of all, Smalltalk is object-oriented, having the advantages outlined above. Secondly, almost the entire Smalltalk environment (including browsers, compiler, debugger, etc.) is written in Smalltalk. Everything in Smalltalk is an object, including data, code, and run-time stack frames. This machine-independent part of Smalltalk is known as the *virtual image*. To build a running Smalltalk system, we need only implement the underlying Smalltalk *virtual machine*, comprising a stack-oriented interpreter for Smalltalk's instruction bytecodes and a memory management subsystem. It is the virtual machine that gives dynamics to the objects in

[1] Trellis is a trademark of Digital Equipment Corporation.

the virtual image. This last point is important, since our approach to persistence in Smalltalk is to modify the virtual machine to provide the illusion of a large heap of objects, only some of which are actually resident in memory. That is, the techniques we shall use are *dynamic*, affecting only the run-time system. When the run-time system detects a reference to the contents of an object that is not in memory it causes what we call an *object fault*: the required object is brought into memory from disk, and a pointer to it is made available.

While we are taking a dynamic approach to persistence here, *static* techniques may be applied in languages that permit compile-time analysis of programs. In such languages we may be able to optimise away residency checks, by scheduling object faults at compile-time, generating code that will ensure that the object is resident when it is needed at run-time. Dynamic techniques are useful for languages such as Smalltalk, that are only weakly typed, although some static analysis might be of benefit. Static techniques may prove to be highly effective for languages that submit to stronger static analysis.

2 Object Faulting

Object faulting is a technique for managing persistence that is language-independent. Its only requirement is that there be some notion of *pointer* to a data item, and the ability to express traversals across such pointers. The area in which all such referenced data items, or objects, are stored is commonly called the *heap*. Analogous to paged virtual memory systems, object faulting provides what might be called a virtual heap. The heap is not preloaded into memory; rather, objects are faulted on demand.

It will serve our discussion best to consider another analogy: the heap as a *directed graph*. The *nodes* of the graph are the objects, and the *edges* of the graph correspond to pointers from one object to another. A computation involves traversing the object graph, which is only partially resident in memory. Traversing an edge from a resident object to a non-resident object causes an object fault, and the link is *snapped* to point to the newly resident object. It is important to note here that merely naming an object does not cause an object fault. Only when the *contents* of the object need to be accessed, and so the link to the object must be traversed, is it required that the object be made resident.

The run-time system must have some way of detecting the traversal of links from resident to non-resident objects. There are effectively just two ways of achieving this:

- *Mark the edges* of the graph that are links to non-resident objects, distinguishing them from links to resident objects (see Figure 1).

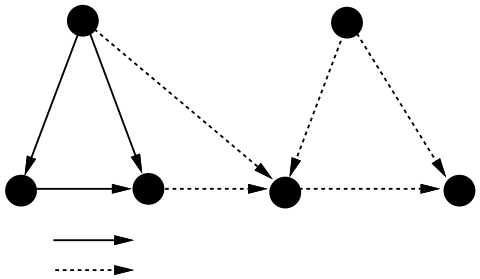

Figure 1: Marking edges

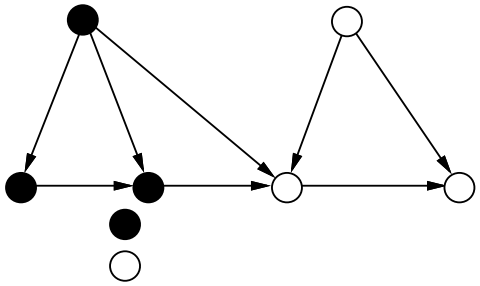

Figure 2: Marking nodes

- *Mark the nodes* of the graph to distinguish resident objects from non-resident objects (see Figure 2).

Edge marking is relatively easy to implement by tagging pointers. Checking whether a pointer refers to a resident object or not is simply a matter of checking the tag. When a marked link is traversed, we must first make sure that the object it refers to is resident, faulting it in if necessary (the object may already be resident if some other link to it has previously been traversed), then the link is snapped to point to the resident object (see Figure 3). Note that it is legal (though suboptimal) for a marked edge to refer to a resident object, but an unmarked edge may never refer to a non-resident object.

