Main Memory Management for Persistence

Antony L. Hosking

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

Abstract

Reachability-based persistence imposes new requirements for main memory management in general, and garbage collection in particular. After a brief introduction to the characteristics and requirements of reachability-based persistence, we present the design of a run-time storage manager for Persistent Smalltalk and Persistent Modula-3, which allows the reclamation of storage from both temporary objects and buffered persistent objects.

1 Introduction

Persistent programming languages are intended to support the uniform manipulation of objects regardless of their lifetimes. Just as ordinary programming languages allow the manipulation of temporary objects (objects that live only until the program terminates), persistent programming languages allow the *transparent* manipulation of permanent objects (objects that outlive the program). In contrast, *non*-persistent programming languages require the explicit use of a file system or database for the storage and retrieval of permanent objects. A particular goal of language designers has been the provision of *orthogonal* persistence, allowing programs to be expressed independently of the longevity of the data they manipulate, and conversely, so that the longevity of an object is independent of the way it is manipulated [1, 2].

A common approach to supporting orthogonal persistence in a programming language is to provide the illusion of a persistent heap. As far as the programmer is concerned, all objects are allocated and manipulated in the persistent heap just as if they were in an ordinary main memory heap. However, objects may survive from one run of a program to the next, by virtue of being reachable from some set of *persistent roots*. When a persistent program begins execution some of its variables are initialized by binding them to (some subset of) the persistent roots. Thus, the program is able to begin manipulating permanent objects in the persistent heap, creating new objects and updating old objects to refer to them. These new objects must also eventually be made to persist if they are in some way reachable from the persistent roots.

Implementing the abstraction of a persistent heap can be achieved in many ways. The simplest approach is that used in programming environments such as Smalltalk [6]. The heap is stored in some file on disk and then loaded into memory in its entirety when the environment begins executing. New objects are created and old objects updated in main memory. Finally, the user may elect to save the updates by asking that the heap, in its entirety, be written back to disk. This approach is acceptable in single-user systems where the heap is small enough to fit entirely in (virtual) memory, and if the load and save times are tolerable. However, there are many data-intensive applications, such as computer-aided design (CAD), that must deal with large amounts of shared permanent data for which this approach is not practical. If the application accesses only a small fraction of a large persistent heap then indiscriminate loading of all the objects is clearly undesirable. Rather, we would prefer to retrieve just those objects that the application needs to manipulate. Without knowing its access patterns in advance, retrieval must be triggered by the program's need to access objects that are not yet resident. Such a mechanism has been called *object faulting*. As in paged virtual memory systems, where a page fault causes a non-resident page to be brought into physical memory, an object fault causes the retrieval of a non-resident object. We call that part of the persistent heap that is in main memory *volatile* since only objects in memory can be manipulated and updated.

Our version of object faulting [7, 8, 9] is being used to implement persistence for Smalltalk and Modula-3 [5]. Here, we concern ourselves with management of the volatile heap, and the requirements imposed by persistence on the techniques used. We present a design that allows reclamation of storage in the volatile heap, with garbage collection of tem-

porary objects and reuse of memory used to buffer permanent objects.

2 Object faulting

The basic idea of object faulting is to perform *residency checks* at run time to ensure that an object is resident before it is manipulated. We implement these checks in a manner similar to LOOM [11], by having resident pseudo-objects called *fault blocks* stand in for non-resident objects. Every memory reference to a non-resident object is actually a pointer to a fault block. The check itself can be performed either explicitly, assuming that fault blocks are specially marked to distinguish them from other objects, or implicitly using paging hardware, by allocating fault blocks in pages of memory set to "no access" and arranging for page traps to invoke a fault handling routine.

The approach is illustrated in Figure 1. In Figure 1(a) we see that a non-resident object is referred to by pointers to its fault block. When a pointer to the fault block is dereferenced an object fault occurs, bringing the non-resident object into memory (Figure 1(b)). The fault block is overwritten with an *indirect block* which contains a pointer to the now resident object in memory. Note that the newly resident object may contain references to other persistent objects. These will typically be represented as *object identifiers* in permanent storage, but will be converted to in-memory pointers when the object is made resident, in a process known as *swizzling* [13]. In Figure 1(b) such a reference has been swizzled to point to a fault block. Note also that there remains a level of indirection via the indirect block. To eliminate some of this overhead we arrange for the garbage collector to bypass any indirect block it comes across. There is a tradeoff to this, since it requires us to check for indirect blocks at run time, similarly to fault blocks. Alternatively, we could adopt the convention that every resident object must be referenced through an indirect block, so that such checks would be unnecessary.

