Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap

Elliot K. Kolodner
IBM Israel Science and Technology
MATAM Advanced Technology Center
Haifa 31905 Israel
email: kolodner@haifasc3.vnet.ibm.com

William E. Weihl
MIT Lab. for Computer Science
545 Technology Square
Cambridge, MA 02139 USA
email: weihl@lcs.mit.edu

Abstract

A *stable heap* is storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. These features enhance reliability and simplify programming by preventing errors due to explicit deallocation, by masking failures and concurrency using transactions, and by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management is useful for programming languages for reliable distributed computing, programming languages with persistent storage, and object-oriented database systems.

Many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. We present garbage collection and recovery algorithms for a stable heap implementation that meet these goals and are appropriate for stock hardware. The collector is incremental: it does not attempt to collect the whole heap at once. The collector is also atomic: it is coordinated with the recovery system to prevent problems when it moves and modifies objects. The time for recovery is independent of heap size, even if a failure occurs during garbage collection.

1 Introduction

A *stable heap* is storage that is managed automatically using garbage collection, manipulated using atomic transactions, and accessed using a uniform storage model. Automatic storage management, used in modern programming languages, enhances reliability by preventing errors due to explicit deallocation (e.g., dangling references and storage leaks). Transactions, used in database and distributed systems, provide fault-tolerance by masking failures that occur while they are running. A uniform storage model simplifies programming by eliminating the distinction between accessing temporary storage and permanent storage. Stable heap management will make it easier to write reliable programs and could be useful in programming languages for reliable distributed computing [10, 26], programming languages with persistent storage [1, 2], and object-oriented database systems [8, 28, 40].

A recovery system provides fault-tolerance for transactions; it manages information that ensures that the effects of successful transactions persist across failures and that unsuccessful transactions have no effect. A garbage collector typically moves and modifies objects as it collects storage that is no longer in use. It moves objects to improve locality of reference and reduce fragmentation; it modifies them in order to speed its work and reduce the amount of additional storage it requires. In a stable heap the movement and modification of objects by the garbage collector may interfere with the work of the recovery system; yet, the recovery system must be able to recover the objects modified by the collector and find the moved ones when a failure occurs. The collector must also have access to the recovery information; objects may be reachable from this information that the collector would not otherwise retain. A collection algorithm that solves these problems and is coordinated correctly with the recovery system is called an *atomic garbage collector*.

Many applications that could benefit from a stable heap (e.g., computer-aided design, computer-aided software engineering, and office information systems) require large amounts of storage, timely responses for transactions, and high availability. This paper presents the design of an integrated atomic garbage collector and recovery system suitable for these applications. The collector is incremental; i.e., it does not attempt to collect the whole heap in one pause. The time for recovery is independent of heap size even if a crash occurs during garbage collection: a crash does not lose all of the work of the collector; rather, recovery can restore the state of an interrupted collection so that incremental collection resumes together with transactions. The design is for a conventional processor with virtual memory; no other special hardware is assumed. We have implemented a stable heap prototype to show the feasibility of our algorithms. The current implementation of Argus [27] serves as the basis for the prototype; we replaced its existing storage management and recovery algorithms.

The reorganization of the heap by a garbage collector is similar to reclustering in an object-oriented database system [18, 28]. Our atomic incremental garbage collector can be used to recluster an object-oriented database on-the-fly without stopping transactions or other system activity.

A previous paper [19] outlines the requirements and issues for atomic garbage collection for a large heap, and outlines the approach on which our current algorithms are based. Our earlier work [21, 20] introduced the notion of a stable heap and presented algorithms suitable for small heaps: the atomic collector is based on a stop and copy collector that suspends transactions until it copies

---

This paper reports on research done by the authors at the Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

The research was supported by the National Science Foundation under grant CCR-8716884, by the Defense Advanced Research Projects Agency (DARPA) under Contract N00014-89-J-1988, and by an equipment grant from Digital Equipment Corporation.

SIGMOD /5/93/Washington, DC,USA
© 1993 ACM 0-89791-592-5/93/0005/0177...$1.50

the whole stable object graph; the recovery system also traverses the whole graph after a crash.

Others have also dealt with the problem of providing fault-tolerant heap storage, but none of their solutions has been satisfactory. Some work (e.g., [7, 34]) provides persistence but not transactions, so it offers less functionality. PS-algol [2] uses a stop-the-world garbage collector and does not permit garbage collection to occur while transactions are in progress. Also, its transaction model is less general and its recovery system imposes a high run-time overhead. Earlier work on Argus [31] uses a normal garbage collector, but treats all crashes as media failures so that recovery from a system failure is slow. Our work grew out of an attempt to design a faster recovery system for Argus.

Detlefs's work [11] is the closest to ours; he has published an algorithm he calls concurrent atomic garbage collection. In his algorithm the pauses for garbage collection are long; each pause requires multiple synchronous random writes to disk. Our algorithm is better integrated with the recovery system; it does not require any synchronous writes to disk.

Here is an overview of the structure of this paper. Section 2 presents our model of a stable heap and failures. Section 3 describes a basic recovery system based on repeating history [29]. Section 4 describes the collector [12] we chose as the basis for our atomic collector and discusses its interactions with the recovery system. In the remainder of the paper we show how to augment the recovery system to support automatic storage management: Section 5 states the invariants maintained by the augmented system; Section 6 shows how to make allocate actions recoverable; Section 7 shows how to make the steps of the garbage collector repeatable; and Section 8 shows the changes to the recovery system necessary to support the collector. Section 9 concludes with an evaluation of the atomic garbage collection algorithm.

2 Background

We abstracted our model of a stable heap from the model of computation used by Argus for local computation at each node in a distributed system. The stable heap model is also appropriate for object-oriented database systems and other persistent programming languages. We begin this section by briefly describing the model. Our previous work has described the model in greater detail [20, 22]. Then we discuss our failure model, and the hardware and operating systems for which our design is appropriate.

2.1 Stable Heap

In our model, computations on shared state run as atomic transactions, which have the usual ACID properties [14] (atomicity, consistency, isolation and durability), and storage is organized as a heap.

A transaction consists of a series of short elementary recoverable actions: a *read* action reads a single object, an *update* action modifies a single object, and an *allocate* action creates a new object. These actions synchronize through logical mutual exclusion locks on objects.