Node marking is complicated by the fact that the non-resident objects are just that, non-resident,

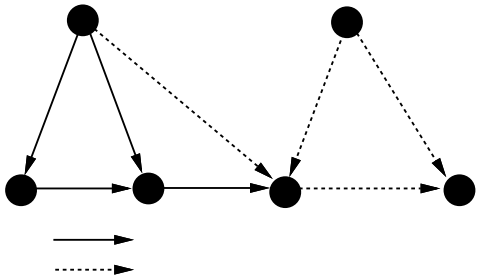

Figure 3: Object faulting with edge marking

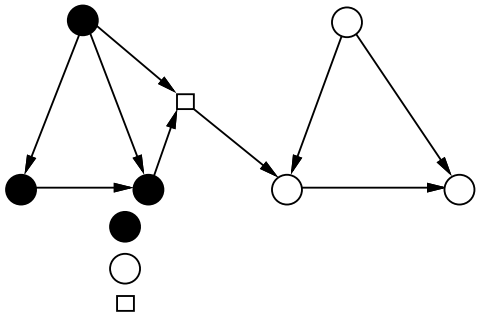

Figure 4: Fault Blocks

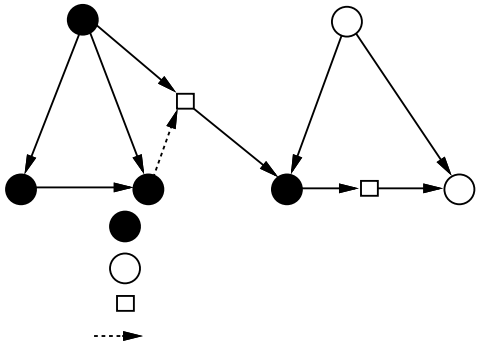

Figure 5: Object faulting with fault blocks

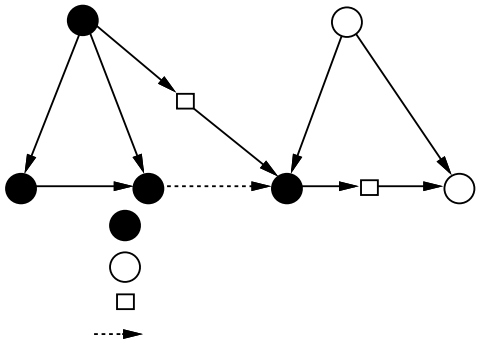

Figure 6: Updating the traversed link

and must (paradoxically) be in memory for them to be checked. To overcome this we can use an approach similar to *leaves* in LOOM [Kaehler and Krasner, 1983]. A leaf is a specially marked resident pseudo-object that stands in for a non-resident object. We call such fake objects *fault blocks*. All references from resident objects to non-resident objects are actually pointers to a fault block (see Figure 4). When a link is traversed to a fault block, we must check if the corresponding object is resident, faulting it in if necessary. "Snapping the link" in this case involves setting the fault block to point to the resident object (see Figure 5). We must also create fault blocks for any objects to which the newly resident object refers. Note that there is now a level of indirection via the fault block; this may be bypassed by also updating the traversed link to point to the object in memory (see Figure 6).

The preceding discussion assumes that we have some way of locating non-resident objects, in order to obtain a pointer to them. We assume the existence of an underlying *object manager* that, given some unique *object identifier*, will return a pointer to the object in memory, faulting it in if necessary. We thus need somewhere to store the identifier that we pass to the object manager to obtain the address of the object. In the edge marking scheme we store the identifier in the bits of the pointer that are not reserved for the tag. In the node marking scheme we store it in the fault block.

Note that these schemes are not mutually exclusive, and may work side by side. They each have their advantages. A particular fault block may be referenced by many resident objects. This means that the object manager need only be called once per fault block, to obtain a memory pointer to the corresponding object, when the first link to the fault block is traversed. The memory address is then cached in the fault block so that subsequent traversals of links to that fault block can pick it up from there, without additional calls to the object manager. However, there are overheads associated with fault blocks: storage management for fault blocks; creation of fault blocks for all the objects referred to by a non-resident object when it is faulted in; and the extra level of indirection that fault blocks imply.

Edge marking has the advantage of eliminating the space consumed by fault blocks, and the level of indirection associated with them. Its disadvantage is that the only link that is "snapped" when an object is faulted in is the link that is traversed. All other links to the object are still marked as pointing to a non-resident object. This means that every traversal of a marked link will result in a call to the object manager to determine the object's address, regardless of whether the object is already resident or not.

