Atomic Garbage Collection: Managing a Stable Heap

Elliot Kolodner, Barbara Liskov, and William Weihl

Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, MA 02139

Abstract: Modern database systems use transactions to achieve a high degree of fault-tolerance. Many modern programming languages and systems provide garbage collected heap storage, which frees the programmer from the job of explicitly deallocating storage. In this paper we describe integrated garbage collection and recovery algorithms for managing a *stable heap* in which accessible objects survive both system crashes and media failures.

A garbage collector typically both moves and modifies objects which can lead to problems when the heap is stable because a system crash after the start of collection but before enough of the reorganized heap reaches the disk can leave the disk in an inconsistent state. Furthermore, collection has to be coordinated with the recovery system. We present a collection algorithm and recovery system that solves these problems.

1 Introduction

A recent trend in database systems and programming languages has been to combine the features of the two types of systems. Modern database systems use transactions to achieve a high degree of fault-tolerance. Many modern programming languages and systems provide garbage collected heap storage, which frees the programmer from the job of explicitly deallocating storage. Relatively few systems, however, support both fault-tolerance and garbage collection. In this paper we describe integrated garbage collection and recovery algorithms for managing a *stable heap* in which accessible objects survive both system crashes and media failures.

Database recovery systems make it easier to write fault-tolerant programs, since a large class of errors is handled automatically by the system. Similarly, automatic heap management enhances reliability, since errors due to explicit deallocation (e.g., dangling references and storage leaks) cannot occur. When the two are combined, reliable programs should be even easier to write. Stable heap management could be useful in object-oriented database systems (e.g., [3, 19, 29, 30]) and in programming systems with persistent storage (e.g., [1, 2, 17]).

Our model of a stable heap is essentially a single-level store. Objects reside in a heap, with a designated set of root objects. The heap is stored on disk, with pages brought into primary memory as needed. Not all objects are treated as stable; instead the set of roots is partitioned into stable and volatile roots, and an object is stable only if it is accessible from one of the stable roots.

Computations in our system run as atomic transactions [11]. Objects are created and modified by transactions; an object becomes stable if a pointer to it is placed in an existing stable object by a transaction that commits. Storage for an object is reclaimed automatically by the system when the object is no longer accessible from any of the roots. In addition, a recovery system ensures that stable objects survive failures: modifications performed by aborted transactions are undone, while modifications performed by committed transactions are guaranteed to survive both system crashes and media failures.

Garbage collection of a stable heap causes problems because a garbage collector typically moves and modifies objects. Objects are moved during compaction to improve locality of reference and reduce fragmentation, thus improving paging performance. Most collection methods also modify objects in an effort to reduce the amount of additional storage needed by the collector itself.

The movement and modification of objects during garbage collection leads to two problems. First, a system crash after the start of collection but before enough of the reorganized heap reaches the disk can leave the disk in an inconsistent state, making it impossible to recover using the usual algorithms for recovery from a system crash. Second, collecting a stable heap requires coordination with the recovery system: objects have to be found on disk by the recovery system even though their locations change during a collection, and some of the roots for collection might be in information managed by the recovery system.

A collection algorithm that solves these problems is called an *atomic garbage collector*. We present an algorithm for atomic collection with the following features: it is based on copying collection, it requires no extra storage beyond the storage required for copying collection, and collections can occur while transactions are in progress. We also discuss how the recovery system must be changed to coordinate with the garbage collector.

With minor modifications, the algorithms in this paper can be used in systems with nested and distributed transactions, such as Argus [17] and Camelot [25]. (In fact, they are based on algorithms originally designed for the Argus system [14].) To simplify our presentation, however, we assume in the rest of the paper that transactions are not nested and that the system consists of a single site.

This research was supported in part by the National Science Foundation under Grant CCR-8716884 and Grant DCR-8503662, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-83-K-0125.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.
© 1989 ACM 0-89791-317-5/89/0005/0015 $1.50

There are several other papers on providing fault-tolerant heap storage. Earlier work on Argus [23] treats all crashes as media failures; as a result, recovery from a system failure is relatively slow, particularly for large databases. PS-Algol [2] uses a less general transaction model, and also does not permit garbage collection to occur while transactions are in progress. Other work (e.g., [21, 27]) does not incorporate any notion of transactions and does not provide recovery from media failures.

The next section describes the system model and introduces background material. Section 3 describes our atomic garbage collection algorithm. Section 4 describes the interactions with the recovery system. In Sections 3 and 4 we assume that all objects are stable; in Section 5 we relax this assumption, and describe how storage is managed for a mixed heap containing both stable and volatile objects. In the last section we summarize our results and discuss future work.

2 System Model

The context for our algorithms is a system in which computations run as atomic transactions and storage consists of a stable heap. Transactions are *serializable* and *total*. Serializability means that when transactions are executed concurrently, the effect will be as if they were run sequentially in some order. Totality means that a transaction is all or nothing, i.e., it completes entirely and *commits* or it *aborts* and is guaranteed to have no effect. A single transaction can create, modify and observe multiple objects.

The stable heap consists of a set of roots and all the objects accessible from them. Some roots are stable and the rest are volatile. The *stable state* is the part of the heap that must survive crashes; it consists of all objects accessible from the stable roots. The *volatile state* does not survive crashes; it consists of all objects accessible from the volatile roots that are not also accessible from a stable root, e.g., local objects of procedure invocations, objects created by actions that have not yet completed, and global objects that do not have to survive crashes.

The programmer sees one heap containing both stable and volatile objects. He can store pointers to stable objects in volatile objects, and can make volatile objects stable by storing pointers to them in an object that is already stable. An object becomes stable when the transaction that makes it accessible from a previously stable object commits. Transactions share a single address space that contains all objects, both shared global objects and objects local to a single transaction. Thus, the programmer does not need to deal with moving objects between secondary storage and a transaction's local memory, or with distinguishing between local and global objects.

The recovery system handles recovery from transaction aborts and from crashes. It maintains information, typically organized as a log, to undo the effects of aborted transactions and to redo the effects of committed transactions. While the system is up and running, the stable heap resides in virtual memory. We view virtual memory as using main memory as a cache for a slower backing store on disk. Main memory is volatile, so the recovery system has to control the movement of pages between main memory and disk to ensure that the information on the disk together with the log can be used to recover after a crash. The disk is non-volatile but not stable, so the recovery system must also maintain redo information on a stable storage device. A stable storage device [15] avoids the loss of information despite failure with very high probability. Typically a recovery system keeps its log on stable storage to avoid storing the redo information more than once.