Objects shared among transactions must be *atomic*. Atomic objects provide the synchronization and recovery mechanisms necessary to ensure that transactions are serializable and atomic. The heap synchronizes access to atomic objects using standard read/write locking (i.e., shared for read, exclusive for write). Appropriate techniques can be used to build objects that permit greater concurrency on top of this substrate [16, 36, 38].

A heap consists of a set of root objects and all the objects accessible from them. Objects vary in size and may contain pointers to other objects. In a *stable heap*, some programmer-specified roots are stable; the rest are volatile. The stable roots are global.

The *stable state* is the part of the heap that must survive crashes; it consists of all objects accessible from the stable roots. The objects in the stable state must be atomic. The *volatile state* does not necessarily survive crashes; it consists of all objects that are accessible from the volatile roots, but are not part of the stable state, e.g., objects local to a procedure invocation, objects created by a transaction that has not yet completed, and global objects that do not have to survive crashes.

In the remainder of this paper we assume that all of the roots of the heap are stable and describe algorithms for recovery and atomic garbage collection that work under this assumption. These algorithms are appropriate for an object-oriented transaction system where an object persists because it is created in a persistent area (e.g., Exodus [8]) as well as our stable heap model. In the first author's dissertation [22] we describe additional algorithms necessary to support a heap where some roots may be volatile.

2.2 Failure Model

A recovery system provides fault-tolerance by controlling the movement of data between the levels of a storage hierarchy. We assume a standard hierarchy with four components: (1) main memory, (2) disk, (3) log, and (4) archive [4, 15].

A stable heap keeps its data on disk, which is non-volatile, and uses main memory, which is volatile, as a cache or buffer pool. A buffer manager decides which pages to keep in the cache. The recovery system may constrain the buffer manager by pinning a page in main memory.

The log is a sequential file, usually kept on a stable storage device [23], to which the recovery system writes information that it uses to redo the effects of a committed transaction and to undo the effects of an aborted transaction. The recovery system does not directly write to the log on stable storage; rather, it spools information to a log buffer. When a buffer fills, recovery writes it to disk asynchronously and begins spooling to the next buffer. A well designed recovery system synchronously writes a buffer to stable storage, or *forces* the log, only at transaction commit when it must ensure that the effects of the transaction survive failure.[1] In this paper when we say *write to the log*, we mean spool to the log buffer. If we want to describe a synchronous write, we use the phrase *force the log*. To distinguish the part of the log on stable storage from the part in the log buffer, we call the former the *stable log* and the latter the *volatile log*.

The archive is an out-of-date copy of the database; it may be on disk or some cheaper non-volatile medium such as magnetic tape.

A recovery system deals with three kinds of failure: (1) transaction, (2) system, and (3) media. A transaction fails when it aborts; the recovery system may use information found in main memory, the disk, or the log to ensure that the transaction has no effect. As a result of a system failure, which we also call a crash, main memory is lost, but the disk and stable log survive. The recovery system uses information in the stable log and on the disk to recover the state of the heap. A crash also aborts transactions that are active when it occurs. A media failure occurs when a page or several pages of the disk get corrupted. The recovery system uses the log together with the archive to recover the pages. We do not consider media failure or archives further in this paper.

2.3 Implementation Platform

Our design is for conventional hardware---stock uniprocessors with virtual memory. No special-purpose hardware to support recovery or garbage collection is assumed.

[1] A high performance transaction system uses *group commit* [13] instead of forcing the log for every transaction.

The design requires an operating system that allows a program some control over the virtual memory system. Primitives are needed to control when a page of virtual memory can be written to the backing store and to set protections on pages. The ability to preserve the backing store for virtual memory after a crash is also required. Mach [32] satisfies these requirements.

3 Recovery

Given the storage architecture and failure model described above in Section 2.2 we describe a way to do recovery based on the repeating history paradigm of Mohan, et al. [29]. We chose repeating history because it is simple compared to previous recovery algorithms [4, 15, 25], and easy to optimize. Our recovery algorithms are simpler than Mohan's; they use physical redo and logical undo for all updates, whereas Mohan's allow logical redo for updates to objects totally contained with a single page. We chose physical redo for our algorithms because it is simpler than logical redo.

3.1 Redo Protocol

The key to repeating history is the *redo protocol*, which employs a write-ahead log. The recovery system follows the protocol for all modifications to objects:

1. pin the object's page in main memory;
2. modify the object;
3. spool a log record with redo information (address at which modification occurred plus new value);
4. and once the redo record is in the stable log, unpin the page.

After step 3, the modification is complete and the protocol returns to its invoker; the page is unpinned in the background.

The redo protocol ensures the following property: if a modification is on disk, the redo record describing the modification is in the stable log. The *repeating history invariant*, which simplifies recovery after a crash, follows directly from this property:

Invariant 3.1 (Repeating History) *The disk state that would be produced by applying the stable part of the redo log to the disk (i.e., carrying out each of the redo actions in the order they are recorded in the log) is a state that actually occurred at some previous point in the computation.*[2]

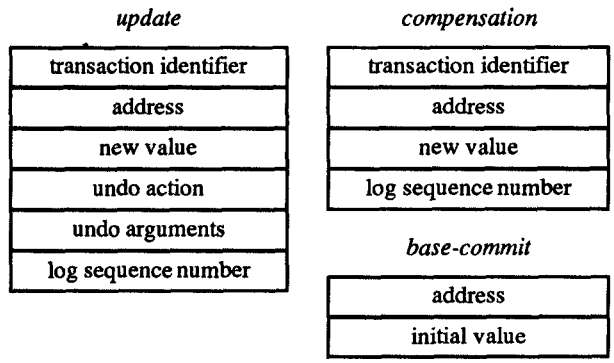
The action of redoing the log is called *repeating history*.

The repeating history invariant simplifies recovery from a system failure (described below in Section 3.2). It is also useful for our atomic garbage collector. In the design of our collector, we depend on the invariant to bring the heap to a state from which the collector can complete its work after a crash.

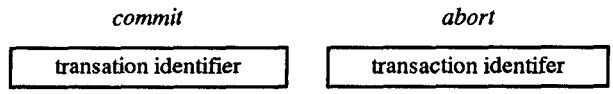
3.2 Basic Recovery Operation

Below we describe a basic recovery system. In the remainder of the paper we show how to augment and change the basic system to support recoverable allocation and atomic garbage collection. Figure 1 illustrates all of the log records written by recovery (including the augmentations).