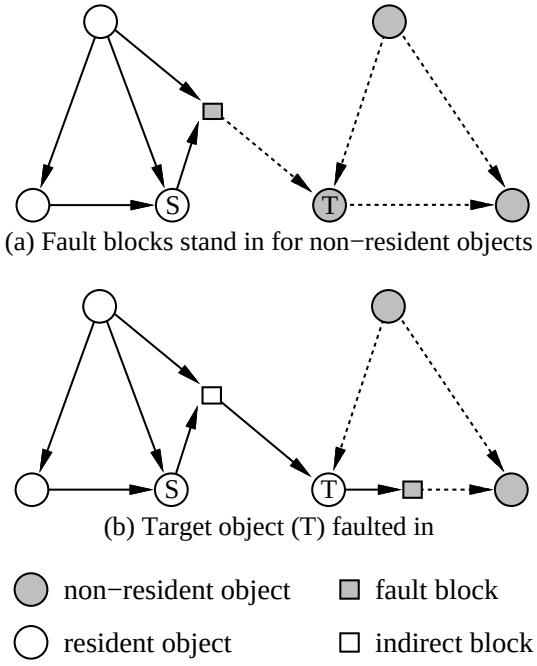

Figure 1: Object Faulting

2.1 Handling an object fault

Object faults are handled by calls to an underlying secondary storage manager, which supports identifier-based retrieval of objects from secondary storage. The fault block actually contains the identifier for the persistent object it represents. At fault time this identifier is passed to the storage manager to retrieve the object. We use the Mneme persistent object store as the underlying storage manager [12].

Retrieving just one object at every object fault has been shown to be extremely inefficient—this was the chief downfall of LOOM. Fortunately, Mneme allows us to group objects into segments for retrieval, so that an object fault actually retrieves an entire segment of objects, which is placed in memory in Mneme's buffers. Clustering techniques may place related objects in the same segment.

As mentioned earlier, objects can be swizzled from their persistent format to the in-memory format expected by the program. We perform *copy swizzling*: when an object is first faulted we make a swizzled copy of it in a specially managed persistent area of the volatile heap. The persistent form of the object may contain references to other persistent objects. Swizzling converts these references to in-memory pointers. To avoid both conditional code and lookup cost in swizzling we can convert all such references into newly allocated fault blocks.[1] This means that there may be more than one fault block for any given object and implies that a fault block may refer to an object that is already resident. To avoid making more than one copy of a persistent object we must keep track of which objects have been swizzled by maintaining some sort of resident object table mapping persistent identifiers to their swizzled copies; Mneme supports this mapping efficiently.

In summary, an object fault is handled as follows (see Figure 2):

[1] In some circumstances the creation of a fault block can be avoided if the reference being swizzled can quickly be determined to refer to a resident object. In general, this is not easy to establish. However, references to objects clustered in the same segment can be specially marked to indicate they should be swizzled to point directly at their target object. Further, we can go ahead and retrieve the target object if it is known that it will eventually be needed. This can be established through compile-time analysis of programs [7, 8].

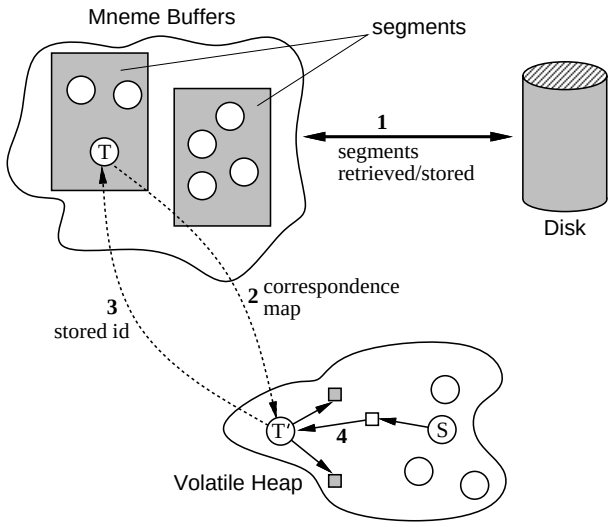

Figure 2: Handling an Object Fault

1. The secondary storage manager retrieves the segment containing the target object (T) if the segment is not already resident in its buffers.

2. If the resident object table contains no entry for the object, then a swizzled copy of it (T′) is made in the volatile heap and the map is updated.

