Concurrent Compacting Garbage Collection of a Persistent Heap

James O'Toole
Scott Nettles
David Gifford

Appears in SOSP'93

Abstract

We describe a replicating garbage collector for a persistent heap. The garbage collector cooperates with a transaction manager to provide safe and efficient transactional storage management. Clients read and write the heap in primary memory and can commit or abort their write operations. When write operations are committed they are preserved in stable storage and survive system failures. Clients can freely access the heap during garbage collection because the collector concurrently builds a compact replica of the heap. A log captures client write operations and is used to support both the transaction manager and the replicating garbage collector.

Our implementation is the first to provide concurrent and compacting garbage collection of a persistent heap. Measurements show that concurrent replicating collection produces significantly shorter pause times than stop-and-copy collection. For small transactions, throughput is limited by the logging bandwidth of the underlying log manager. The results suggest that replicating garbage collection offers a flexible and efficient way to provide automatic storage management in transaction systems, object-oriented databases and persistent programming environments.

Authors' addresses: otoole@lcs.mit.edu, nettles@cs.cmu.edu, gifford@lcs.mit.edu. Laboratory of Computer Science, Massachusetts Insitute of Technology, 545 Technology Square, Cambridge, MA 02139. School of Computer Science, Carnegie Mellon University, 5000 Forbes Avenue, Pittsburgh, PA 15213.

This research was sponsored by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, Arpa Order No. 7597, by the Air Force Systems Command, by the Advanced Research Projects Agency under Contract F19628-91-C-0128, and by the Department of the Army under Contract DABT63-92-C-0012.

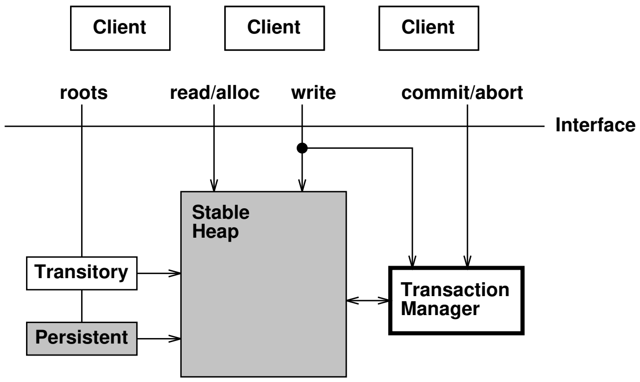

Figure 1: The Transactional Heap Interface

1 Introduction

Operating systems, persistent programming environments, and object repositories must store dynamically allocated persistent data structures. Unfortunately, using explicit deallocation to manage these data structures can easily cause catastrophic system failures due to the effects of dangling pointers and storage leaks. To address these problems, we have designed a storage management system that uses concurrent replicating garbage collection to provide safe and efficient automatic storage management for persistent data.

Traditionally, operating systems have used explicit deallocation and reference counting to manage persistent storage. Explicit deallocation can result in dangling pointers and storage leaks. Reference counting can be expensive and results in storage leaks because it does not recover cyclic structures. We believe that these risks are unacceptable when storing data that is so valuable that it must survive system failures.

Permanent data storage management should meet the much higher safety standard achieved by tracing garbage collection. Modern garbage collectors are very efficient and can often be competitive with explicit deallocation [23]. However, existing garbage collectors are not practical for use in systems applications. Existing implementations either stop the client while garbage collecting or do not maintain a heap image that is resilient to system failure.

We have designed and implemented the first concurrent compacting garbage collector for a persistent heap. Clients are permitted to read and write the heap while the collector concurrently replicates objects to create a new stable heap. Clients are free to modify objects that have already been copied because the modifications are recorded in a log. The log is used by the collector to ensure that the new stable heap contains all pertinent data before it is used to replace the old stable heap. The log is processed concurrently, reducing garbage collection interruptions imposed on the clients to brief synchronization pauses.

There are several important advantages that derive from using concurrent replicating collection in a transactional system:

- The garbage collector is simple and easy to implement.
- Transaction processing and garbage collection operations are decoupled so that collector activity need not affect transaction throughput or commit latencies.
- A single log is shared by the transaction manager and the garbage collector.