3 Faulting Smalltalk Objects

Our implementation of Smalltalk is based on the definition of the Smalltalk-80[2] language found in the "blue book" [Goldberg and Robson, 1983]. We have made only minor extensions to the virtual image. While we have retained the standard bytecode instruction set, our implementation of the virtual machine is somewhat different from that defined in the blue book.

The blue book implementation of the heap used an *object table*. All object references were actually indices into this table. Each object table entry then contained a pointer to the actual object in memory. There were a number of reasons for this extra level of indirection. One is that when an object was moved by the garbage collector the only pointer to the object that needed to be updated was the pointer in the object table entry for that object. Also, given that the blue book design was for a 16-bit architecture, using an object table realised a significant gain in address space. Since object table entries were significantly smaller than the average size of an object, more objects could be referenced by indirecting through the object table than could be directly addressed. More recently, *generation scavenging* garbage collection schemes have been devised that eliminate the need for an object table, while retaining performance [Ungar, 1984, Ungar, 1987]. Furthermore, our implementation is for a 32-bit paged virtual-memory architecture, reducing addressability concerns.

We have implemented Smalltalk on VAX[3] hardware, with no object table, using a variant of generation scavenging garbage collection. Object pointers are 32-bit tagged entities. We allocate objects on 32-bit word boundaries so that on a byte-addressed machine we have two bits left over for tagging. One bit is used to distinguish *immediate*[4] values from memory pointers. The other bit distinguishes immediate 30-bit signed integers (of class SmallInteger in Smalltalk) from a few other immediate values (points, characters, true, false, nil), that are distinguished from one another by extending the tag (see Table 1).

Our design of object faulting for Smalltalk uses the Mneme persistent object store [Moss and Sinofsky, 1988, Moss, 1989a, Moss, 1989b] as the underlying object manager. Edge marking is achieved by making use of the as yet unassigned tag value, 0001, to tag pointers to non-resident objects, with the rest of the pointer being used to store the 28-bit object identifier expected by Mneme. For node marking we use fault blocks that are faked up to look like the object header of a memory-resident object, with a 32-bit flags field and a 32-bit class field. However, in place of the object's class, we store the tagged Mneme identifier, overwriting it with a pointer to the object when it is faulted. A bit in the flags field indicates that this is actually a fault block (see Figure 7).

[2] Smalltalk-80 is a trademark of Xerox Parc.

[3] VAX is a trademark of Digital Equipment Corporation.

[4] Immediates are values stored directly in the object pointer.

memory pointer	...00
unassigned	...0001
immediate point	...0101
immediate character	...1001
nil	0...0001101
false	0...0011101
unassigned	...101101
true	0...0111101
unusable	...10
immediate integer	...11

Table 1: Tagging Object Pointers

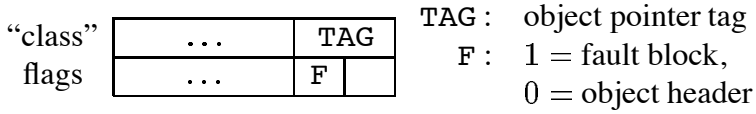

Figure 7: Format of a fault block

Whether the "class" field contains a Mneme object identifier or a pointer to the object in memory is determined by the low-bit of the tag: a zero indicates that the value stored is a pointer, a one indicates that it is an identifier.[5]

3.1 Efficiency Issues

Computation in Smalltalk proceeds by *sending messages* to objects. A message consists of a *message selector* and a number of arguments. The effect of sending a message is to invoke a *method* on the receiver of the message. Invoking a method may be thought of as a procedure call. A stack frame is initialised for the call, the arguments are pushed on the stack and the object code for the method is executed. The method to be executed is determined at run-time, based on the message selector and the class of the receiver. Every class object in Smalltalk has a pointer to a *method dictionary* that associates selectors with *compiled methods*. A compiled method consists of the bytecodes that implement the method, along with a *literal frame*, containing the shared variables, constants, and message selectors used by the method's bytecodes. Determining which method to execute when a message is sent proceeds as follows. The receiver's class is checked to see if its method dictionary contains the message selector. If it does then the corresponding

[5]The "class" field of a fault block should never contain any other immediate value, allowing this to be a single-bit check!

compiled method is invoked. Otherwise, the search continues in the superclass of the object, and so on, up the class hierarchy. If no matching selector is found then a run-time error is signalled.