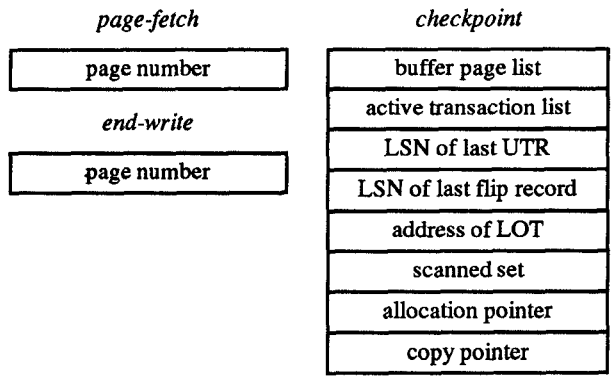
[2]Formally, an invariant is a predicate on state; at first glance this statement of the repeating history invariant may not appear to be such a predicate. However, we can formalize it by defining the redo function determined by the log, and defining a history variable that captures the sequence of states through which the computation passes.



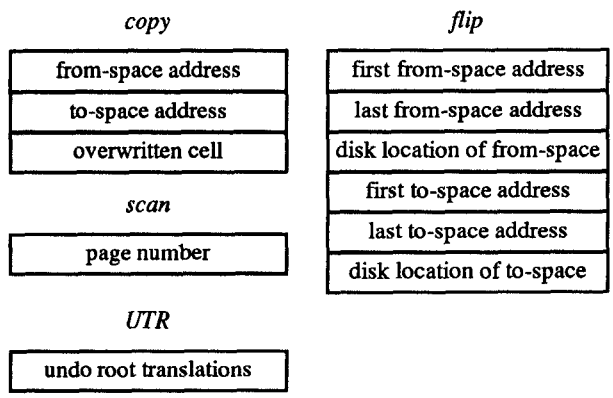
1.a: Update Entries



1.b: Outcome Entries



1.c: Buffer Manager and Checkpoint Entries



1.d: Garbage Collection Entries

Figure 1: Format of Log Entries

An update action follows the write-ahead log protocol and writes an *update record* to the log. The update record contains a transaction identifier, the address of the updated region in the object, a new value for the region, and undo information. The undo information, which is required for transaction abort, is *logical*, the name and arguments of a compensating operation. To facilitate undo, the log records for a transaction are back chained using the log sequence number (LSN) of the previous record for the same transaction.

When a transaction commits, it writes a *commit record* to the log and forces the log buffer to the stable log. The record contains the transaction's identifier.

When a transaction aborts, it undoes its update actions in reverse order by following the backward chain of its update records and invoking the undo operation recorded in each record. Undoing an update is a modification to the heap so abort follows the write-ahead log protocol for each undo operation and writes a redo record describing the operation to the log. The redo record, called a *compensation log record* or CLR, is like an update record except that it has no undo information, and its LSN points to the next update to be undone. The transaction writes an *abort record* containing its identifier when all of its updates have been undone.

To recover from a system failure the recovery system repeats history, i.e., it applies the redo records (update and CLR) in the stable log to the disk. Then it aborts the transactions that were active at the time of the crash.

Two optimizations shorten recovery times after a system failure [29]. First, the buffer manager writes *page-fetch* and *end-write* records to log. Recovery uses these records after a crash to deduce a superset of the pages that need recovery work; the set is kept small by regularly writing dirty pages to disk. Second, at regular intervals the recovery system checkpoints in an action-quiescent state and writes a checkpoint record to the log; these records can be used after a crash to deduce the point in the log at which repeating history must begin. Taking a checkpoint is cheap: it does not require any synchronous writes, and it halts the system for a very brief period. Checkpoint records are described in Section 8.4.

4 Requirements And Issues

Our choice of a garbage collection algorithm on which to base our atomic algorithm is motivated by two requirements: (1) that it be suitable for a large heap, and (2) that it work on stock hardware. Below we describe these requirements, a collector that meets the requirements, and the interactions between the collector and the recovery system.

4.1 Garbage Collection for a Large Heap on Stock Hardware

There are two consequences of the requirement that the collector be suitable for a large heap: (1) the pauses for garbage collection must be short enough to support interactive response times, and (2) the collector must interact well with virtual memory. Two general techniques have been used to shorten pauses: (1) incremental garbage collection [3, 6], and (2) dividing the heap into independently collectible areas [5, 17]. Steps of an incremental collector are interleaved with normal program steps such that the pause due to each incremental step is small. Many incremental collectors have been based on Baker's algorithm [3], which is a copying collector.

A good division of the heap into independently collectible areas leaves few inter-area references and places objects with similar lifetime characteristics into the same area. For programs without persistent storage, one automatic way of dividing the heap without programmer intervention is based on the age of objects; this is called generational collection [24, 30, 35]. Generational collection depends on an observed behavior of program heaps that new objects are more likely to become garbage than old objects. Since persistent heaps do not necessarily conform to this behavior, generational collection does not appear to be appropriate. Recently, Hudson and Moss [17] described a new way to divide a large heap into areas; their algorithm merits further study.

For large heaps implemented in virtual memory, an important purpose of garbage collection is to reorganize the heap to provide good paging performance. Reorganization requires a collector that can move objects, e.g., a copying collector, which can increase locality of reference and reduce paging by moving objects that are referenced together to the same page [9, 30, 39].

The other requirement for our algorithm is that it work well on stock hardware. Without hardware assists, Baker's incremental garbage collector is expensive---it requires a comparison on every heap reference. Ellis, Li and Appel [12] have shown how the virtual memory hardware on stock hardware can be used to facilitate incremental copying garbage collection with lower overhead.

4.2 Ellis's Algorithm

We base our atomic incremental garbage collector on the collector of Ellis, Li and Appel (hereafter attributed to Ellis). Ellis's collector is based on Baker's algorithm [3]. As in other copying collectors, memory is divided into from-space and to-space. In one collection cycle, the collector copies the objects accessible from the roots from from-space to to-space. As each object is copied, a forwarding pointer is inserted in its from-space copy. Forwarding pointers preserve sharing in the object graph.

The program doing useful computation (often called the mutator) and the collector run as coroutines subject to a synchronization constraint that we discuss below. The mutator calls the collector to do some work each time it allocates a new object. When the garbage collector runs, it either *scans* a fixed number of locations in to-space (i.e., it converts from-space pointers in those locations to to-space pointers, copying objects if necessary) or it *flips* (i.e., to-space becomes from-space, a new to-space is allocated, and the collector copies the root objects to the new to-space). The algorithm can be extended in the obvious way to allow the collector and multiple mutators to run in separate threads.