In practice, our implementation provides clients more responsive access to the heap than does a stop-and-copy collector. Concurrent replicating collection eliminates lengthy interruptions caused by garbage collection. The implementation supports transactions, recovers from system failures, and provides good performance. Transaction throughput for simple small transactions can easily exceed 70 transactions per second. In this case the transaction manager is limited by the bandwidth of the underlying stable storage log manager. We believe that our design offers a practical approach to storage management in persistent programming environments, object repositories, and similar applications.

The focus of this paper is on how a garbage-collected persistent heap can be implemented with complete safety and good performance. The remaining sections describe our system interface (Section 2), present the design of the storage system in detail (Section 3), present an implementation of our design (Section 4), summarize our experimental results with the implementation (Section 5), discuss related work (Section 6), and discuss elaborations and applications of the basic algorithm (Section 7).

2 Interface

Our interface provides basic heap operations, transaction operations, and two distinguished roots. The complete interface is shown in Figure 1. The basic heap operations are read, write and allocate. The transaction operations are commit and abort.

The transitory and persistent roots are distinguished and available to the clients. No deallocation operation is exposed to the clients. Instead, objects that the clients can access by dereferencing pointers starting from either root are considered live and will be preserved by the system. A garbage collector will identify unreachable objects and recycle their storage.

Our interface provides *orthogonal persistence* [4]: objects that are reachable via the persistent root are guaranteed to survive system failures. In Figure 1, the stability of the persistent root and the stable heap to which it points are indicated by their gray background. In contrast, objects reachable only via the transitory root are not available after a failure; the transitory root is reset when the system recovers from a failure. Clients can use the transitory root to maintain access to temporary objects without requiring that the system ensure the persistence of those objects.

Our interface also includes the ability for multiple clients to perform transactions on the heap by using the commit and abort operations. The transaction manager is responsible for tracking modifications to objects and implementing the transaction semantics implied by commit and abort. Support for multiple clients, multithreaded clients, and nested transactions can be provided by appropriate choice of the transaction manager.

The exact transaction semantics supported by the transaction manager are not of primary concern in this paper. In considering the design of our system, we will assume that the complexities of multiple clients and multiple transactions are completely hidden from the rest of the system by the transaction manager. We describe the system as if it contains only a single client that performs a linear sequence of top-level transactions. Every modification to an object is part of a transaction and each transaction can be either committed or aborted. The heap must be restored to the most recently committed state if a system failure occurs.

3 Design

Our design shows how the transactional heap interface can be implemented efficiently. The interface specifies a high standard of programming safety and data stability. We provide programming safety by using garbage collection and orthogonal persistence. Data stability is ensured through the use of transactions and stable storage. Good performance comes from caching stable data in volatile memory and from the use of concurrent replicating collection. Using an extra processor to perform concurrent replicating collection improves throughput and provides the client with low-latency access to the heap.

We will present the design of the storage manager as a series of three refinements to a basic design. Each refinement improves either the functionality or the performance of the basic design:

- *Basic Design: Replicating Collection of Stable Heaps.* Our basic design uses a replicating collector on stable spaces. Clients operate on *stable from-space*, and the collector concurrently copies from-space into *stable to-space*. In this design each write is individually

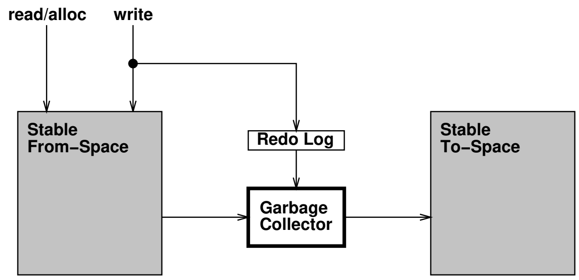

Figure 2: Replicating Collection of a Stable Heap

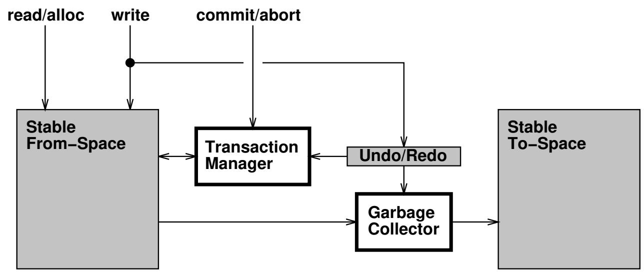

Figure 3: A Transactional Heap

committed and the client must access stable storage for every read and write operation.