3. The object's identifier is stored with its swizzled copy.

4. A pointer to the copy swizzled object is returned.

2.2 Writing objects back

When a program finishes execution, all its persistent objects must be written back to disk. This means that the buffers of the secondary storage manager must be updated to reflect any changes before they are written out to secondary storage. Updating the buffers involves *un*swizzling all modified objects in the buffer: every in-memory pointer is replaced with the persistent identifier of the object it refers to. The persistent identifier stored with each swizzled copy enables this. If a pointer refers to a newly created object then that object must be *promoted*: space must be allocated for it in the persistent store and a persistent identifier assigned to it. In its turn it will also eventually need to be unswizzled, perhaps dragging further new objects with it into the persistent store.

2.3 Buffer management

Buffer management allows the buffering in memory of just that part of the persistent heap that is needed by the program at any stage of its execution. It is necessary for a number of reasons. Firstly, programs that manipulate large amounts of persistent data may tie up memory with persistent objects that no longer need to be resident. We would like to be able to reuse this memory for other purposes. Secondly, our eventual goal is to provide for multi-user access, in which concurrently executing programs compete for access to objects in the same persistent heap. Objects that are no longer being manipulated should be returned to the shared persistent store so that other programs needing to access them can proceed.

2.4 Implications for memory management

Our model of persistence has certain implications for run-time memory management. Firstly, only transient—as yet non-persistent—objects should be subject to run-time garbage collection, since they are the only objects for which finding all references is relatively cheap (their references all reside in memory).[2] Note that the root set for collection of transient objects must include all references from resident persistent objects to transient objects. Secondly, promotion of a transient object requires moving it into the separately managed persistent area. We achieve this by copying the object over to the persistent area and leaving an indirect block in its place. To reclaim the space occupied by the original transient object we must guarantee that all pointers to the indirect block are eventually found and updated to refer to the promoted object in the persistent area. This limits the style of garbage collection we can apply by excluding collection schemes that do not accurately find all pointers to objects, such as ambiguous roots collectors [3, 4].

3 Design and rationale

As we have mentioned, the volatile heap is partitioned into two areas: a transient area, and a persistent area (see Figure 3). The transient area is managed by a multi-generational scavenging garbage collector (for details of this collector see [10])—such collectors focus their efforts on scavenging younger generations, based on the observation that young objects tend to die young [14]. New objects are allocated in the youngest generation of the transient area. Objects that have survived a certain number of scavenges at one generation are promoted to the next higher generation, on the bet that they are likely to survive even longer and so should be moved out of the more frequently scavenged younger generations.

[2] This does not preclude garbage collection of the objects in the persistent store. However, we see this as an "off-line" activity since it requires tracing all objects in the store reachable from the persistent roots. Doing this as part of a running application would be disastrously expensive since it involves touching (and faulting) many more objects than the application might ever otherwise retrieve.

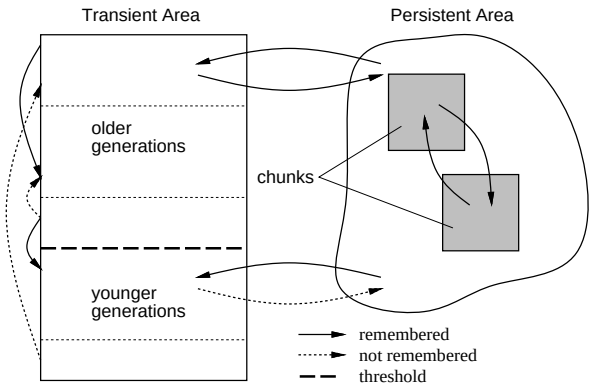

Figure 3: The volatile heap and remembered pointers

As is typical of generational collectors, each generation g has an associated *remembered set* which keeps track of pointers from older objects to objects in g.[3] This includes pointers from older generations to younger generations, as well as pointers from the persistent area to the transient area. Since pointers from old objects to young objects have been observed to be relatively rare in systems such as Smalltalk, the remembered sets are unlikely to be overly large. On the other hand, pointers from young objects to old objects are much more common, so we do not keep track of them. Thus, in addition to pointers from the run-time stack and registers, the root set used to find all objects that must survive a scavenge of g includes all the pointers indicated by its remembered set as well as any pointers from generations younger than g. These last could be found by scanning the younger generations. However, since scanning costs are on a par with scavenging, we elect to scavenge them instead. That is, initiating a scavenge of g causes g and all generations younger than g to be scavenged as a unit.