4.2.1 Read Barrier

Synchronization between the mutator and the collector depends on the invariant that the mutator never sees a pointer into from-space. This invariant is established at a flip: the objects referenced by a register, the stack, or an own variable (the root objects) are copied to to-space, and the corresponding references are updated to point to the to-space copies. The invariant is enforced during the collection by the so-called "read barrier". The read barrier prevents the mutator from seeing pointers into from-space.

Ellis suggests a cheap implementation of the read barrier for stock hardware. After the root set is copied to to-space at a flip, the collector uses the virtual memory hardware to protect the unscanned pages of to-space against access. If the mutator accesses an unscanned page, the collector fields the resulting trap and scans the page, translating all from-space addresses on the page to the corresponding to-space addresses. Scanned pages do not contain from-space addresses; the mutator never accesses an unscanned page, so it never sees a from-space address.

Ellis's approach is cheap; it adds little to the overall garbage collection time since there will be at most one trap per page of to-space. However, Ellis's collector might not be as incremental as we would like. The distribution of read barrier traps will be skewed to be very frequent just after a flip and each trap requires that a whole page be scanned. The seriousness of this problem depends on the mutator's rate of access to the heap and locality of reference.

We have built a prototype that will enable us to measure the length and frequency of the pauses attributable to the read barrier.

4.2.2 Scanning An Arbitrary Page

Ellis's algorithm requires the capability of scanning an arbitrary page of to-space. A typical heap implementation constructs objects such that the first cell contains a descriptor giving the object's low-level type, the object's length, and the positions of pointers in the objects. Since objects may cross page boundaries, the collector needs an additional mechanism to scan an arbitrary page. For that purpose it creates a data structure called the Last Object Table [33]. This table is an array indexed by to-space page number; the entry for a page contains the location of the last object on that page. To scan an arbitrary page, the collector finds the last object on the previous page using the table. Then it uses the object descriptors to parse the objects on the page, and find and scan their pointers.

4.3 Interactions Between Garbage Collection And Recovery

An atomic garbage collector interacts with the recovery system such that both the collector and the recovery system are correct. Below we discuss the interactions. Some of the interactions concern correctness; others affect the speed of crash recovery.

4.3.1 Correct Interaction

There are four interactions that must be handled correctly for the collector to be atomic; these were originally described in our report on earlier research [20]. First, copying collectors modify objects. They overwrite parts of objects in from-space with forwarding pointers, copy objects to to-space, and change from-space pointers in the to-space copies to to-space pointers. These modifications must be done recoverably.

Second, copying collectors move objects. The recovery system records values for objects in the log. The value of an object usually contains the names of other objects. Since collectors move objects, simply using virtual addresses as names is not enough. In the event of media failure, the recovery system needs some way of mapping virtual addresses before a collection to the corresponding addresses after a collection. In a recovery system based on redo/undo logging, transaction abort (undo) must also be able to translate addresses.

Third, garbage collection has to be synchronized with transactions. Without synchronization, the collector might process an object in an inconsistent state, miss a pointer in it to a second object, and inadvertently reclaim the second object. This synchronization has to be cheap.

Fourth, for objects being updated by an active transaction, the recovery system keeps track of undo and redo information in its log. This information has to be made available to the collector so that it will not reclaim objects that might still be accessible if the active transaction commits or aborts.

4.3.2 Fast Recovery

A crash may occur toward the end of an incremental atomic garbage collection. Without information on the progress of the collection, the recovery system may have to redo the entire collection. To allow fast crash recovery, the state of the garbage collector must be recoverable. Its state includes allocation pointers, a record of which pages have been scanned, and the Last Object Table.

5 Invariants

To support automatic storage management we augment the basic recovery system with support for allocation and garbage collection. Below we define reachability (and other useful terminology) and then we state the primary invariants that the augmented recovery system maintains in terms of objects reachable from the roots.

The *committed value* of an object is its value just after the commit of the last transaction to update it. If a transaction holds a write lock on an object, the *current value* of the object is the value of the object seen by that transaction. Otherwise, the object's current value is equal to its committed value.

Object A is said to be *directly reachable* from object B, if object B contains a reference to object A. Object A is said to be *reachable* from object B if it is equal to B, it is reachable from an object directly reachable from B's current value, or it is reachable from an object directly reachable from B's committed value. Object A is said to be *commit reachable* from object B if it is equal to B, or it is commit reachable from an object directly reachable from B's committed value.

A transaction is said to be *active in the log* if there are one or more update records for the transaction in the log, but no subsequent commit or abort record.

Invariant 5.1 relates the information in the log to the state of the stable heap in virtual memory.

Invariant 5.1 *The disk state that would be produced by redoing the log (both its stable and volatile parts) against the disk backing store is the same as the current state of the stable heap in virtual memory.*

The recovery system maintains this invariant by spooling a log record every time an object in the stable heap is modified.

Invariant 5.2 tells how the committed value of an object can be restored from the log and the disk during normal operation.

Invariant 5.2 *If an object is reachable from a root, then its committed value is in the disk state that would be produced by redoing the whole log (both its stable and volatile parts) against the disk, and then aborting the transactions that are active in the whole log.*

Invariant 5.3 tells how to recover the stable state after a crash. It holds because Invariant 5.2 holds and transaction commit forces the log.

Invariant 5.3 *If an object is commit reachable from a root, then its committed value is in the disk state that would be produced by redoing the stable log against the disk, and then aborting the transactions that are active in the stable log.*

6 Allocate Action

Recoverability for allocate actions depends on the layout of memory. Both the collector and the mutator (transactions) allocate space for objects in to-space: the collector for the objects it copies and the mutator for the new objects it creates. Since the read barrier ensures that the new objects allocated by the mutator never contain pointers into from-space, these objects do not have to be scanned by the collector. Baker proposed an appropriate layout for to-space, shown in Figure 2, which we adopt. The collector copies objects contiguously to the low part of to-space and allocates by adding to the *copy pointer*. The mutator allocates objects contiguously in the high part of to-space and allocates by subtracting from the *allocation pointer*.

An allocate action subtracts the size of a new object from the allocation pointer, initializes the object, and writes a *base-commit record* to the log. The base-commit record contains the address for the object and the initial value of the object. The action does not follow the write-ahead log protocol, neither for reasons of undo nor for reasons of redo: the newly allocated area for the object is fresh, so no information is overwritten; the object can be accessed after a crash only if the base-commit record is in the stable log; and the allocation pointer can be recovered from the last base-commit record in the stable log.