- *Refinement 1: Transactions Group Updates.* Our first refinement adds transactions to the basic design. Transactions allow the client to perform a group of modifications atomically. The log required to support transactions also serves as the log used by replicating garbage collection.
- *Refinement 2: Volatile Images Improve Performance.* Our second refinement adds volatile main-memory images of from-space and to-space to improve the performance of the client and the collector. Committed from-space operations must be recovered upon failure. Thus it is necessary for the transaction manager to ensure that all committed operations are recorded in stable from-space.
- *Refinement 3: Transitory Heaps for Temporary Data.* Our third and final refinement adds a *transitory heap*. All objects are initially allocated in the transitory heap, and are automatically promoted to the persistent heap when they are made reachable from the persistent root. The transitory heap is garbage collected with respect to the *transitory root* and is not recovered after a failure.

3.1 Replicating Collection of Stable Heaps

As shown in Figure 2, using stable storage and replicating collection for the persistent heap provides an interface that lacks only transaction support. In this design each write operation is individually committed and becomes permanent as soon as it is performed. Good performance can be achieved by using battery-protected random access memory to implement high speed stable storage. While the client operates on the heap, a replicating garbage collector concurrently builds a compact replica of it.

Replicating garbage collection [16] is a new technique for building incremental and concurrent garbage collectors. The key idea of replicating garbage collection is to copy objects non-destructively. As in any copying garbage collection algorithm, the collector copies the reachable objects from the from-space to the to-space. However, most copying collectors destroy the original object when it is copied because the contents of the object are overwritten by relocation information containing its new address.

In contrast, a replicating collector avoids destroying the contents of the original object. This can be accomplished by storing the relocation information about the object in a reserved portion of the object or elsewhere. The effects of replication are unobservable by the client, so it continues to operate on the original objects in from-space. This differs from other concurrent compacting collection algorithms, which require the client to operate on objects in to-space [5, 3].

Because the client and the collector execute concurrently, the replicas become inconsistent when the client modifies the original objects. To solve this problem, the client records all write operations in a redo log, shown in Figure 2. The collector processes the log to ensure that all replicas are consistent. When to-space contains an exact replica of the entire object graph, the collector *flips*, updating the client's roots to refer to the replicas and exchanging the roles of to-space and from-space.

After a failure the client can be immediately restarted on stable from-space without any recovery processing. The stable nature of from-space guarantees it will survive failures. After a crash all that is necessary is to locate the stable from-space in memory. Because from-space and to-space exchange roles when the garbage collector flips, the garbage collector must store a from-space identifier to indicate which of the two stable spaces is the current from-space. The flip occurs when this identifier is updated atomically.

The garbage collector can be resumed after a failure if sufficient state is recorded in stable storage. If the collector is to be recoverable, the redo log and the relocation information containing the new address of each replicated object must be in stable storage. These items will collectively be called the *garbage collector state*. Alternatively the garbage collector state can be kept in volatile storage and the garbage collector can be restarted with an empty to-space upon failure. If failures are infrequent this option will result in better performance.