3.1 Managing the persistent area

The persistent area is subdivided into *chunks*—these are the unit of reclamation in the area. Chunks exist in one of four states (see Figure 4):

Live: persistent objects are being copy swizzled into the chunk, with allocation using a free pointer and limit check.

Closed: the chunk is no longer available for allocation, although there may still be pointers into it.

Evacuated: all objects in the chunk have been relocated and indirect blocks left in their place; there may still be pointers into the chunk.

[3] Our remembered sets record the memory addresses of pointers that (may) refer to objects in g.

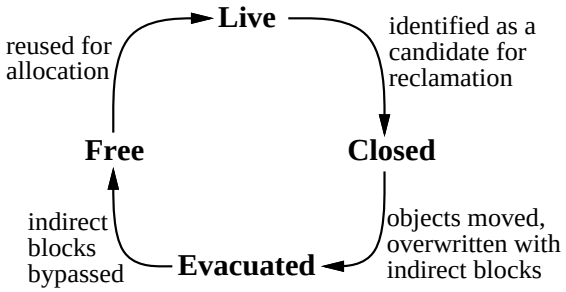

Figure 4: Life cycle of chunks in the persistent area

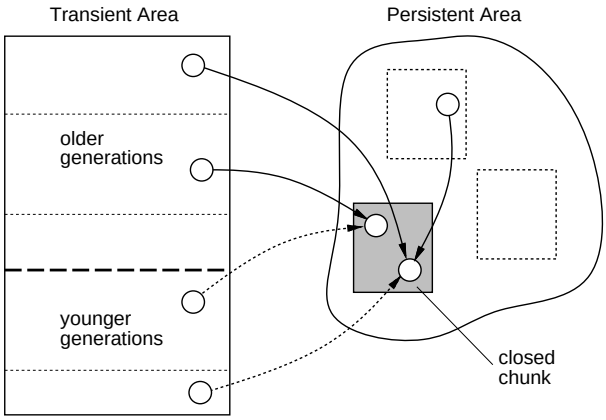

Figure 5: Chunk is closed

Free: the indirect blocks have been bypassed so there are no pointers into the chunk, allowing it to be reclaimed.

When a persistent object is to be copy swizzled, space for the copy is found in some live chunk. If no live chunk exists with sufficient free space then a free chunk is made live. When the buffer manager indicates the need (or opportunity) to reclaim some live chunk, it identifies a candidate chunk for reclamation and marks it as closed, so that no more swizzled persistent objects are allocated in it (Figure 5).

The closed chunk is then scanned to evacuate each of its objects, leaving indirect blocks in their place. Evacuating an object can be achieved in one of two ways: the object can be copied to another live chunk, or the object can be unswizzled into the secondary storage manager's buffers, thence to be written back to disk. In the first case, the indirect block will point to the copied object. In the second case, the indirect block is set to point to a fault block for the persistent object. If the buffer manager made a good choice of candidate chunk for reclamation then there are unlikely to be many pointers into the chunk, and indirectly to the fault blocks. For this reason, the fault blocks are allocated in the youngest generation of the transient area where they will quickly be reclaimed. Note that there may still be pointers into an evacuated chunk from other parts of memory, but

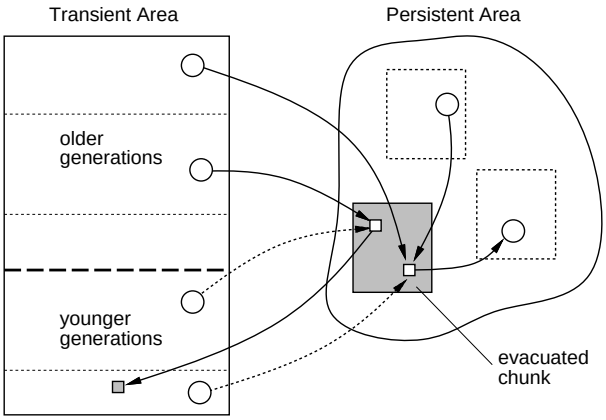

Figure 6: Evacuated chunk: contains only indirect blocks

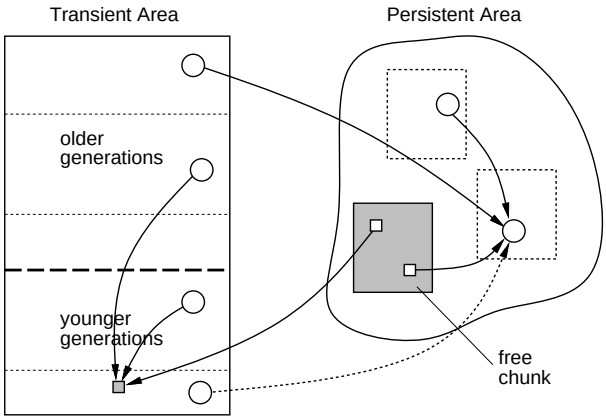

Figure 7: Free chunk: indirect blocks bypassed

they will all point at indirect blocks (Figure 6).