Looking up methods as just described is an expensive process. To reduce this lookup cost a method lookup cache is used. Entries in the cache store a selector, class pointer, and compiled method pointer. Before proceeding to a full method lookup, the selector and class are hashed to index an entry in the cache. If the selector and class of the cache entry match those of the message send, then the compiled method has been found. If they do not, then a full lookup must take place, updating the corresponding cache entry as well.

The object faulting approach is dynamic, imposing a check on every traversal of a link. Our discussion of message sends has illustrated just how many objects must be accessed as computation proceeds. For performance reasons it is crucial that the bytecode interpreter not be overly taxed by having to perform a residency check on every object that it needs to access. To overcome this we preload a number of these critical objects, the result of which is to restrict residency checks to message sends.

Because computation is driven by the sending of messages, most objects will become resident only when a message is sent to them. In send bytecodes, a residency check must be performed on the receiver, since the receiver's class is required for method lookup. To eliminate other residency checks from sends, whenever an object is made resident we insist that its class object and all its superclasses be made resident, along with their method dictionaries and the message selectors in those dictionaries. If we also make sure that only resident methods are entered in the method lookup cache, then all references in the cache will be to resident objects. This means that the method lookup code will not have to perform residency checks unless it takes a cache miss and is forced to do a full lookup. A full lookup must make sure the method is resident before its cache entry is loaded.

Whenever a method is made resident (usually through its invocation), we require that the literals in its literal frame be made resident. This forces the selectors, constants, and shared variables (Association objects with two fields, one for a name and one for a value) referred to by the bytecodes to be resident. It does not force the objects *referred to* by the shared variable associations to be resident. This permits the bytecodes accessing the selectors, constants, and shared variables of the literal frame to do so without performing residency checks. In short, there is no need for residency checks in the stack bytecodes.

Stack frames are also objects in the Smalltalk system, known as *contexts*, and so may be persistent. By requiring all context objects to be resident we eliminate residency checks in return bytecodes. Otherwise we must check whether the context being returned to is resident or not.

In summary, by preloading objects that are critical to the forward progress of computation, we

are able to restrict all residency checks to message sends.[6] Furthermore, when everything is resident the execution overhead is just one check per message send.

3.2 Heuristics

Depending on whether we are using node marking or edge marking we may also apply a number of heuristics. We have already mentioned some heuristics that may be employed with fault blocks, to avoid extra calls to the object store and to eliminate the indirection that fault blocks imply.

In the case of edge marking, we are faced with the problem that since most objects are made resident when a message is sent to them, the marked link that ends up getting snapped (the receiver) is on the stack of the virtual machine. We cannot update the receiver link at its source, in the object from which it was pushed onto the stack, so snapping the link buys us nothing. However, a heuristic we can apply is to scan the active context, and possibly further up the stack, updating all other marked links to the object. How far to scan is a parameter that we can tune, based on our experience with the running system.

If it turns out that doing a scan *every* time a marked link is traversed is too expensive, we can perform the scan only when we traverse a *marked* link to a *resident* object (i.e., traversing the link does not cause an object fault). We are betting here that traversing a second marked link to an object is an indicator that there are further marked links to it floating around, so scanning the stack for them would be helpful.

Our intention is that these heuristics, along with preloading of critical objects, will allow object faulting to have minimal impact on the performance of the Smalltalk system when only resident objects are being accessed. Only when the system is fully implemented will we be able to experiment with, and evaluate, these techniques.

4 Object Faulting and the Mneme Persistent Object Store

So far we have been concerned solely with the impact of object faulting on the programming language. We now discuss the interactions between the run-time system and the object store.

The Mneme object store provides the illusion of a large, shared, persistent heap of objects, directly accessible from client applications. A Mneme object is a collection of *slots* and *bytes*, along with a 1-byte *attributes* field. Mneme does not specify what the attributes are to be used for. The bytes part is simply a vector of 8-bit bytes. The slots part is a vector of 32-bit slots. Each slot contains one of two things, depending on the sign bit of the slot: an immediate 31-bit integer value, or an *object identifier*. There is a distinguished *empty* object identifier.

[6] Primitives may need to perform additional residency checks on objects they need to access.

Mneme groups objects together into units called *files*. A file of objects can be separately named and located within the overall store. Files are a convenient unit for storage, providing modularity of the object space, and are intended to be reasonable units of backup, recovery, and transfer between different Mneme stores.