2.1 Failure Model

There are two kinds of crashes: system crashes and media failure. System crashes occur as a result of a software failure (e.g., inconsistent data structures in the operating system) or hardware failure (e.g., power failures). We assume that when the system crashes, bad information is not written to the disk. Main memory is lost in a system crash; the disk and log survive. The recovery system uses the disk together with the log to recover the stable state in virtual memory.

A media failure occurs when information on disk is lost. For simplicity we assume that no usable information on disk survives a media failure. Only the log on stable storage survives. The recovery system recovers the entire stable state from the log.

2.2 Assumptions

Several assumptions are made in this paper about the hardware and operating system of the machine for which the recovery system is designed.

The design is for a standard architecture, a general purpose register machine with virtual memory (e.g., a VAX[1]). No special-purpose hardware to support the recovery system or garbage collection is assumed.

We assume that the operating system provides primitives that give a program control over the paging of its virtual memory. Primitives are needed to pin and unpin pages of virtual memory, and to tell the system to write a specific page of virtual memory to the backing store. A pinned page cannot be written to its place on the backing store until it is unpinned. Pinning primitives are used for buffer management by database systems [12] and in other transaction systems that tie recovery to virtual memory [9, 26].

We also assume that the operating system preserves the backing store for virtual memory after a crash and allows it to be accessed. Some operating systems, e.g., Multics [7] and Mach [24], satisfy this requirement by allowing files to be directly mapped into the virtual address space of a process. Such a file could be used for the backing store of a stable heap.

Finally we make some technical assumptions about the implementation of objects. Objects consist of a *descriptor* and a *body*. The descriptor identifies the type and the length of the object; the body contains the object's value including pointers to other objects. We assume that the descriptor is large enough to contain a pointer, that it does not span pages of virtual memory, and the collector can distinguish a valid descriptor value from a pointer. One bit of the descriptor can be used to distinguish it from a pointer. Note that there is no restriction on the size of object bodies. These assumptions are common for heaps containing variable-sized objects implemented on standard architectures.

[1] VAX is a trademark of Digital Equipment Corporation

3 Atomic Garbage Collection

The garbage collection of a stable heap causes the problems discussed in the introduction: a crash after the start of collection but before enough of the reorganized heap reaches the backing store leaves the backing store in an inconsistent state, and collection requires coordination with the recovery system. A collection algorithm that solves the crash problem and is coordinated with the recovery system is called an *atomic garbage collector*. In this section we deal with the crash problem; in Section 4 we will discuss how to coordinate collections with recovery.

Garbage collection techniques can be classified according to whether they are real-time or stop-the-world. A real-time collector works in parallel with the program using the heap: steps of the collection are interleaved with program steps. A stop-the-world collector runs when a program needs to reclaim storage in its heap. While a stop-the-world collector is working, the program is suspended.

In this paper we consider only stop-the-world collection. Making a stop-the-world collector atomic seems easier than making a real-time collector atomic. Furthermore, a stop-the-world collector is reasonable for systems with small heaps and no hard real-time constraints. A real-time collector is needed for large heaps, since otherwise the delays associated with collection are prohibitive. We are currently studying how to make real-time collection atomic.

We consider only copying collection [20, 10, 4]. Copying collection is better suited to virtual memory than other collection techniques such as mark, sweep, and compact [5]. It requires fewer traversals of the accessible objects; each traversal involves extra paging overhead. Copying collection also increases locality of reference by moving objects that access each other closer together.

In this section we make the simplifying assumption that the heap contains only stable objects. We begin with a brief review of copying collection and show how a crash during collection might leave the backing store in an inconsistent state. Then we present an atomic collector based on copying collection that solves the crash problem.

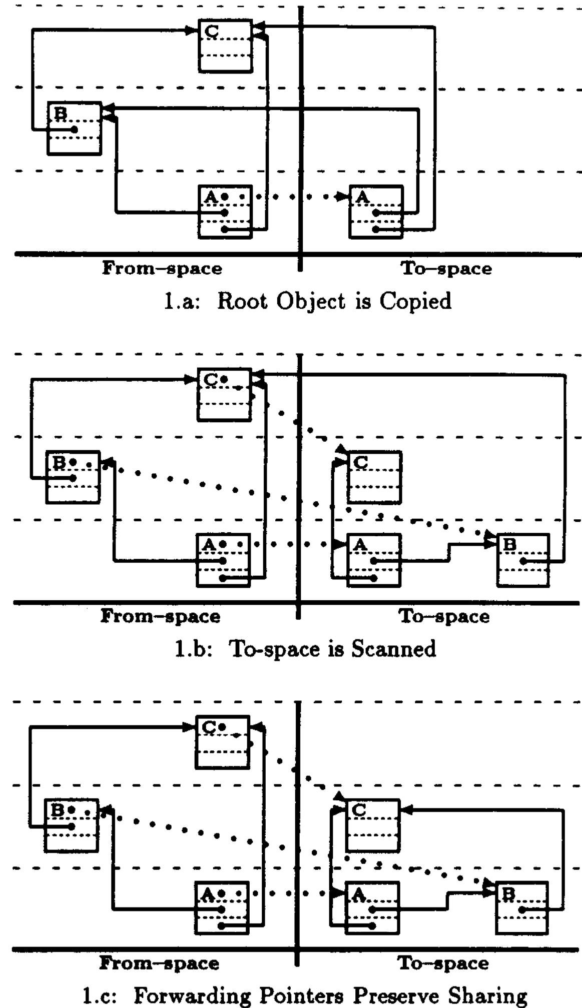


Figure 1: Example of Copying Garbage Collection

3.1 Copying Garbage Collection

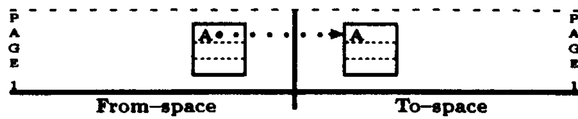
In a system that uses copying garbage collection, the address space is divided into two semispaces: *from-space* and *to-space* [10, 4]. The program allocates all new objects in from-space until the memory in from-space is exhausted or the paging behavior of the program needs to be improved. At that point, a collection is initiated.

During a collection, all accessible objects are copied from from-space to to-space. As each object is copied, a forwarding pointer is inserted in its from-space copy. The purpose of forwarding pointers is to preserve sharing in the object structure and prevent an object from being copied more than once into to-space.

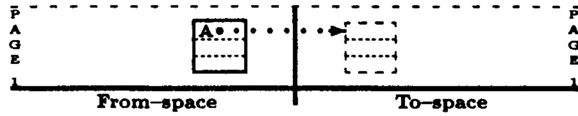
The collector uses two pointers to to-space for bookkeeping during the collection. The *allocation pointer* points to the next to-space cell to be allocated. The *scan pointer* points to the next to-space cell to scan. At the beginning of a collection the allocation and scan pointers both point to the beginning of to-space.