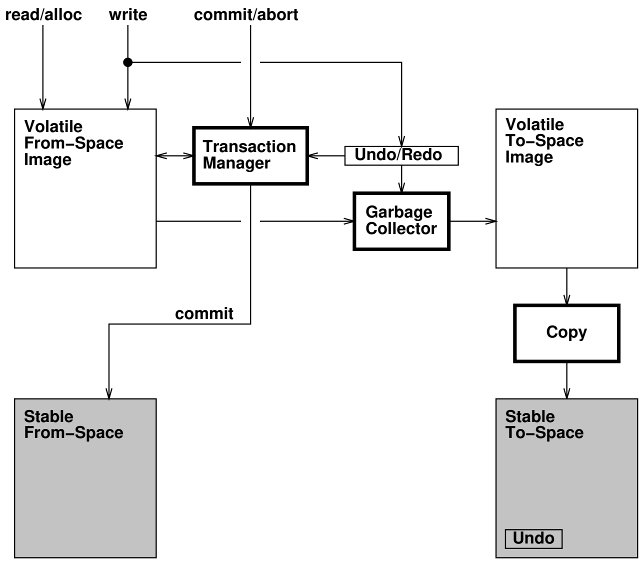

Figure 4: Using Volatile Images

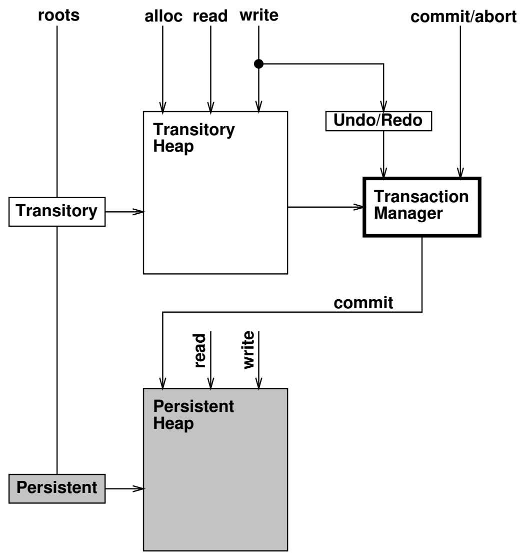

Figure 5: Using a Transitory Heap

3.2 Transactions Group Updates

Figure 3 shows how transactions can be added to the basic design. In this model, each modification is part of a transaction and transactions can be independently committed or aborted. All modifications are applied directly to from-space and an *undo log* describing uncommitted modifications is maintained. If a transaction aborts, all of its modifications are undone by using the undo log. If a transaction commits, then its modifications are atomically removed from the undo log. An important advantage of using replicating collection in a transactional setting is that the same undo log necessary to support transactions can also be used by the collector as a redo log. This minimizes the cost of using replicating collection.

After a failure the client can be resumed on stable from-space once the undo log is applied. Undoing the modifications in the undo log ensures that all uncommitted updates are erased before the client resumes execution. The undo log must be stored in stable storage so that it will survive failures and be available for recovery processing after a failure. After recovery the garbage collector can be resumed if the garbage collector state has been preserved in stable storage.

3.3 Volatile Images Improve Performance

When slow stable storage media are used, forcing all operations to access stable storage will not provide acceptable performance. Figure 4 shows how this problem can be addressed by caching images of the stable heaps in volatile memory. In this design the client reads, writes, and allocates in a volatile image of from-space. The garbage collector reads the volatile from-space image and writes the volatile to-space image.

When a transaction commits, its updates to volatile from-space must be made stable. Upon commit the transaction manager uses the redo log to identify portions of the volatile from-space image that have been updated and writes these modifications to stable from-space. It also writes any newly allocated objects to stable from-space. Applying these updates to stable from-space must be performed atomically in order to guarantee a consistent version of stable from-space in the presence of failures.

The garbage collector directly writes the volatile to-space image and a background process copies the volatile to-space image onto the slow stable storage media. Thus a slow stable storage medium can be used for to-space without slowing the garbage collector. A flip may occur when all live objects in volatile from-space have been copied, the redo log is empty, and volatile to-space has been completely copied to stable to-space. The flip causes both stable and volatile from-space to be replaced by their to-space counterparts.

However, the stable to-space may contain uncommitted data because it is an exact copy of volatile to-space, which is a replica of volatile from-space. To prevent this uncommitted data from being used after a crash, the undo log is also written to stable storage as part of the stable to-space. This allows any uncommitted transaction to be rolled back during recovery.

After a failure the client can use the volatile from-space image once it is recovered from stable from-space and the undo log is applied. After recovery the garbage collector can be resumed if its state has been preserved in stable storage. Otherwise it is restarted with an empty volatile to-space.

3.4 Transitory Heaps for Temporary Data

Figure 5 shows how a transitory heap may be added to our previous design. The persistent heap, shown in gray, represents the volatile and stable versions of from-space and to-space from the previous design. This figure emphasizes that the role of the transitory heap is to store objects that are not reachable from the persistent root. The transitory heap has its own root and is stored in volatile memory. All objects are initially allocated in the transitory heap.

Upon commit the transaction manager uses the log to detect objects in the transitory heap that have become persistent. The log is used to locate modified objects in the persistent heap that contain pointers to objects in the transitory heap. These objects are newly persistent because they have become reachable via the persistent root.