Files also allow us to take advantage of modularity of name space. *Persistent* object identifiers, as stored in objects within the Mneme store, always name objects within the *same* file as the object containing the identifier. This allows identifiers to be relatively short (28 bits). References to objects in other files are made by referring to *forwarder* objects within the same file. A forwarder contains enough information to name and locate the intended target object.

Because clients (e.g., Smalltalk) may have many files open at the same time during a session of interaction with the store, a persistent identifier, which is unique only within one file, must be converted into a *client* identifier for use by the client. An object's client identifier is guaranteed to name the object uniquely for the duration of a Mneme *session*. This also permits the persistent identifiers to be reassigned between sessions, allowing reclustering, garbage collection, reuse of the limited space of identifiers, and even explicit deletion. Since identifiers can be reassigned by the object store between sessions they cannot simply be synthesised and presented to the store with any reliability. Thus it is essential that each file have a distinguished *root*: a slot that can be set at will to indicate a starting place for naming objects in the file.

A Mneme session is a period of interaction with the store, establishing a context of use, including open Mneme files and identifiers of objects within those files. We can view a session as being a window onto the store. While the limited size of Mneme client identifiers does limit the number of objects that may be uniquely addressed during any one Mneme session, it does not limit the overall number of objects in the store.

The unit of retrieval in Mneme is the *physical segment*. A physical segment physically groups a number of objects together. When one of the objects in the physical segment is to be faulted in, the whole segment is placed in a buffer in memory. A file constitutes a number of physical segments.

When a client requires access to a persistent object it presents a *client* identifier to Mneme. If the object is already resident Mneme simply returns a pointer to the object in virtual memory. If the object is not already resident, Mneme allocates a buffer in memory in which it places the object's physical segment, and then returns a pointer to the requested object within that buffer.

4.1 Client Interaction with the Store

Objects stored within Mneme files and faulted into Mneme buffers have a format substantially different from that expected by Smalltalk. Mneme objects have their own header information, followed by the slots and then the bytes. The slots may contain persistent object identifiers, or

immediates. We store a Smalltalk object in a Mneme object as follows. The Smalltalk object's header fields are stored in the first few slots of the Mneme object as immediates, except for the class field, which is stored as a persistent identifier. The bytes or words fields of a Smalltalk object are uninterpreted, so they may be stored in the bytes part of a Mneme object. The object pointer fields must be converted, however, for storage in a Mneme slot. Since Smalltalk immediates always have the low bit set we can store the remaining 31 bits in a Mneme immediate. Memory pointers must be converted to a persistent identifier for the object they address. Mneme provides routines for converting client identifiers to persistent identifiers, and vice versa. To convert a pointer to an object in a Mneme buffer to an identifier, we store the identifier in the first slot of the Mneme object.

Whenever a persistent object is made resident we must convert it from the external Mneme format to the internal Smalltalk format. We use one of the Mneme object's attribute bits to indicate whether the object is converted or not. When Smalltalk requests a pointer to a persistent object we check this bit, to determine if we need to convert the object. If conversion is necessary we proceed as follows. We first set the attribute bit to indicate that the object has been converted. If a slot contains an immediate, then we convert it by shifting and then setting one bit. If a slot contains a persistent identifier we must convert the identifier to a client identifier using the Mneme routines provided. Then, if we are using fault blocks, we allocate one, storing the (appropriately tagged) client identifier in it, and overwrite the original slot with a pointer to the fault block. Otherwise we simply store the (appropriately tagged) client identifier back in the slot. As for the bytes part of a Mneme object, we have already indicated they do not need converting.

Our discussion so far has only considered bringing objects *into* memory from the object store. We must also deal with writing Mneme buffers back to disk. This is complicated by the fact that a buffer about to be written out, and the rest of object memory, are in what might be termed a "deadly embrace." Objects in the buffer refer to objects outside the buffer and vice versa. Before we can write out the buffer we must disentangle it from the rest of memory. First, there may be some volatile objects that must now be made persistent, since objects in the buffer refer to them. We must perform a transitive closure operation to make persistent each volatile object that can be reached from the buffer, copying its contents into a Mneme buffer, and leaving in its place a fault block that points to the now persistent object in the Mneme buffer. The next step is to convert all the objects in the buffer to Mneme format, reversing the conversion procedure described earlier. Finally, all external pointers to objects in the buffer must be found and updated to point to a fault block, or, if we are using edge marking, marked to indicate they refer to a non-resident object.