The first step of a collection is to copy the root objects to to-space. Then to-space is scanned sequentially for pointers into from-space. As each such pointer is found, it is dereferenced to find the from-space object it references. If the from-space object has a forwarding pointer, that object has already been copied to to-space, so the pointer in to-space is changed to point to the to-space copy. If there is no forwarding pointer, the object is copied to to-space, a forwarding pointer is left behind, and the pointer in to-space is updated to point to the copy. A collection ends when the scan pointer is equal to the allocation pointer. At that point all of to-space has been scanned, and all of the accessible objects have been copied to and compacted in to-space. Then the roles of from-space and to-space are reversed and the program proceeds.

An example of copying collection can be seen in Figure 1. In Figure 1.a object A, the root object, is copied to to-space. A forwarding pointer is placed in the from-space copy of object A. Then to-space is scanned sequentially for pointers into from-space. Pointers to objects B and C are found in object A. Objects B and C are copied to to-space to the next free locations in Figure 1.b. As the sequential scan of to-space continues, a pointer to object C is found in object B. The forwarding pointer in object C indicates that it has already been copied, so object B is updated to point to object C in Figure 1.c.



2.a: Virtual Memory Just Before Crash



2.b: Backing Store After Crash

Figure 2: Lost Object Descriptor

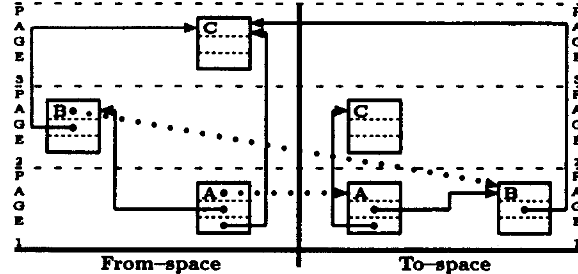


Figure 3: Lost Forwarding Pointer

3.2 Crashes

To understand why copying collection is not atomic, consider what happens if a crash occurs in the middle of a collection. The pages of virtual memory are paged in and out of the physical memory according to the way the graph of accessible objects is traced and copied. A crash can easily leave the contents of the backing store in an inconsistent state. The following two examples show the two kinds of problems that can occur as a result of a crash in the midst of a collection and that need to be prevented by an atomic collector.

Figure 2 shows an example in which object descriptors are lost. Figure 2.a shows an object copied from from-space to to-space; a forwarding pointer was placed in the from-space copy. The forwarding pointer takes the place of the object descriptor. The page of from-space on which the old object version resides is then written to disk. Figure 2.b shows what happens if the system crashes before the new version in to-space reaches the disk. The backing store will not contain a valid version of the object after the crash. The object descriptor of the object has been overwritten with the forwarding pointer, and is not available on the backing store for from-space. Neither is it available on the backing store for to-space.

The second example, illustrated in Figure 3, shows why the pointers on the backing store for to-space cannot be used for recovery after a crash. If an object is copied, but its forwarding pointer does not survive the crash, then recovering on the basis of information already copied to to-space would not preserve the sharing in the graph of accessible objects. Suppose the collection illustrated in Figure 1 were interrupted by a crash after objects A, B, and C had been copied to to-space, but before the pointer to object C from object B had been replaced by a to-space pointer. Figure 1.b shows virtual memory just before the crash. Figure 3 shows a possible state of the backing store after the crash. The crash occurred after pages 1 and 2 of from-space and pages 1 and 2 of to-space had been written to the backing store, but before page 3 of from-space had been written. The forwarding pointer for object C has been lost even though the object has already been copied to to-space.

3.3 Solution

As illustrated in the last section, a crash after the start of collection but before enough of the reorganized heap reaches disk can leave the disk in an inconsistent state. Our approach is to ensure that only crashes that occur when a collection is in progress can cause inconsistencies of this sort. The normal recovery algorithms are used if a crash occurs during normal operation. A special recovery algorithm is used if a crash occurs in the middle of a collection.

We can tell after a crash whether the crash occurred during a collection by keeping a flag on non-volatile storage. The flag can be set whenever a collection starts, and reset when the collection completes. If the flag is true after a crash, the crash occurred during a collection. To ensure that only crashes during a collection can cause problems, we require all dirty pages of virtual memory that contain accessible stable objects to be written to the backing store before the flag is reset. This ensures that the addresses and descriptors in the heap on disk are once again consistent.

Using this approach, the simplest method for atomic garbage collection would be to treat a crash during a collection as a media failure. Then any algorithm could be used for collection. However, the relative cost of recovery from a crash during a collection compared to the cost of recovery at other times would be quite high. Recovery from a media failure requires that the whole log be scanned, whereas the system is tuned to make recovery from system crashes as short as possible. The expected cost for recovery would depend on the fraction of time spent collecting. If that fraction were low enough, then the simple method might be acceptable. Otherwise, an atomic collector that allows fast recovery from a system crash needs to be devised. For such a collector to be viable, the extra costs it imposes for collection need to be kept to a minimum. The remainder of this section deals with such a collector.

3.3.1 Possible Approaches

This section presents a progression of ideas each of which makes copying collection atomic. These ideas simplify the explanation of the actual algorithm and show that the algorithm is correct. The motivation for the ideas is: since the object descriptors are the only information overwritten in from-space during copying collection, reconstructing the object descriptors repairs the inconsistencies caused by collection.

The first idea is to allocate an extra cell per object to hold the forwarding pointer so that no essential information is overwritten during a collection. After a crash no work is required to restore the object descriptors. The cost of this method is the extra cell per object. For a heap with many small objects, this could be a large space overhead. Furthermore, the density of objects per page is decreased, which increases the virtual memory working set.

The extra cell can be avoided at the cost of extra computation. A write-ahead log [11] can be used to record changes to from-space as object descriptors are overwritten with forwarding pointers. The log is an undo log. For each object copied,

a pair of values is entered in the log, the first giving a from-space address of the descriptor and the second, the original contents of the descriptor. Write-ahead logging requires that the from-space page on which the object descriptor resides be pinned in physical memory until the log record recording the change is written to the log. The write-ahead log can simply be recorded on disk rather than stable storage, since its storage need not be any more fault-tolerant than the non-volatile memory used for the backing store.

After a crash during a collection, the backing store for to-space is discarded. To recover, the undo records in the log are used to restore the object descriptors in from-space. This restores from-space to its state just before the collection.

The write-ahead log requires no extra storage in virtual memory to make copying collection atomic. However, extra disk storage is still required for the log. In addition, time is spent pinning pages, copying descriptors into the log, and writing the log to disk.