In order to free an evacuated chunk we must ensure that there are no pointers into it—i.e., that all the indirect blocks have been bypassed (Figure 7). To help achieve this we maintain a remembered set for every chunk to track pointers into the chunk. Since there are likely to be many more pointers from young transient objects to (old) persistent objects we do not track pointers from transient generations that are younger than some threshold. This threshold may vary subject to the policy preferences of each application. Remembered pointers into the chunk can be updated to bypass the indirect blocks. To eliminate any pointers into the chunk from generations below the threshold we need only wait until those generations have each been scavenged at least once—recall that we arrange for the garbage collector to bypass all indirect blocks it comes across. If the chunk must be freed immediately then the scavenge can be initiated immediately. Remembered set information can also be used by the buffer manager to determine which chunk to target for reclamation—the chunk with the smallest remembered set is a likely candidate.

4 Summary and Conclusions

We have described a scheme for managing main memory storage of temporary and persistent objects in languages that provide the abstraction of a large persistent heap of objects. Such a scheme must satisfy a number of requirements. Firstly, for performance, run-time garbage collection should be performed only on transient objects. Secondly, buffering of persistent objects requires the ability to move objects within memory. Finally, the scheme must support accurate discovery of all pointers to any given object, so that they may be updated when the object moves.

References

[1] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, Nov. 1983.

[2] M. P. Atkinson and R. Morrison. Procedures as persistent data objects. *ACM Trans. Prog. Lang. Syst.*, 7(4):539–559, Oct. 1985.

[3] J. F. Bartlett. Compacting garbage collection with ambiguous roots. Research Report 88/2, Western Research Laboratory, Digital Equipment Corporation, Feb. 1988.

[4] H.-J. Boehm and M. Weiser. Garbage collection in an uncooperative environment. *Software: Practice and Experience*, 18(9):807–820, Sept. 1988.

[5] L. Cardelli, J. Donahue, L. Glassman, M. Jordan, B. Kalsow, and G. Nelson. Modula-3 report (revised). Technical Report DEC SRC 52, DEC Systems Research Center/Olivetti Research Center, Palo Alto/Menlo Park, CA, Nov. 1989.

[6] A. Goldberg and D. Robson. *Smalltalk-80: The Language and its Implementation.* Addison-Wesley, 1983.

[7] A. L. Hosking and J. E. B. Moss. Towards compile-time optimisations for persistence. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 17–27, Martha's Vineyard, Massachusetts, Sept. 1990. Published as *Implementing Persistent Object Bases: Principles and Practice*, Morgan Kaufmann, 1990. Also available as COINS Technical Report 90-74, University of Massachusetts.

[8] A. L. Hosking and J. E. B. Moss. Compiler support for persistent programming. COINS Technical Report 91-25, University of Massachusetts, Amherst, MA 01003, Mar. 1991.

[9] A. L. Hosking, J. E. B. Moss, and C. Bliss. Design of an object faulting persistent Smalltalk. COINS Technical Report 90-45, University of Massachusetts, Amherst, MA 01003, May 1990.

[10] R. L. Hudson, J. E. B. Moss, A. Diwan, and C. F. Weight. A language-independent garbage collector toolkit. COINS Technical Report 91-47, University of Massachusetts, Amherst, Sept. 1991. Submitted for publication.

[11] T. Kaehler and G. Krasner. LOOM—large object-oriented memory for Smalltalk-80 systems. In G. Krasner, editor, *Smalltalk-80: Bits of History, Words of Advice*, chapter 14, pages 251–270. Addison-Wesley, 1983.

[12] J. E. B. Moss. Design of the Mneme persistent object store. *ACM Trans. Inf. Syst.*, 8(2):103–139, Apr. 1990.

[13] J. E. B. Moss. Working with persistent objects: To swizzle or not to swizzle. COINS Technical Report 90-38, University of Massachusetts, Amherst, MA 01003, May 1990. Submitted for publication.

[14] D. 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*, pages 157–167, Pittsburgh, Pennsylvania, Apr. 1984. *ACM SIGPLAN Not. 19*, 5 (May 1984).