To perform this last operation (updating all the external references to objects in the buffer) is not as formidable as it seems. We use a technique that is already employed by the generation scavenging garbage collector. A *remembered set* is associated with each buffer to record the memory location

of all pointers into the buffer. Since generation scavenging requires the maintenance of such sets for each generation, maintaining the same information for Mneme buffers requires little additional mechanism.

5 Related Work

We have already mentioned LOOM as incorporating techniques similar to those we have used for object faulting. It uses a node marking scheme, with *leaves* having a similar function to our fault blocks. However, our object faulting model is an advance over LOOM in that we incorporate edge marking as well. The goal of LOOM was to provide extended virtual memory support for Smalltalk systems on machines with a narrow (16-bit) word width. Object pointers are stored in 32 bits on disk, and an object table is used to translate between the short and long forms. When an object is brought into memory, its 32-bit persistent pointer is hashed to find an entry for it in the object table. All in-memory references to the object are then indirected through its object table entry. LOOM uses a reference counting garbage collector, and takes advantage of this to recycle the object table entries. LOOM permits up to 2^{31} objects to be addressed, although only 2^{16} objects may actually be resident at any time. An important difference between our design for persistent Smalltalk and LOOM is that we do not use an object table, and use generation scavenging instead of reference counting for garbage collection. More significantly, our goals are much more ambitious than virtual memory for Smalltalk. We can store many more objects in our object store than will fit in virtual memory.

We also believe that our implementation of persistent Smalltalk will have much better performance than LOOM, which suffered because an object fault caused the retrieval of just one object. Smalltalk objects are too small a unit for retrieval. Our design, using Mneme, makes the physical segment (containing possibly thousands of Smalltalk objects) the unit of retrieval. Using the extensible policy mechanisms of Mneme we intend to cluster related objects in each physical segment, so that retrieving one object will retrieve objects related to it as well. This will result in a marked improvement in performance over LOOM. We also intend to make provision for sharing and reliability by building on Mneme's concurrency control and recovery facilities.

The Alltalk system [Straw *et al.*, 1989] shares many of our goals. However, the approach it takes is very similar to LOOM, using an object table to translate between object pointers and memory addresses. Alltalk does not translate objects between disk and memory formats, so that its object pointers are always external identifiers, and consequently must always be looked up. We permit object pointers to be both real memory addresses and object identifiers, and our design demonstrates an alternative approach to persistence that does not require an explicit object table.

GemStone [Purdy *et al.*, 1987], is another effort to expand the Smalltalk heap to include objects

on disk. However, it extends Smalltalk to provide considerable database functionality, including queries and an execution model. Integration with Smalltalk systems is not totally "seamless," since the virtual image is modified to include *proxy* objects that act as forwarders to GemStone objects. Proxies, because they are objects in the virtual image, are visible to applications programmers. Our approach to persistence does not modify the virtual image, providing a more seamless environment.

Our approach to persistence is to modify the run-time system of the programming language, building from the programming language down to the database. Object-oriented databases, such as Orion [Kim *et al*., 1988], and Exodus [Carey *et al*., 1986], approach language-database integration from the database up. Persistence in our object-faulting Smalltalk is orthogonal, unlike Orion and the Exodus database implementation language, E [Richardson, 1989].

6 Conclusions and Future Work

We believe that the object-faulting approach to persistence will prove to be effective. We have demonstrated that persistence does not require an explicit object table for translation of persistent identifiers to memory addresses, and that consequently generation scavenging garbage collection techniques may be applied. We have also restricted the essential overhead for persistent Smalltalk to one residency check per send. Without modifying the Smalltalk compiler (or adding some native code translation scheme) to perform some static analysis of Smalltalk methods, we cannot further reduce this overhead. Static approaches could be inspired by the techniques used in the Self compiler [Ungar and Smith, 1987, Chambers and Ungar, 1989, Chambers *et al*., 1989].

Our plans for the future call for completion of our implementation of object-faulting persistent Smalltalk, allowing us to evaluate the design for performance, and experiment with its variations. More ambitious goals are to incorporate sharing and reliability. An issue that we have not addressed here is how to permit Smalltalk users to exert control over the underlying mechanisms of the object store, such as object placement strategies, checkpointing, and recovery. The object store cannot make calls back into Smalltalk code, so we must provide some sort of declarative mechanism by which the user may influence the decisions of the store.