The final idea reduces the extra storage and time required to make copying collection atomic. It is based on the following observation: ***to-space can be used as a write-ahead log for from-space***. Instead of recording descriptors in a log during collection, we use the copies of the descriptors written in to-space. Thus, we eliminate the space cost of the log, and the time spent constructing log records and writing the log to disk. The details are presented below.

3.3.2 Atomic Copying Garbage Collection

The basic step of a copying collector involves copying an object from from-space to to-space. This is the step in which our atomic copying collection differs from plain copying collection. We describe the copying step in detail, and then present the full algorithm, including recovery from crashes that occur during a collection.

Copying Step. The copying step of our atomic collector works as follows. First, the page in from-space on which the object to be copied resides is pinned in physical memory. Then, the object is copied to to-space and a forwarding pointer is put in the object in from-space, overwriting its descriptor. The from-space page of a copied object is unpinned after the to-space page to which the object was copied reaches the disk.

The copying step uses the write-ahead log principle; to-space is being used as a write-ahead log for from-space. Pinning the from-space page prevents the problem of lost object descriptors, ensuring that there is always a valid copy of an object's descriptor on the backing store: if a from-space page on disk contains a forwarding pointer for an object, the to-space copy of the object will also be on disk and will contain a valid descriptor.

The Collection Algorithm. Now we describe the full collection algorithm.

1. Write all dirty pages of from-space on which stable objects reside to disk.
2. Record that garbage collection is in progress.
3. Use copying collection substituting the copying step described above.
4. Record bookkeeping information required by the recovery system.
5. Write all dirty pages of to-space to the disk.
6. Record that the collection has completed, and reverse the roles of from-space and to-space.

The first step ensures that all stable objects are in a consistent state on the backing store for from-space at the outset of the collection. This reduces the interaction between the garbage collector and the recovery system. If the step were omitted, the recovery system would need to reconstruct the portions of from-space that had not reached the backing store before the beginning of the collection before it could undo the damage caused by the collection.

In step 2 the indication that a collection is in progress can be recorded on any medium that survives system crashes, including the backing store of virtual memory. This indication must be physically recorded on its medium before the collection begins. In the event of a crash, this notifies the system that it needs to use its special algorithm for recovery during a collection.

The bookkeeping information in step 4 coordinates the collector with the recovery system. It relates virtual addresses before the collection to the corresponding addresses after the collection. It is discussed in section 4.

Flushing all dirty pages of to-space to the backing store in step 5 ensures that all stable objects are in a consistent state on the backing store for to-space. Note that flushing is not required for dirty from-space pages; from-space pages are not needed by the recovery system once the collection completes.

The Crash Recovery Algorithm. The simplest way to recover from a crash that occurs during a collection would be to use to-space as an undo log. To do this, the recovery algorithm traverses the stable state in from-space starting with the stable root. During the traversal, every time a forwarding pointer is encountered in place of a descriptor, it is dereferenced to retrieve the descriptor from to-space, and the original descriptor is returned to from-space. When the whole stable state has been traversed, all dirty pages of from-space are written to the backing store, the storage for to-space is released, and the collection is restarted. This solution requires the stable state to be traversed twice, once to restore the descriptors and once to collect. Alternatively, the restoration of object descriptors and the collection could be carried out in one traversal if objects are copied to a fresh copy of to-space.

The observation that copying collection is deterministic is the key to designing a more efficient algorithm. Since the from-space reconstructed using to-space as an undo log is identical to the original from-space, the to-space produced by restarting the collection is identical to the original to-space. This means that the restoration of object descriptors and the copying algorithm can be carried out in one traversal and that the copy of to-space can be reused.

Like the collector, the algorithm for recovery uses two pointers to to-space for bookkeeping: a *reconstruction pointer* and a *scan pointer*. The reconstruction pointer points to the next to-space cell in which an object will be reconstructed. Recovery uses the reconstruction pointer in the same way the collector uses the allocation pointer: when an object is reconstructed in to-space, the reconstruction pointer is increased

according to the object's size. At the beginning of recovery, both pointers point to the beginning of to-space.

Here is the algorithm. First, the stable root is reconstructed in to-space. It is reconstructed on the basis of the contents of the descriptor cell of its from-space copy according to the method described below. Then to-space is scanned sequentially for pointers into from-space starting with the reconstructed stable root. As each pointer to from-space is processed, the object to which it points is reconstructed based on the contents of the cell holding its descriptor in from-space. The restarted collection and the recovery of the stable state is complete when the scan pointer is equal to the reconstruction pointer. At that point all of to-space has been scanned, and all of the accessible stable objects have been reconstructed in to-space.

When processing the pointer to an object in from-space, including the pointer to the stable root, the action taken depends on the contents of the descriptor cell to which the pointer points. There are three cases (only the first two can apply to the stable root):

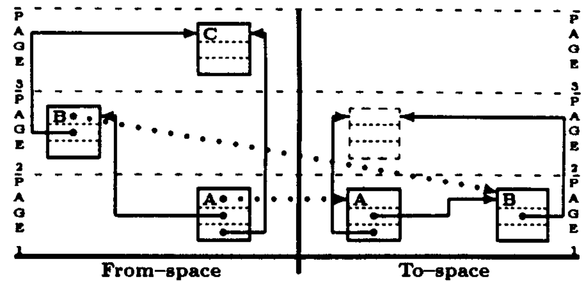
1. The cell contains a descriptor. The object's descriptor was not overwritten on the backing store; use the copying step of the atomic garbage collector to copy the object to to-space.

2. The cell contains a forwarding pointer equal to the reconstruction pointer. Thus, the pointer being processed points to an object that has not yet been reconstructed during recovery. The location of the object in to-space from the original collection and the next location to which the recovery algorithm would copy an object are identical. It is also the location of the object's descriptor in to-space. Reconstruct the object in to-space using its descriptor in to-space and its body from from-space.

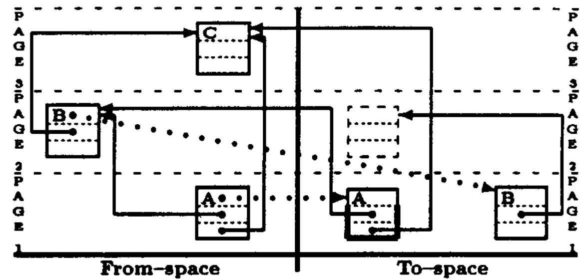
3. The cell contains a forwarding pointer whose value is less than the reconstruction pointer. Thus, the cell is a forwarding pointer to an object that has already been reconstructed during recovery. Replace the from-space pointer by the forwarding pointer.