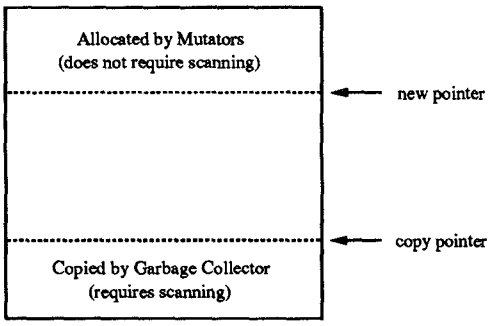


Figure 2: Layout of To-space

7 Atomic Garbage Collection

Our atomic garbage collector runs below the level of user transactions and provides for its own recovery. A user transaction consists of read, update and allocate actions. We add two additional types of actions for the atomic garbage collector: a *copy* action copies a single object from from-space to to-space, and a *scan* action scans a single page of to-space. We make these actions recoverable using the redo protocol. The repeating history invariant guarantees that the state reached by applying the redo log to the disk is an actual state from the history of the system. In particular it is a state from which the garbage collector can continue and complete the collection.

In the remainder of this section we describe the recoverable copy and scan actions. We also show that the information written to the log by these actions solves the problems introduced by the movement of objects, and that it is sufficient to recover the Last Object Table. In Section 8 we deal with the other interactions between the collector and the recovery system.

7.1 Copy Action

A copy action makes two modifications to memory: it inserts a forwarding pointer in an object in from-space, and it copies that object to to-space. Thus, a naive application of the redo protocol would pin at least two pages (the from-space page of the forwarding pointer and the to-space page of the copied object) and it would write two redo records to the log. The redo record describing the modification to to-space would contain the value of the object, so that the whole object graph would be written to the log during the course of a collection. However, by relying on the repeating history invariant, we optimize the copy action so that it pins a single page and writes a small record to the log.

Here is the copy action:

1. pin the from-space page of the object cell that will be overwritten by the forwarding pointer;
2. copy the object to to-space and insert a forwarding pointer in its from-space copy;
3. spool a *copy record* containing the from-space address of the object, the to-space address of its copy, and the contents of the cell overwritten by the forwarding pointer;
4. and once the copy record is in the stable log, unpin the from-space page.

After step 3 the copy action is over and the collector can continue. Figure 1 illustrates the copy record.

First we show that the pinning optimization is correct, i.e., that only a single page has to be pinned. It is easy to ensure that the cell in from-space overwritten by the forwarding pointer resides entirely on a single page---always overwrite the first cell of the object and during allocation make sure that the first cell of an object never crosses page boundaries. Thus, only one from-space page needs to be pinned even if the object spans more than one page.

The to-space pages to which an object is copied do not need to be pinned. The redo protocol pins pages to prevent partial modifications from reaching disk. However, the partial modifications to to-space by the copy action can be masked without pinning pages. According to the allocation scheme outlined Section 6 the collector writes copy records to the log in the order in which it copies objects. Thus, the copy pointer can be recovered after a crash by looking at the last copy record in the stable log. Any partial modifications to to-space by copy actions whose copy records did not reach the log are at addresses greater than the recovered copy pointer. Thus, these modifications will never be observed.

Next we show how to repeat the copy action using the information in the copy record.

1. Re-copy the object from the from-space address recorded in the record to the to-space address.
2. Take the contents of the cell overwritten by the forwarding pointer from the copy record and write it to its place in the to-space copy of the object.
3. Using the from-space address of the object and the to-space address of the object from the copy record, reinsert the forwarding pointer in from-space.

Finally we argue that the repeat of the copy action is correct. There are two parts to the argument: (1) the forwarding pointer in from-space is recovered correctly, and (2) the object in to-space is recovered correctly. The first part is trivial because the redo protocol guarantees correct recovery of the forwarding pointer.

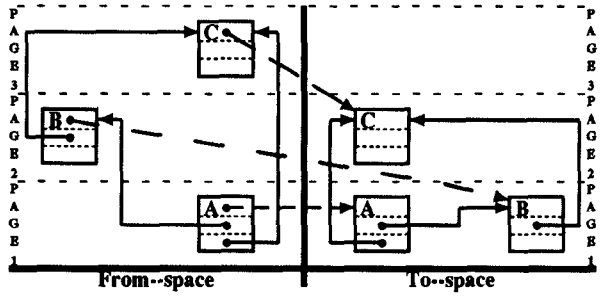
To argue the second part, we rely on the repeating history invariant. After the flip, the only modification to the from-space copy of an object is the insertion of a forwarding pointer by a copy action of the collector. Thus, if we apply the redo log to the disk (i.e., repeat history) up until the point in the log where the flip occurred, the from-space object is recovered to the same state as at the time of the flip except for the cell containing the forwarding pointer. As we continue to repeat history from the time of the flip until the copy record, the from-space object does not change. Thus, when we redo the copy, the correct value of the object is in from-space except for the cell overwritten by the forwarding pointer. We recover the value for this cell from the copy record.

Because of the dependence on from-space for the value of an object that needs to be recopied, disk storage for the from-space object cannot be discarded until the to-space copy has reached disk. This means that the disk storage for from-space must be kept until both (1) all accessible objects have been copied to to-space, and (2) every to-space copy has reached disk. The discussion of the scan action will show that this condition must be made even stronger.

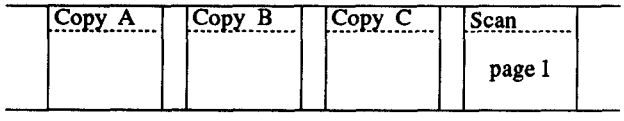
7.2 Scan Action

A scan action scans a unit of to-space, looks for pointers into from-space, and converts them into to-space pointers. The unit could be a single location, an object, or a page. The page is the best unit for two reasons: (1) it is the unit updated on disk atomically, so it is the natural unit of recovery, and (2) it is the unit of synchronization for Ellis's algorithm. Choosing a unit larger than a page would make the collector less incremental.

A scan action is also made recoverable using the redo protocol. The redo protocol prevents the loss of forwarding pointers, i.e., it



3.a: Virtual Memory



3.b: Log Showing Scan Record

Figure 3: The Scan Action

prevents a to-space pointer to a copied object from reaching disk before the forwarding pointer for the object is recoverable. A naive application of the redo protocol writes a redo record containing the entire scanned page and would lead to the entire object graph being written to the log. However, we can optimize the scan ation by writing a scan record whose sole content is the number of the scanned page.