The modified objects are used as the roots of a stop-and-copy collection that promotes the newly persistent objects into the persistent heap. The promoted objects must be written to stable storage along with any persistent objects modified by the transaction.

It is also possible for persistent objects to become transitory. This happens when objects in the persistent heap are reachable from the transitory root but have become unreachable from the persistent root. Such objects are live but not persistent. When these objects are discovered they may be moved back into the transitory heap or left in the persistent heap. We have chosen to leave these objects in the persistent heap because it simplifies our implementation. After a failure these objects are unreachable and will be reclaimed by the next collection.

After a failure the client is restarted on the persistent heap using the algorithm outlined in the previous section. During recovery the transitory heap is initialized to be empty.

4 Implementation

Our prototype implementation provides concurrent compacting garbage collection for Standard ML of New Jersey. We added a persistent heap and a transaction manager to the runtime system to support persistence. We re-implemented the garbage collector as a concurrent thread using replicating collection. The purpose of the prototype implementation was to test the feasibility of our design.

Standard ML of New Jersey (SML/NJ) is an implementation of a type-safe programming language that includes an optimizing compiler, a runtime system, and a generational garbage collector [2]. The SML/NJ source code is freely available and is easy to modify for experimental purposes.

We chose to test our garbage collection algorithm in the SML/NJ environment primarily for reasons of convenience. Our previous work on persistence [18] and replicating garbage collection [16, 17, 19] using ML provided us with several of the components needed for our prototype.

The SML/NJ runtime system is similar to most language implementations with copying garbage collection. We therefore believe that our results apply to languages such as Modula-3, Lisp, Scheme, Smalltalk and, if suitably modified to enable copying garbage collection, C++.

Overview

An implementation of the basic design and its three refinements requires a replicating collector, an undo/redo log, stable storage, and a transaction manager. Here is a summary of how each of these components is implemented:

- The replicating garbage collector is implemented as in our previous work, except that it runs as a separate thread of control so that it provides concurrent collection. It occasionally interrupts the client to obtain more up-to-date roots and log entries. It also pauses the client in order to perform a flip.

- The undo/redo log is written by the client. We modified the SML/NJ compiler to emit instructions in the client code that generate appropriate log entries for every write operation. The original SML/NJ implementation included a simpler log to support generational garbage collection. We expanded the log to include undo information about every write operation in order to support transaction operations and replicating collection.

- The stable heaps are maintained on disk and their volatile images are maintained in main memory. We use Recoverable Virtual Memory to manage the stable heaps on disk. RVM allows us to apply changes atomically to the stable heap by writing the changes into a stable log. After a failure, the RVM log is used to recover the stable heap.

- The transaction manager in our current implementation performs transaction commit by reading the log, moving objects from the transitory heap to the volatile heap image, and logging the resulting changes to the stable heap via RVM. The transitory heap is merely the simple generational heap present in the original SML/NJ implementation.

In the sections that follow we discuss some choices we faced when implementing our prototype. We then describe the structure of the persistent heap and review the threads of control used by our implementation. We then present the step by step procedures used to implement the commit, collection, flip and recovery operations.

4.1 The Transitory Heap

The transitory heap contains only temporary objects that will be discarded when a failure occurs. It consists of the two generational heaps present in the original SML/NJ implementation. The details of the these transitory heaps are mostly irrelevant to our persistent heap implementation.

However, when constructing our prototype we faced several implementation choices that involved the transitory heap. There are several situations in which pointers that cross between the transitory and the persistent heap may have to be adjusted when objects are moved.

For example, when newly persistent objects are promoted into the persistent heap, any other temporary objects that contain pointers to them must be updated. Similar adjustments may be required to either the transitory or the persistent heap when the other heap is being flipped by the garbage collector.

In these situations, the implementation could use any one of at least four different solutions:

1. Scan the heap that contains the pointers requiring adjustment.
2. Garbage collect the heap that contains the pointers requiring adjustment.
3. Somehow track the locations of the pointers that require adjustment so that they can be updated more quickly.
4. Use replication to move the objects, but refrain from updating any pointers to the objects until some later opportunity such as a flip.