Suppose the collection pictured earlier in Figure 1 were atomic and interrupted by a crash. Figure 1.c shows a possible state for virtual memory just before the crash. Figure 4 shows how recovery works for this example. Figure 4.a shows a possible state for the backing store after the crash. The crash occurred after pages 1 and 2 of from-space and page 1 of to-space had been written to the backing store, but before page 3 of from-space or page 2 of to-space had been written. Note that the state pictured in Figure 4.a is one that could be produced by the atomic copying collector. Both objects whose descriptors have been overwritten with a forwarding pointer in from-space (A and B) have survived in to-space. The forwarding pointer for object C can not have overwritten the descriptor for object C on the backing store for from-space, because object C has not been written to the backing store for to-space.

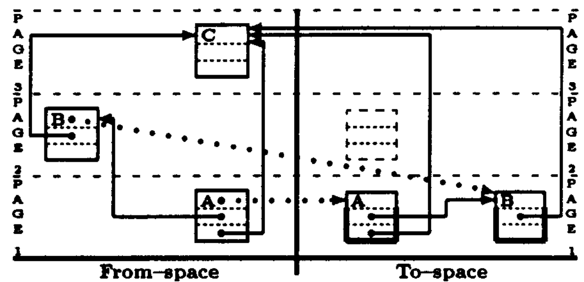
Recovery starts with the reconstruction of the root object in to-space in Figure 4.b. The root object is an example of case 2 from the algorithm; its descriptor in from-space has a forwarding pointer to the next place to which an object would be copied in to-space (the first location of to-space). The descriptor of the root object is taken from to-space, and its body is recopied from from-space to to-space.



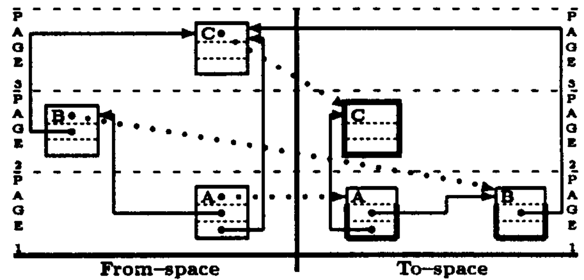
4.a: Backing Store After Crash



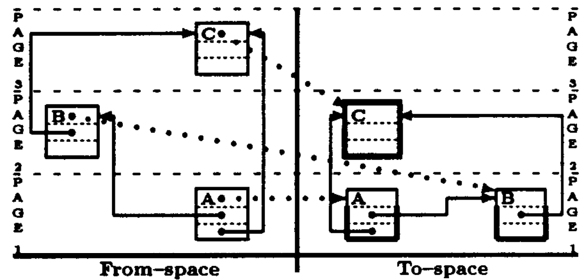
4.b: Root Object is Reconstructed



4.c: Case 2



4.d: Case 1



4.e: Case 3

Figure 4: Recovery Example

Continuing with Figure 4.b, to-space is scanned sequentially for pointers into from-space starting with the newly reconstructed root object. The first such pointer is a pointer to object B in from-space. Object B is another example of case 2 from the algorithm; its descriptor in from-space has a forwarding pointer to the next place to which an object would be copied in to-space. Thus, the object has not yet been reconstructed during recovery. It is reconstructed in to-space using the descriptor from to-space and its body from from-space. The result is pictured in Figure 4.c.

Continuing the scan of to-space for pointers into from-space in Figure 4.c, a pointer to object C is found. Object C is an example of case 1 from the algorithm; the descriptor for object C is on the backing store for from-space. It is copied as is from from-space to to-space. The result is pictured in Figure 4.d.

The next pointer into from-space from to-space is the pointer to object C in object B. This is an example of case 3. Object C is an object that has already been reconstructed by the recovery algorithm; the forwarding pointer in from-space points to an area of to-space that has already been reconstructed. The result is pictured in Figure 4.e, at which point the algorithm completes.

For case 2, a tempting "optimization" would be to recover the body of an object from to-space. This would not work. The to-space body might have been scanned before the crash, after which it would contain pointers to the to-space copies of objects. If the pages containing those to-space copies were not written to disk before the crash, some objects in from-space might never be reconstructed in to-space. This can be seen in Figure 4.a. If the "optimization" were tried on the root object, object C would never be recopied to to-space.

We stated an assumption in Section 2 that object descriptors be big enough to hold a pointer. Our atomic garbage collector depends on that assumption. The forwarding pointer cannot overwrite any memory location in the from-space copy of the object; it can only overwrite a location whose contents could be recovered from to-space after a crash. Object bodies in to-space change as the pointers in the objects are scanned. If part of the object body in from-space were overwritten by a forwarding pointer, that part of the body could not be recovered from to-space after a crash. But object descriptors are not changed in to-space after being copied, so they are an appropriate place for the forwarding pointers.

3.3.3 Discussion

Atomic copying collection is more costly than normal copying collection. Below we discuss the added costs.

The added costs are proportional to the number of stable objects. In the worst case, every object being copied resides on a page of from-space that has not yet been pinned, and there is one page pinned (and unpinned) for every object copied. Thus, pin and unpin operations should be fast. A possible implementation follows. The pin operation sets a bit in the page or frame table; the unpin operation resets the bit. Keeping track of the pinning is a little more complicated. The atomic collector keeps a list of pinned from-space pages for each to-space page to which objects are being copied. To avoid conflicts of unpinning a page of from-space when it is pinned on behalf of more than one to-space page, it also keeps a counter for each pinned from-space page that counts the number of to-space pages for which the from-space page has been pinned. The atomic collector calls a primitive of the virtual memory system to notify the system that a to-space page has to be written to the backing store. The virtual memory system notifies the collector when the page has actually been written by setting a flag. The collector checks the flag when it is ready to write the next page of to-space to the backing store. The collector decrements by one the counter for each from-space page in the list for a to-space page that has reached the backing store. If the counter for a from-space page is 0, the page is unpinned.

We did not specify in our description of the copying step when to-space pages are written to disk. The timing of these writes effects the performance of the algorithm. Each page of to-space could be written to disk when it has been filled with copied objects. However, this is not necessary, and could result in extra disk writes. A better approach would be to delay writing each to-space page to disk until the page has been scanned, since pointers on the page will be updated during scanning. Then each page of to-space would be written to disk just once.

Using this second approach, the atomic collector requires more main memory than the normal collector. With enough memory, the collector can pin all the from-space pages that need to be pinned for a to-space page until the to-space page has been scanned and its pointers updated. If there is too little memory, the atomic collector might need to write a page of to-space more than once. Successive collections increase locality of reference so the amount of extra memory should be small.

Several authors [8, 22, 6] have suggested that the order in which the collector copies objects to to-space affects locality of reference. The atomic collector can copy objects in any order as long as the order is chosen deterministically and the recovery algorithm reconstructs objects in the same order.