Here is the scan action:

1. pin page to be scanned;
2. scan page and update its from-space pointers to to-space pointers, copying objects as necessary using copy actions;
3. spool a scan record containing the address of the page;
4. and once the scan record is in the stable log, unpin the page.

After step 3 the scan action is over and the collector can continue.

Figure 3 shows the effects of a scan action. Object A is the root object. At the flip, the collector copies A to page 1 of to-space. As a result of the scan action for page 1, objects B and C are also copied to to-space. Figure 3.a shows virtual memory after page 1 has been scanned. Figure 3.b shows the scan record for page 1 in the log. Notice that copy records for objects A, B, and C precede the scan record.

The address of the scanned page is sufficient to make the scan action repeatable. By the time the scan record has been written to the log, there are copy records in the log for all objects referenced by pointers on the scanned page. That means that when the redo log is applied to the disk, all of these objects will be copied to the same place in to-space, and their forwarding pointers will be the same as when they were first copied. Therefore when the page is re-scanned, all of its pointers will be assigned the same values that they were assigned during the first scan.

Since the repeatability of the scan action depends on forwarding pointers in from-space, from-space cannot be discarded until every page of to-space has been scanned, and every scanned page has reached disk. The writing of scanned pages to disk does not need to occur synchronously; rather, the garbage collector informs the buffer page manager as it scans each page. Then the buffer page manager can schedule the writes according to its own policies. The buffer page manager must also provide an operation that allows the collector to check if all of the scanned pages have been written.

7.3 Scanning An Arbitrary Page

To allow the scanning of an arbitrary page of to-space after a crash, the Last Object Table must also be recoverable. Fortunately, the copy records contain sufficient information to recover the table. The collector keeps the table in the stable heap. When repeating history the copy records act as redo records for the table.

7.4 Movement of Objects

The information written to the log by the copy actions also solves the naming problem caused by the movement of objects. Each copy record contains a from-space, to-space address pair for an object. There is one copy record for each accessible object. Unfortunately it is expensive to access this information for undo. In Section 8.2.2, we describe a cheap way to keep track of the translations that the recovery system needs for undo.

7.5 Size of Copy Records

Assume that a pointer takes up four bytes. That means that the overhead for a copy record is greater than 12 bytes, four bytes for each component plus the extra overhead to delineate the record and indicate its type. This is high for small objects such as a cons cell in Lisp or a variant in Argus (both are eight bytes). The importance of this issue depends on two factors: (1) the frequency of small objects compared to larger objects, and (2) the overhead of writing to the log. If there are few small objects, the total space taken up by copy records in the log will be minor. If the overhead for writing is low, the extra overhead for copy records will be tolerable.

A further optimization reduces the log space taken by copy records: a *block copy action* copies a group of objects consecutively to to-space and writes a single *block copy record* to the log. To satisfy the redo protocol, the action pins all of the from-space pages modified by the objects' forwarding pointers until the record is in the stable log. The record contains the number of objects copied, the from-space address and old contents of the cell overwritten by the forwarding pointer for each object copied, and the to-space address of the first object copied. By re-copying the objects in the same order after a crash, recovery can deduce the to-space addresses of the remaining objects. A natural unit of blocking is the set of the objects copied during the scan of a single to-space page; in this case the block copy record also serves as the scan record.

8 Other Interactions With Recovery

In the previous sections we showed how to make allocate, copy and scan actions recoverable. Here we describe the other interactions between garbage collection and recovery: (1) synchronization between the garbage collector, the transaction system and recovery, (2) roots in the recovery information, (3) allocating spaces recoverably, and (4) fast recovery for a system failure during garbage collection.

8.1 Synchronization

Because the collector moves and modifies objects it must be synchronized with the transaction system. Since it writes recovery information to the log, it also has to be synchronized with the parts of the recovery system that write to the log. This synchronization has to be cheap. First we describe synchronization with transactions; then we describe synchronization with other logging.

8.1.1 Synchronization With Transactions

Our approach to synchronization with transactions is to require that a flip occur in an *action-quiescent state*, a state when no elementary action (read, update, allocate, copy or scan) is in progress. Since the elementary actions are short, a flip is not significantly delayed by waiting for an action-quiescent state. Given that a flip occurs in action-quiescent state, the read barrier ensures that a copy or scan action only observes an object in an action-consistent state.

The correctness of the above approach can be seen by viewing the system as a multi-level transaction system with two levels [37]. At the high level, there are user transactions; at the low level, the elementary actions. At the level of transactions, a transaction obtains read and write locks that it holds for its duration. These locks ensure serializability at the transaction level. At the level of actions, a synchronization mechanism ensures that only one action accesses a given object at a time, so that an action always observes an action-consistent object state. Since the garbage collection actions do not change the abstract values of objects, they are invisible to the transaction level: an object can be copied or scanned even while a transaction holds a write lock.

The read barrier is the synchronization mechanism between garbage collection actions and the other elementary actions. A flip occurs while no elementary action is active. At the flip the collector protects the unscanned pages of to-space, thereby preventing update and read actions from executing. As each page of to-space is scanned, the collector changes the page's protection so that update and read actions that access objects on it can execute.

8.1.2 Synchronization With Other Logging

Because the collector writes to the log, we must ensure that a garbage collection trap cannot occur while the recovery system is writing a record to the log. If it did, the log would not be readable after a crash. To avoid this problem the recovery system constructs its records in memory outside of the heap; then it copies the record to the log buffer, which is also outside of the heap, in a critical section, which is uninterruptable by the garbage collector.

8.2 Roots in Recovery Information

The addresses in an update record (including the addresses in its undo information) are addresses in the to-space that is current at the time of the update. However, a transaction that updates an object and logs an update record during one garbage collection cycle when one space is current may be aborted during a subsequent cycle when another space is current. For the abort to succeed, the addresses in the undo information in the update record must be translatable to current to-space addresses and the objects reachable from those addresses must still be in the heap.

We call the addresses in the undo information in the update records of the active transactions *undo roots*. Potentially any active transaction could be aborted; thus, the undo roots must be treated by the garbage collector as roots for collection and be translatable to to-space addresses. Translation information for the undo roots is in copy records in the log, but searching the log for the copy records is expensive. Reading the log during garbage collection to find the undo roots is also expensive. Below we show how to avoid most accesses to the log for undo roots and translation.

In contrast to undo information, the collector can ignore the redo information in update records: the recovery system updates objects directly; thus, the latest redo information for an object is already reflected in the state of the object in the heap.