In our implementation, pointers from the persistent heap to the transitory heap are always the result of uncommitted write operations. They can be easily located via the transaction log. Therefore, we use the third solution to update these pointers when the transitory heap is garbage collected. To deal with pointers in the other direction, we implemented both of the first two methods: doing a scan and doing a garbage collection.

4.2 Stable Storage

Our design assumes the ability to make multiple updates to the stable heap atomically. To achieve this our implementation uses the Recoverable Virtual Memory system described by Satyanarayanan, et al. [20] RVM provides simple non-nested transactions on byte arrays and uses a disk-based log for efficiency. We do not exploit the rollback features of RVM at all; in our implementation, RVM provides only disk logging and recovery services.

RVM allows us to establish a one-to-one correspondence between a file and a portion of virtual memory. We use separate files to store stable from-space and stable to-space. These files also contain starting and ending addresses of the spaces and the sequence number that serves as the stable from-space identifier.

RVM defines a simple interface that allows changes to the volatile heap to be atomically transferred to the stable heap. During commit we begin an RVM transaction, inform RVM of the location of each modification to the volatile heap as well as of any newly persistent data, and then end the transaction. RVM guarantees that this set of operations will be atomic.

When a flip occurs the from-space identifier must be updated to indicate which of the stable spaces is the stable from-space. This update does not need to be made stable until the next client transaction commit after the flip. When the client first commits a transaction, that transaction will be applied to the new stable from-space. We write the from-space identifier via RVM, but using a "no-flush" transaction that does not require a synchronous disk write. Thus, although the client is paused while the garbage collector finalizes the flip and updates the roots, no synchronous disk write occurs during this interruption.

4.3 Writing Stable To-space

Volatile to-space is written to stable to-space by an asynchronous process. No writes that synchronize with the client are required. Volatile to-space can be written directly to the file containing stable to-space or indirectly via RVM.

During the garbage collection, large contiguous regions of new data are created in volatile to-space. These regions can be written directly to disk efficiently. Logging them through RVM is less efficient because RVM first writes them into its log and then later writes them again to the data file.

However, the garbage collector also performs small random updates on volatile to-space when it uses the redo log to update inconsistent to-space replicas. RVM is ideal for applying these changes to stable to-space, because its use of logging converts the small random writes into a single efficient log write. Writing these changes to disk without using a log is of course more expensive.

We tried writing stable to-space using several different methods. We found that using RVM for all of the writes put so much data into the RVM log that the client transaction flow was substantially impeded. However, when we only wrote stable to-space directly we found it difficult to schedule the flip to avoid blocking while waiting for the last few disk writes to complete.

Now our implementation uses a mixed strategy in which the large contiguous writes are done directly to the file containing stable to-space and the small random writes are done using RVM.

4.4 The Persistent Heap

The persistent heap is maintained both in virtual memory and on stable storage through the cooperation of the transaction manager, the garbage collector and RVM. Figure 6 shows the primary data structures used by the runtime system to maintain and garbage collect the persistent heap:

- **undo/redo log** — This log is maintained by the client and is used by both the transaction manager and the collector. When the stable heap flips, the log is written to stable storage to enable rollback of uncommitted transactions.
- **volatile from-space** — The client accesses all persistent data through volatile from-space. When a transaction commits, all newly persistent objects are copied from the transitory heap into volatile from-space.