4 Coordinating With Recovery

The recovery system handles recovery from transaction aborts and from crashes. It maintains information, typically organized as a log, to undo the effects of aborted transactions and to redo the effects of committed transactions. In this section, we assume for simplicity that this information is stored as *redo* records, which record the new values of objects, and as *undo* records, which store the old values of objects.[2]

This section describes the interactions of atomic garbage collection with the recovery system. There are three areas of interest: the collector must run at a time that does not interfere with the recovery system, it must take care to account for modifications made by active transactions (transactions that have started but have not yet either committed or aborted), and it must inform the recovery system about new addresses for objects that it moves. These issues are discussed below.

The first issue involves the timing of collection. There are times at which a collection cannot occur. Each transaction is a sequence of elementary actions that read and update individual objects. An update action updates an object and its corresponding undo or redo information. A collection in the middle of an update action might see the wrong value for the object or miss a root for collection in the recovery informa-

[2] Distributed systems complicate matters slightly, since the recovery system must cope with transactions that are *prepared* – i.e., in the middle of two-phase commit – after a crash. The discussion below applies with minor modifications to distributed systems; see [14] for details.

tion. This could lead to the inadvertent collection of accessible objects.

To ensure proper timing, we require that the recovery system be in an *action-quiescent* state at the start of collection. The system is action-quiescent when no elementary action is in progress. Since the elementary actions are short, the collection is not significantly delayed. Note that the collector can run while there are active transactions. This is desirable, because waiting for all active transactions to terminate would either take a long time or would disrupt the system (if we forced all active transactions to abort).

The second issue arises because the collector runs while transactions are active: the collector must be careful to account for the modifications made by active transactions to ensure that no objects are lost. For example, suppose a stable object A contains a pointer to object B, and that B is not accessible from any other object. Now suppose that a transaction T modifies A to point instead to some object C. If T aborts, the pointer to B should be restored, while if T commits, the pointer to C should be installed permanently, and B becomes garbage. Suppose a collection takes place after T has modified A, but before it commits or aborts. If T modified A directly, and the collector does not look at the recovery data (the undo record), the storage for B will be reclaimed. If T then aborts, there is no way to restore the heap to its original state. Similarly, if T's modification is kept separately from A (in a redo record) and the collector does not look at this recovery data, the storage for C will be reclaimed; if T commits later, we will have a problem.

To solve this problem, the collector must use the information in undo and redo records for active transactions in determining which objects are accessible. An object must be considered accessible if (1) it is directly accessible from the stable root; (2) it is directly accessible from an undo or redo record for an active transaction that has modified some other accessible object; or (3) it is accessible from some other accessible object. If the recovery information is organized as a log, the records for active transactions are used as roots for collection, in addition to the usual roots. If objects are updated directly, so the latest redo record for an object is identical to the state of the object in the heap, then the redo records can be ignored.

Reading the log during collection to find the records for active transactions can be expensive. We can avoid the expense by storing the recovery records needed by the collector in the heap. For example, the Argus system [17, 18] stores each object as a header that contains pointers to a committed version and a current version; if the transaction that wrote the current version commits, the committed version is replaced by the current version, while if the transaction aborts, the current version is discarded. As the collector traces the graph of objects, it will naturally find all objects accessible from both the old state and the new state of a modified object. Notice that keeping recovery records in the heap introduces additional collection overhead. A recovery record for a transaction will become garbage after the transaction completes, thus requiring more frequent collections of the heap. This problem can be solved by dividing the heap into stable and volatile areas, as discussed in Section 5, and keeping the recovery records in the volatile area.

The third interaction involves the addresses of objects and affects recovery for both system crashes and media failures. The recovery system stores addresses of objects for two reasons: first, if an undo or redo record is recorded for an object, the object's address must also be recorded so the object can be found upon recovery to apply the undo or redo record; second, the undo and redo records themselves may contain pointers to other objects. Since the collector moves objects, some mechanism is needed for translating the addresses maintained by the recovery system when a collection occurs.

Part of the problem has already been solved for system crashes. To solve the timing problem, we required that the system be in an action-quiescent state when collection starts. Recall that the first step of atomic garbage collection is to write all dirty pages of from-space to disk; given that the system is action-quiescent, this is essentially an action-consistent checkpoint [13]. This checkpoint guarantees that no redo records for modifications made before the checkpoint need to be examined after a crash. However, undo records written before the checkpoint may still be needed. We need to record some bookkeeping information to allow the addresses in these undo records to be translated from from-space addresses to the corresponding to-space addresses. This is the bookkeeping information mentioned in the description of atomic garbage collection.

The media failure problem requires more information. After a media failure, the redo records in the log must be used to reconstruct the state of the heap. Because the addresses of objects change during every collection, a single object might have many different addresses associated with it in the log, and two different objects might have the same address at different points in time. Thus, the recovery system must have some way to uniquely identify objects.

We suggest two approaches for dealing with the above problems. The first and simplest approach uses a single mechanism for both system crashes and media failures. It records a complete translation map in the log: whenever the collector copies an object from from-space to to-space, it records the address translation. Notice that this is more than enough bookkeeping information to handle system crashes since all addresses in undo records can be translated using the complete map. In addition, the full address map allows the recovery system to uniquely identify objects after a media failure. The cost for this approach is incurred during collection: a translation record must be written for each accessible object, which may result in substantial overhead.

The second approach uses separate mechanisms for system crashes and media failure. To deal with system crashes we can construct a partial translation map that records the address translations only for the addresses contained in the undo records that might be needed after a crash. In this case, the amount of bookkeeping information needed should be relatively small, and depends on the number of transactions active at the time of the collection. To deal with media failure, we can assign each object a unique identifier (UID), and record that identifier along with the address of the object in log records. After a media failure, the UIDs are used in place of the addresses to reconstruct the object graph. The expense of this approach is incurred during normal operation: writing log records is more expensive. When the value of an object is logged, each pointer contained in it must be augmented with the UID of the object that the pointer refers to.

The basic issue in choosing between the two approaches is a tradeoff between extra time spent during collection and higher overhead during normal operation. Further work is needed to evaluate the trade-offs among these and other approaches.

5 Managing a Mixed Heap

The atomic collection algorithm presented above assumes that the whole heap is stable; therefore every collection has to be atomic. In reality volatile and stable objects coexist on the heap. Furthermore, volatile objects are likely to live shorter lives than stable ones. Since atomic collections are more expensive than normal collections, we would like to avoid the cost of an atomic collection when reclaiming storage used by volatile objects.