8.2.1 Avoiding Access to the Log to Find Undo Roots

By following a simple heuristic, the garbage collector avoids most if not all accesses to the log to find undo roots. Only the transactions that were active at the last flip may have written update records to the log before the flip. Furthermore, most transactions are shorter than the time between flips. Thus, if the collector defers its check for undo roots until all the objects reachable from the other roots have been copied to to-space and scanned, most if not all of the transactions that were active at the time of the flip will be complete and little or no work will remain.

8.2.2 Avoiding Access to the Log to Translate Undo Roots

During a garbage collection the current from-space and to-space are *active* spaces. We use forwarding pointers to translate undo roots for an active from-space to to-space addresses. A from-space becomes *inactive* when it is discarded at the completion of a garbage collection. The collector builds a data structure called the Undo Translation Table (UTT) to speed translation for the undo roots in update records that were written for inactive spaces. There is an entry in the UTT for each such undo root. The entry maps the undo root to its corresponding address in an active space.

In particular, the UTT maps from a <flip number, undo root> to a <current address, list of transaction identifiers> pair. The flip number is the number of the flip that started the log interval in which the update record was written. Because virtual addresses may repeat every second garbage collection interval but refer to different objects, we pair the flip number together with the address in order to uniquely identify the object referenced by the undo information. The list of transaction identifiers associated with an entry allows the entry to be deleted when no active transaction needs it.

The UTT is kept in volatile memory, so after a crash it is lost. In order to allow fast translation of undo roots after a crash, the information in the UTT is written to the log in an Undo Translation Record (UTR). The UTR contains the address translation map of the UTT, except that its entries do not contain the transaction identifiers. The transaction identifiers are not required because the recovery system aborts all active transactions when it recovers from a crash and it discards the information in the UTR as soon as recovery is complete.

When a stable heap is first created, the UTT is initialized to the empty state. At every garbage collection, the following procedure for updating the UTT and tracing the objects reachable from the undo roots is executed after all of the objects reachable from the other roots have been copied to to-space and scanned.

1. Delete entries for completed transactions from the UTT.
2. Use the addresses in remaining entries as roots for collection, copying each referenced object to to-space if it has not yet been copied and updating the address in the entry to the to-space address of the copy. (This step covers the undo roots in update records written before the penultimate flip.)
3. For each active transaction that was also active at the last flip, trace backwards through its chain of update records and find the records that were written between the penultimate and last flips. For each undo root in these update records:
   (a) Copy the object it references to to-space.
   (b) If no entry for <flip no., undo root> exists in the UTT, create an entry mapping it to its to-space address.
   (c) Add the transaction to the list of transactions associated with the entry for this undo root in the UTT.
4. Finish scanning to-space.
5. Construct a UTR from the UTT and write it to the log.

Since transaction abort and garbage collection depend on the UTT for the translation of addresses in undo roots, from-space cannot be discarded until the UTR is in the stable log.

8.3 Recoverable Allocation of Spaces

At a flip, the garbage collector allocates a new to-space. The new space requires a chunk of virtual addresses and a disk backing store. When a garbage collection completes, the collector deallocates from-space. These allocations and deallocations need to be recoverable so that the recovery system can reinitialize the virtual memory maps after a crash and determine which updates to from-space should be redone.

For the allocation, the garbage collector writes a flip record, shown in Figure 1. A flip record contains the virtual address range and the location of the disk backing store for both from-space and to-space. The information for from-space is also available in the last flip record, but we repeat it for easier access.

For the deallocation, we attach additional meaning to the UTR and wait to write it to the log until from-space can be deallocated. From-space can be deallocated when (1) the garbage collection is complete, (2) every scanned page of to-space has been written to disk since it was scanned, and (3) the UTR is in the stable log. To use the UTR as a sign that from-space can be deallocated, we delay writing the UTR until the first two conditions hold. From-space is not actually discarded until the UTR is in the stable log.

8.4 Recovery Independent of Heap Size

Frequent checkpoints and regular flushing of dirty pages to disk by the buffer manager keep the time for recovery short and independent of heap size. Recovery records information in a checkpoint record so that it can avoid processing the prefix of the log written before the corresponding checkpoint. In this section we describe the contents of the checkpoint record. In the first author's dissertation[22] we describe in detail how the checkpoint record, the other records in the log, and the disk are used after a system failure to recover the stable heap.

Figure 1 shows the contents of a checkpoint record. The first two items in the record, the buffer page list and the transaction list, also appear in Mohan's algorithm [29]. The last six items, the LSN of the last UTR, the LSN of the last flip, the address of the Last Object Table (LOT), the Scanned Set (SS), the allocation pointer, and the copy pointer are related to atomic garbage collection.

The buffer page list summarizes the information in the page-fetch and end-write records in the prefix of the log before the checkpoint. It is a list of <page number, LSN> pairs, one entry for each dirty page at the time of the checkpoint. The LSN is the LSN when it first became dirty.

The transaction list summarizes information about active transactions in the prefix of the log before the checkpoint. It is a list of <transaction identifier, LSN> pairs, one for each active transaction. The LSN points to the last update or compensation log record written for the transaction.

After a crash, recovery uses the translation information in the last UTR written to the stable log in order to translate addresses in the undo information for the transactions it aborts. It also uses the mapping information in the last flip record to re-initialize the virtual memory maps. The LSN of the UTR and the LSN of the flip record in the checkpoint record allow the last UTR and flip records written to the log to be found quickly. Alternatively, the contents of these records could be included directly in the checkpoint record. The address of the Last Object Table (LOT) allows the garbage collector to find the LOT in the stable heap after a crash.

The garbage collector keeps track of pages it has scanned so that it can avoid reprotecting and rescanning pages after a crash. The Scanned Set (SS) is the set of to-space pages that have been scanned. It is represented as a bitmap, one bit for each page of to-space. A checkpoint records the bitmap in its record to summarize the information in scan records written since the last flip and until the checkpoint.

To allow recovery of the allocation and copy pointers after a crash, the values of these pointers at the time of the checkpoint are also recorded in the checkpoint record.

9 Conclusion

A major goal in the design of any computer system is to achieve the required functionality at the minimum cost. For a recovery system, this means paying a minimal run-time cost in order to ensure that the heap can be recovered quickly after a crash. The repeating history recovery algorithm and its accompanying redo protocol have been designed with this goal in mind.