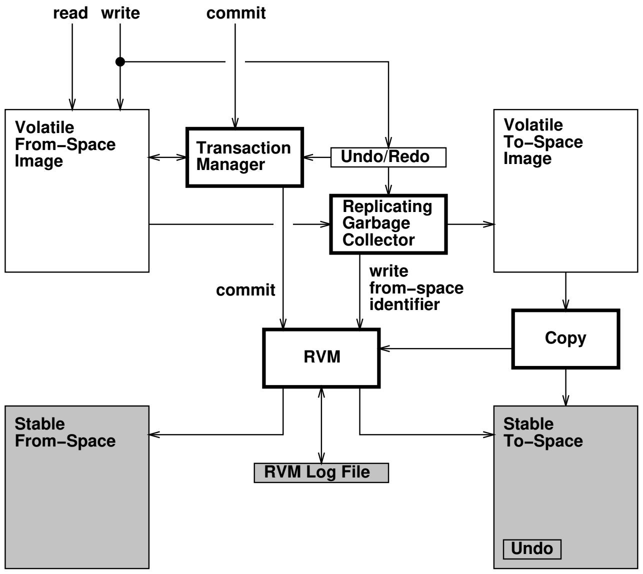

Figure 6: The Persistent Heap

- **volatile to-space** — The garbage collector copies the live objects in volatile from-space to volatile to-space.
- **stable from-space** — The stable representation of from-space is stored on disk. Upon commit the newly persistent objects and the new values of any modified locations in volatile from-space are written to stable from-space by the transaction manager using RVM.
- **stable to-space** — The stable representation of to-space is stored on disk. It is a copy of volatile to-space and must be written to disk before a flip.
- **the RVM log** — The stable log is maintained by RVM on disk. The RVM log contains records that describe modifications to stable from-space and stable to-space. RVM updates this log atomically and uses it to recover the contents of the stable spaces after a failure.

4.5 Threads of Control

The prototype contains three threads of control:

- The **Client** thread executes the application program and periodically commits transactions by acting as the Transaction Manager. The Client thread is responsible for writing the undo/redo log. The Client thread also traps into the runtime system when it must synchronize with the garbage collector and perform a flip.
- The **Collector** thread performs replicating garbage collection, copying reachable objects from the volatile from-space into the volatile to-space. When the Collector thread has constructed a complete replica of the volatile from-space, it signals the Client thread that a flip should take place.
- The **Copy** thread writes the contents of volatile to-space onto the stable to-space, either directly to the disk or via the RVM log manager. It performs I/O operations on behalf of the garbage collector so that the Collector thread need not block on I/O.

4.6 Operations on the Persistent Heap

Now we consider in detail the steps needed to perform the key operations of our design.

Commit

When the Client thread commits a transaction, it acts as the Transaction Manager and performs the following operations:

1. Scan the undo/redo log to locate references that created pointers from the volatile from-space to transitory data. These references are used to identify the roots of the newly persistent data.
2. Promote all newly persistent objects into volatile from-space, using a standard stop-and-copy garbage collection algorithm. This collection uses the locations identified in step 1 as roots.
3. Atomically update the stable from-space by logging and committing an RVM transaction containing all modifications and additions to volatile from-space.
4. Update all of the transitory heap data to point to the newly promoted objects in volatile from-space. This is done either by scanning the transitory heap for pointers to promoted objects or by garbage collecting the transitory heap.

Persistent GC

When a commit adds enough new data to the volatile from-space to exceed a predetermined threshold, a garbage collection of the persistent heap is initiated. At this point the Collector thread performs the following steps asynchronously with respect to the Client thread:

1. Copy all objects in volatile from-space that are reachable from the persistent root into volatile to-space. This preserves the persistent data.
2. Copy all objects in volatile from-space that are reachable from the transitory heap into volatile to-space. This preserves the objects that are live and have become transitory.
3. Scan the undo/redo log, and update replicas of modified objects in the volatile to-space.

The undo/read log is reset to empty after its entries have been processed by both the transaction manager and the garbage collector. The transaction manager and the garbage collector maintain separate pointers into the shared log to keep track of what prefix of the log they have already processed.

When a collection is in progress the Copy thread asynchronously copies the volatile to-space image into stable to-space. The Copy thread occasionally synchronizes with the Collector thread to obtain information about what portions of volatile to-space have been modified.

Persistent GC Flip

A flip is attempted after the Collector thread has successfully replicated volatile from-space in volatile to-space and the Copy thread has copied it into stable to-space. Then the Collector thread stops the Client thread and performs any remaining collection work required by recent client operations. Finally, the Collector thread performs the following steps before resuming the client:

1. Update the client's roots to point to volatile to-space.
2. Update all pointers in the transitory heap that point to volatile from-space to point to volatile to-space.
3. Write the stable from-space identifier via RVM to indicate that stable to-space is now stable from-space.

The stable replicating collector must ensure that the undo/redo log is preserved in stable storage at the time of a flip. In the prototype, this is accomplished very simply because the transaction manager stores the log in the volatile from-space. Therefore, it is preserved in volatile to-space by the Collector thread and written to stable storage by the Copy thread.