This section presents a method for organizing and collecting a mixed heap. The heap is divided into two areas whose address spaces are disjoint: the volatile area and the stable area. All objects are created in the volatile area. Objects that become stable are copied to the stable area at an appropriate time. The volatile area can be collected independently of the stable area and without requiring the collection to be atomic.

Since garbage collection of the volatile area is not atomic, the contents of the volatile area cannot be relied on after a crash. Therefore we assume that the disk storage for the volatile area is discarded after a crash, and recovery is performed using the log and the disk storage for the stable area. *Newly stable objects*, objects that became stable in the volatile area and might not have been copied to the stable area before the crash, must be recovered solely from the log.

Partitioning the heap requires solutions to four problems: determining which modifications to objects in the volatile area need to be logged; deciding which objects to move from the volatile area to the stable area and when to move them; garbage collecting the volatile area; and garbage collecting the stable area. We discuss these four problems below.

5.1 Determining What to Log

Modifications to objects in the stable area should always be logged. However, modifications to objects in the volatile area usually do not need to be logged, since if the objects are not accessible from a stable root they do not need to survive a crash. To minimize the overhead of logging, it is important to avoid logging modifications when possible.

We use a mechanism developed by Oki [23] to decide which objects in the volatile area are accessible from objects in the stable area, and therefore need to have their modifications logged. An object in the volatile area can become accessible from a stable root only if an object that is already accessible from a stable root has been modified to contain a pointer to the object in the volatile area. We maintain an *Accessibility Set* (AS) of objects in the volatile area that are accessible from a stable root. The AS contains at least all objects that are accessible from a stable root by a path of pointers involving only committed versions of objects; a newly accessible object that is accessible only through the new version of an object modified by an active transaction will be added to the AS by the time the transaction commits.

Maintaining the AS requires procedures for determining when an object becomes accessible and must be inserted into the AS, and when an object can be deleted. Objects can become accessible only as a result of a modification to an object that is already accessible. Modifications to accessible objects must be recorded in the log before the modifying transaction commits to ensure that they survive crashes. Thus, potential additions to the AS can be noticed as the modifications are recorded in the log. When a transaction commits its potential additions join the AS. An object ceases being accessible when it is no longer accessible from a stable object. These objects are deleted from the AS by collecting the stable area.

As the potentially stable objects are detected, values are also recorded for them in the log. These log records are specially marked to enable the recovery system to determine after a crash which objects were newly stable.

5.2 Moving Objects to the Stable Area

We could move an object from the volatile area to the stable area as soon as it becomes accessible from a stable object (even though the transaction that made it accessible by modifying a stable object has not yet committed), or we could wait until some later point. Since the transaction that made the object accessible could abort, moving the object as soon as it becomes accessible could cause more garbage to accumulate in the stable area. This will cause atomic garbage collection of the stable area to occur more frequently.

To reduce the amount of garbage in the stable area, we will move an object to the stable area only after the transaction that made it accessible commits. Since moving an object requires copying it and updating any pointers to it, it makes sense to move objects to the stable area when the volatile area is garbage-collected, as discussed in the next section.

We could use the AS to determine which objects are candidates for being moved to the stable area. However, we also need to find and update all pointers in the stable area to these objects when they are moved. Thus, we maintain a separate data structure, the *Stable Pointer Set* (SPS), which contains all objects in the stable area whose committed versions contain pointers to objects in the volatile area. (The SPS could contain other objects as well, but it is guaranteed to contain at least these objects.) Potential additions to the SPS can be collected as an action's modifications to stable objects are recorded in the log. When an action commits, its potential additions join the SPS.

The objects to be moved from the volatile area to the stable area are determined as follows. We scan the committed version of each object in the SPS, looking for pointers to the volatile area. (If an active transaction has modified the object since the transaction that entered it in the SPS committed, it may be necessary to use the undo record for the active transaction to find the committed version of the object.) Each such pointer refers to an object that must be moved. For each object that needs to be moved, we scan its committed version, looking again for pointers to objects in the volatile area. Again, each such pointer refers to an object that must be moved. We continue this process, effectively taking the transitive closure of the objects accessible from the stable area through committed versions.

5.3 Collecting the Volatile Area

The volatile area can be garbage collected at any time. Copying collection is used. The steps taken at a collection are:

1. The stable objects in the volatile area are moved to the stable area. These objects are found as discussed in the previous subsection. As each object to be moved is found, it is copied to the next available location in the stable area. As is usual for copying garbage collection, a forwarding pointer is inserted in the copy in the volatile area of each object that is moved. In addition, the pointer to

the object in the stable object that was used to find it (either an object in the SPS or an object that was already moved) is updated to refer to the object's new location in the stable area. If an object has already been moved when a pointer from a stable object is followed to it, the forwarding pointer left in the object is used to update the pointer in the stable object.

2. The volatile area is garbage collected. The usual roots for garbage collection are augmented by all pointers from new versions of objects in the stable area to objects in the volatile area. These pointers can be found in redo records for active transactions, or be keeping track of a *Modified Object Set* for each transaction, as is done in Argus [23].

Updating pointers in stable objects in step 1 does not require coordination with the recovery system. The values for these objects can be recovered from the log if the system crashes. We discuss recovery briefly in Section 5.5.

Garbage collection of the volatile area is the appropriate time to move newly stable objects from the volatile area to the stable area since they have to be copied anyway. Moving the newly stable objects at an earlier time such as commit would incur a heavy run-time overhead during normal processing: a forwarding pointer would have to be left in every moved object, and every access to an object during normal processing would require a check to make sure that it did not contain a forwarding pointer. This is not a problem at garbage collection because all pointers to the newly stable objects are updated during collection to point directly to the copy made in the stable area.

5.4 Collecting the Stable Area

Two allocation pointers are maintained for the stable area. One is the real allocation pointer that shows exactly where the next object would be allocated. The second ensures that there is enough space in the stable area for all of the newly stable objects in the volatile area: it is updated as potentially stable objects are found when writing to the log; if there is not enough space, the stable area is collected. The stable area may also be collected to improve paging performance.

The collection of the stable area must be atomic. It also requires an accompanying collection of the volatile area. The volatile collection follows the stable collection so that pointers in the volatile area pointing to the stable area are updated correctly. The following steps are taken:

1. The stable objects in the volatile area are moved to the stable area as in step 1 of garbage collection of the volatile area.

2. The stable area is collected using atomic garbage collection.

3. The volatile area is collected as in step 2 of garbage collection of the volatile area. During the collection, references from the volatile area to the stable area are updated using the forwarding pointers in the stable from-space. Objects in the stable from-space that are accessible from the volatile area, but no longer accessible from the stable variables are copied back to the volatile area.