Recovery systems keep the run-time cost of recovery low in three ways: writing to a sequential log avoids random synchronous writes to disk; synchronous writes to the log are eliminated except for transaction commit; and the amount of data written to the log is kept to a minimum. By basing the atomic garbage collector on the redo protocol, we avoided synchronous I/O to disk and the log. Then we optimized the protocol for the copy and scan steps to minimize writing to the log.

Our atomic garbage collector never delays itself or transactions because it is waiting for the completion of a synchronous write to disk or to the log. In comparison, Detlefs [11] focused on minimizing writing to the log; as a result his concurrent atomic garbage collector requires both random synchronous writes to the disk and synchronous writes to the log.

We have completed the implementation of a stable heap prototype. In the future we plan to measure the prototype. Unfortunately, the Argus implementation is itself a prototype and has few users. Thus, we hope to incorporate our atomic incremental collector into an object-oriented database, which will provide a better platform for testing our algorithm.

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 Persistent Programming. *The Computer Journal*, 26(4):360--365, 1983.

[3] H. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, 21(4):280--294, April 1978.

[4] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company, Reading, Ma., 1987.

[5] P. B. Bishop. Computer Systems with a Very Large Address Space and Garbage Collection. Technical Report MIT/LCS/TR-178, Laboratory for Computer Science, MIT, Cambridge, Ma., May 1977.

[6] H.-J. Boehm, A. J. Demers, and S. Shenker. Mostly Parallel Garbage Collection. In *Proceedings of the SIGPLAN Symposium on Programming Language Design and Implementation*, pages 157--164, June 1991.

[7] A. Brown and J. Rosenberg. Persistent Object Stores: An Implementation Technique. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice. Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 199--212. Morgan Kaufmann Publishers, San Mateo, Ca., 1990.

[8] 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.

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

[10] D. Detlefs, M. Herlihy, and J. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, 21(12), December 1988.

[11] D. L. Detlefs. Concurrent, Atomic Garbage Collection. Technical Report CMU-CS-90-177, Department of Computer Science, Carnegie Mellon University, Pittsburgh, Pa., October 1990.

[12] J. R. Ellis, K. Li, and A. W. Appel. Real-time Concurrent Collection on Stock Multiprocessors. Technical Report 25, Systems Research Center, Digital Equipment Corporation, Palo Alto, Ca., February 1988.

[13] D. Gawlick and D. Kinkade. Varieties of Concurrency Control in IMS/VS Fast Path. *Database Engineering*, 8(2):63--70, June 1985.

[14] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publishers, San Mateo, Ca., 1993.

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

[16] M. P. Herlihy and J. M. Wing. Avalon: Language Support for Reliable Distributed Systems. In *Proceedings of the Seventeenth International Symposium on Fault-Tolerant Computing*, July 1987.

[17] R. L. Hudson and J. E. B. Moss. Incremental Collection of Mature Objects. In Y. Bekkers and J. Cohen, editors, *Memory Management: International Workshop IWMM 92*, volume 637 of *Lecture Notes in Computer Science*, pages 388--403. Springer-Verlag, Berlin, 1992.

[18] S. E. Hudson and R. King. Cactis: A Self-Adaptive, Concurrent Implementation of an Object-Oriented Database Management System. *ACM Transactions on Database Systems*, 14(3):291--321, September 1989.

[19] E. Kolodner. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. In A. Dearle, G. M. Shaw, and S. B. Zdonik, editors, *Implementing Persistent Object Bases: Principles and Practice. Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 185--198. Morgan Kaufmann Publishers, San Mateo, Ca., 1990.

[20] E. Kolodner, B. Liskov, and W. Weihl. Atomic Garbage Collection: Managing a Stable Heap. In *Proceedings of the 1989 ACM SIGMOD International Conference on the Management of Data*, pages 15--25, June 1989.

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

[22] E. K. Kolodner. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. Technical Report MIT/LCS/TR-534, Laboratory for Computer Science, MIT, Cambridge, Ma., February 1992.

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

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

[25] B. G. Lindsay, P. G. Selinger, C. Galtieri, J. N. Gray, R. A. Lorie, T. G. Price, F. Putzolu, I. L. Traiger, and B. W. Wade. Notes on Distributed Databases. Technical Report RJ2571, IBM Research Laboratory, San Jose, Ca., July 1979.

[26] B. Liskov. Distributed Programming in Argus. *Communications of the ACM*, 31(3):300--312, March 1988.

[27] B.. Liskov, P. Johnson, and R. Scheifler. Implementation of Argus. In *Proceedings of the Eleventh Symposium on Operating Systems Principles*, November 1987.

[28] D. Maier, J. Stein, A. Otis, and A. Purdy. Development of an Object-Oriented DBMS. In *Proceedings of the Object-Oriented Programming Systems, Languages and Applications*, pages 472--482, November 1986.

[29] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Grained Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Transactions on Database Systems*, 17(1):94--162, 1992.

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

[31] B. Oki, B. Liskov, and R. Scheifler. Reliable Object Storage to Support Atomic Actions. In *Proceedings of the Tenth Symposium on Operating Systems Principles*, pages 147--159, December 1985.

[32] R. F. Rashid. Threads of a New System. *Unix Review*, 4(8):37--49, August 1986.

[33] M. Reinhold. Personal communication.

[34] S. M. Thatte. Persistent Memory: A Storage Architecture for Object-Oriented Database Systems. In U. Dayal and K. Dittrich, editors, *Proceedings of the International Workshop on Object-Oriented Databases*, Pacific Grove, CA, September 1986.

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

[36] W. Weihl and B. Liskov. Implementation of Resilient, Atomic Data Types. *ACM Transactions on Programming Languages and Systems*, 7(2):244--269, April 1985.

[37] G. Weikum. A Theoretical Foundation of Multi-Level Concurrency Control. In *Proceedings of the Fifth ACM SIGACT-SIGMOD Symposium on the Principles of Database Systems*, pages 31--42, Cambridge, Ma., March 1986.

[38] G. Weikum. Principles and Realization Strategies of Multilevel Transaction Management. *ACM Transactions on Database Systems*, 16(1), 1991.

[39] P. R. Wilson, M. S. Lam, and T. G. Moher. Effective "Static-graph" Reorganization to Improve Locality in Garbage-Collected Systems. In *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 177--191, June 1991.

[40] S. Zdonik and P. Wegner. Language and Methodology for Object-Oriented Database Environments. In *Proceedings of the 19th Annual Hawaiian Conference on Systems Science*, January 1986.