References

[Atkinson and Morrison, 1985] Malcolm P. Atkinson and Ronald Morrison. Procedures as persistent data objects. *ACM Transactions on Programming Languages and Systems 7*, 4 (October 1985), 539–559.

[Banerjee *et al*., 1987] Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, Won Kim, Darrell Woelk, Nat Ballou, and Houng-Joo Kim. Data model issues for object-oriented applications. *ACM Transactions on Office Information Systems 5*, 1 (January 1987), 3–26.

[Bobrow *et al*., 1988] Daniel G. Bobrow, Linda G. DeMichiel, Richard P. Gabriel, Sonya E. Keene, Gregor Kiczales, and David A. Moon. Common Lisp object system specification. *ACM SIGPLAN Notices 23*, special issue (September 1988). ANSI X3J13 Document 88-002R.

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

[Chambers and Ungar, 1989] Craig Chambers and David Ungar. Customization: Optimizing compiler technology for SELF, a dynamically-typed object-oriented programming language. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation* (Portland, OR, June 1989), vol. 24, no. 7 of *ACM SIGPLAN Notices*, ACM, pp. 146–160.

[Chambers *et al*., 1989] Craig Chambers, David Ungar, and Elgin Lee. An efficient implementation of SELF, a dynamically-typed object-oriented language based on prototypes. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (New Orleans, LA, October 1989), vol. 24, no. 10 of *ACM SIGPLAN Notices*, ACM, pp. 49–70.

[Copeland and Maier, 1984] George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Boston, MA, June 1984), vol. 14, no.2 of *ACM SIGMOD Record*, ACM, pp. 316–325.

[Fishman *et al*., 1987] D. H. Fishman, D. Beech, H. P. Cate, E. C. Chow, T. Connors, J. W. Davis, N. Derrett, C. G. Hoch, W. Kent, P. Lyngbaek, B. Mahbod, M. A. Neimat, T. A. Ryan, and M. C. Shan. Iris: An object-oriented database management system. *ACM Transactions on Office Information Systems 5*, 1 (January 1987), 48–69.

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

[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.

[Kim *et al*., 1988] Won Kim, Nat Ballou, Jay Banerjee, Hong-Tai Chou, Jorge F. Garza, and Darrell Woelk. Integrating an object-oriented programming system with a database system. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (San Diego, California, November 1988), vol. 23, no. 11 of *ACM SIGPLAN Notices*, ACM, pp. 142–152.

[Maier *et al*., 1986] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an object-oriented DBMS. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Portland, OR, September 1986), vol. 21, no. 11 of *ACM SIGPLAN Notices*, ACM, pp. 472–482.

[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* (September 1988), vol. 334 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 298–316.

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

[Moss, 1989b] J. Eliot B. Moss. The Mneme persistent object store. COINS Technical Report 89-107, Department of Computer and Information Science, University of Massachusetts, Amherst, MA, October 1989. Submitted for publication.

[Purdy *et al*., 1987] Alan Purdy, Bruce Schuchardt, and David Maier. Integrating an object server with other worlds. *ACM Transactions on Office Information Systems 5*, 1 (January 1987), 27–47.

[Richardson, 1989] Joel Edward Richardson. *E: A Persistent Systems Implementation Language*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, WI, August 1989. Available as Computer Sciences Technical Report #868.

[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, September 1986), vol. 21, no. 11 of *ACM SIGPLAN Notices*, ACM, pp. 9–16.

[Straw *et al*., 1989] Andrew Straw, Fred Mellender, and Steve Riegel. Object management in a persistent smalltalk system. *Software: Practice and Experience 19*, 8 (August 1989), 719–737.

[Ungar and Smith, 1987] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications* (Orlando, FL, October 1987), vol. 22, no. 11 of *ACM SIGPLAN Notices*, ACM, pp. 227–241.

[Ungar, 1984] David Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments* (Pittsburgh, PA, April 1984), ACM SIGPLAN Notices, ACM, pp. 157–167.

[Ungar, 1987] David Michael Ungar. *The Design and Evaluation of a High Performance Smalltalk System*. ACM Distinguished Dissertations. The MIT Press, Cambridge, MA, 1987. Ph.D. Dissertation, University of California at Berkeley, February 1986.