5.5 Discussion

After a crash, recovery is performed using the log and the disk storage for the stable area. Since newly stable objects might have been in the volatile area and not yet copied to the stable area before the crash, these objects must be recovered solely from the log. These objects are recognized during recovery by the mark in their log records. Storage for these objects is reallocated in the stable area during recovery and a map is constructed in the volatile area during recovery to relate their addresses before the crash to their reallocated locations. This map is used when processing the redo records for other objects that contain pointers to them.

The scheme for managing the mixed heap is similar to generation based garbage collection [22, 28, 16]. In those methods the heap is divided according to the age of objects whereas the heap is divided here according to accessibility from a set of stable roots. The method described here for dividing the heap is easily adapted to allow a generation scavenging collector [28] in the volatile area.

6 Conclusions

In this paper we have suggested the integration of transactions and automatic garbage collection. We have presented an algorithm for atomic collection with the following features: it is based on copying collection, it requires no extra storage beyond the storage required for copying collection, and collections can occur while transactions are in progress. We also discussed how the recovery system must be changed to coordinate with the garbage collector and how to manage storage on a heap with both stable and volatile objects.

We are currently working on algorithms for real-time, incremental, and concurrent atomic collection that will be suitable for collecting large stable heaps.

References

[1] A. Albano, L. Cardelli, and R. Orsini. A Strongly Typed Interactive Conceptual Language. *ACM Transactions on Database Systems*, 10(2):230–260, June 1985.

[2] M. P. Atkinson, P. J. Bailey, K. J. Chisholm, P. W. Cockshott, and R. Morrison. An Approach to Persistant Programming. *The Computer Journal*, 26(4):360–365, 1983.

[3] M. Carey, D. DeWitt, J. Richardson, and E. Sheikta. Object and File Management in the EXODUS Extensible Database System. In *Proceedings of the 12th International Conference on Very Large Databases*, August 1986.

[4] C. J. Cheney. A Nonrecursive List Compacting Algorithm. *Communications of the ACM*, 13(11):677–678, November 1970.

[5] Jacques Cohen. Garbage Collection of Linked Data Structures. *Computing Surveys*, 13(3):341–367, September 1981.

[6] Robert Courts. Improving Locality of Reference in a Garbage-collecting Memory Management System. *Communications of the ACM*, 31(9):1128–1138, September 1988.

[7] Robert C. Daley and Jack B. Dennis. Virtual Memory, Processes, and Sharing in MULTICS. *Communications of the ACM*, 11(5):306–312, May 1968.

[8] Jeffrey L. Dawson. Improved Effectiveness from a Real Time Lisp Garbage Collector. In *Proceedings 1982 ACM Symposium on Lisp and Functional Programming*, pages 159–167, 1982.

[9] Jeffrey L. Eppinger and Alfred Z. Spector. *Virtual Memory Management for Recoverable Objects in the TABS Prototype*. Technical Report CMU-CS-85-163, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., December 1985.

[10] Robert R. Fenichel and Jerome C. Yochelson. A LISP Garbage-Collector for Virtual-Memory Systems. *Communications of the ACM*, 12(11):611–612, November 1969.

[11] James N. Gray. *Notes on Database Operating Systems*, pages 393–481. Volume 60 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1978.

[12] James N. Gray, Paul McJones, Mike Blasgen, Bruce Lindsay, Raymond Lorie, Tom Price, Franco Putzolu, and Irving Traiger. The Recovery Manager of the System R Database Manager. *ACM Computing Surveys*, 13(2):223–242, June 1981.

[13] Theo Haerder and Andreas Reuter. Principles of Transaction-Oriented Database Recovery. *ACM Computing Surveys*, 15(4):287–317, December 1983.

[14] Elliot K. Kolodner. *Recovery Using Virtual Memory*. Technical Report MIT/LCS/TR-404, Laboratory for Computer Science, MIT, Cambridge, Ma., July 1987.

[15] Butler. W. Lampson. *Atomic Transactions*, pages 246–265. Volume 105 of *Lecture Notes in Computer Science*, Springer-Verlag, New York, 1981. This is a revised version of Lampson and Sturgis's unpublished *Crash Recovery in a Distributed Data Storage System*.

[16] Henry Lieberman and Carl Hewitt. A Real-Time Garbage Collector Based on the Lifetimes of Objects. *Communications of the ACM*, 26(6):419–429, June 1983.

[17] Barbara Liskov. *Overview of the Argus Language and System*. Programming Methodology Group Memo 40, Laboratory for Computer Science, MIT, Cambridge, Ma., February 1984.

[18] Barbara Liskov, Paul Johnson, and Robert Scheifler. Implementation of Argus. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, November 1987.

[19] David Maier, Jacob Stein, Allen Otis, and Alan Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, pages 472–482, November 1986.

[20] Marvin L. Minsky. *A LISP Garbage Collector Algorithm Using Serial Secondary Storage*. AI Memo 58, MIT AI Lab., October 1963.

[21] Nathaniel Mishkin. *Managing Permanant Objects*. Technical Report YALEU/DCS/RR-338, Department of Computer Science, Yale University, New Haven, Ct., November 1984.

[22] David Moon. Garbage Collection in a Large Lisp System. In *Proc. of the 1984 Symposium on Lisp and Functional Programming*, pages 235–246, 1984.

[23] Brian Oki, Barbara Liskov, and Robert Scheifler. Reliable Object Storage to Support Atomic Actions. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 147–159, December 1985.

[24] Richard F. Rashid. Threads of a New System. *Unix Review*, 4(8):37–49, August 1986.

[25] Alfred Z. Spector, J. J. Bloch, Dean Daniels, R. P. Draves, Daniel Duchamp, Jeffrey L. Eppinger, S. G. Menees, and D. S. Thompson. The camelot project. *Database Engineering*, 9(4), December 1986.

[26] Alfred Z. Spector, Dean Daniels, Daniel Duchamp, Jeffrey L. Eppinger, and Randy Pausch. Distributed Transactions for Reliable Systems. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 127–146, December 1985.

[27] Satish M. Thatte. Persistent Memory: Merging AI-Knowledge and Databases. *Texas Instruments Engineering Journal*, 3(1), January-February 1986.

[28] David Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. In *ACM SIGSOFT/SIGPLAN Practical Programming Environments Conference*, pages 157–167, April 1984.

[29] Daniel Weinreb, Neal Feinberg, Dan Gerson, and Charles Lamb. An Object-Oriented Database System to Support an Integrated Programming Environment. 1988. Submitted for publication.

[30] Stanley Zdonik and Peter Wegner. Language and methodology for object-oriented database environments. In *Proceedings of the 19th Annual Hawaiian Conference on Systems Science*, January 1986.