The flip has occurred in volatile memory when these steps are complete. The Collector thread does not write the stable from-space identifier synchronously. Instead, it uses a no-flush transaction that will update the from-space identifier before any subsequent client transactions.

Recovery

When a crash occurs and the system restarts the following steps are performed for recovery:

1. RVM applies pending committed log records from its stable log to bring the stable from-space and stable to-space files to their most recently committed state.
2. The garbage collector examines the stable from-space identifier stored in the stable heaps to determine which heap is the stable from-space.
3. The transaction manager uses the undo log stored in stable from-space to roll back the uncommitted operations of any partially complete transactions. These changes are made atomically using RVM.

The Client thread can now be restarted using the recovered volatile from-space and an empty transitory heap. The collector begins with an empty volatile to-space.

5 Performance

We designed and ran a series of experiments to test whether our algorithm:

- significantly reduces the duration of collector pauses.
- increases transaction throughput by reducing storage management overhead.

The experiments compare the implementation described in the previous section to our implementation of a stop-and-copy collector for the persistent heap. We are unable to compare our work directly to other concurrent collector implementations because they do not support the collection of stable heaps.

Our experiments demonstrate that replicating collection interferes with the client less than stop-and-copy collection. For our collector, the longest pauses are a few hundred milliseconds, the same general magnitude as commits. For heap sizes in the megabyte range, the pause times achieved by our technique are a factor of ten shorter than stop-and-copy collection. For larger heaps, such as those found in a production object database, the difference would be even greater. We are also able to demonstrate substantial improvements in throughput due to our technique.

We also measured the commit performance of the system. Commit performance depends mostly on the choice of persistence model, the transaction profile, and the performance of the underlying log manager. We measured this aspect of the system to better understand our prototype implementation. We found that the commit performance can be quite good.

5.1 Benchmarks

We used three benchmarks to test our implementation. Each was chosen to measure and stress different aspects of the system. Two of the benchmarks performed a significant number of garbage collections, while the third was used to measure transaction throughput. Although we did not measure recovery performance we did crash and recover each of our benchmarks to verify that they were recoverable.

- The *Compiler* benchmark is Standard ML of New Jersey compiling a portion of the SML/NJ implementation. We modified the compiler to store all of its data in the persistent heap and to commit its state every time a module (file) was compiled, modeling the behavior of a persistent programming environment. This 100,000 line program is compute-intensive and contains long-running transactions.

- The *TP-OO1-V* benchmark is a variant of the OO1 Engineering Database benchmark described by Cattell [6]. The benchmark models an engineering application using a database of parts, performing traversals of the database, adding parts, etc. We implemented this algorithm in order to have a representative object-oriented database application. However, the OO1 benchmark, as specified, does not require garbage collection, so we added deletion operations to make it a more realistic application for our system.
- The *TPC-B* benchmark performs a large number of bank teller operations that perform transfers among various bank accounts. This benchmark is our slightly non-standard implementation in Standard ML of the TPC-B benchmark from Gray [8].

5.2 Experimental Setup

All benchmarks were executed on a Silicon Graphics 4D/340 equipped with 256 megabytes of physical memory. The clock resolution on this system is 1 millisecond. The machine contains four MIPS R3000 processors clocked at 33 megahertz. Each processor has a 64 kilobyte instruction cache, a 64 kilobyte primary data cache, and a 256 kilobyte secondary data cache. The secondary data caches are kept consistent via a shared memory bus-watching protocol and there is a five-word deep store buffer between the primary and the secondary caches. With this configuration, the collector can copy between 1 and 2 megabytes per second.

All benchmarks were executed using a *transactional process* model. Each time the application requests a commit, its entire process state is committed to the persistent heap, including processor register contents. We chose to run the benchmarks using this model because it generally minimizes the size of the transitory heaps and maximizes the workload on the stable heap. The transactional process model itself is implemented in SML using the persistence facilities described here.

5.3 Pause Times

The most important property of our collector is that the pause times it imposes on the client are short and bounded. Very large stable heaps imply very large stop-and-copy garbage collection pauses. We want to demonstrate that the stable replicating collector pause times are independent of heap size. We also want to examine the distribution of pause times caused by both the concurrent and stop-and-copy collectors and compare them to pauses caused by commit operations.

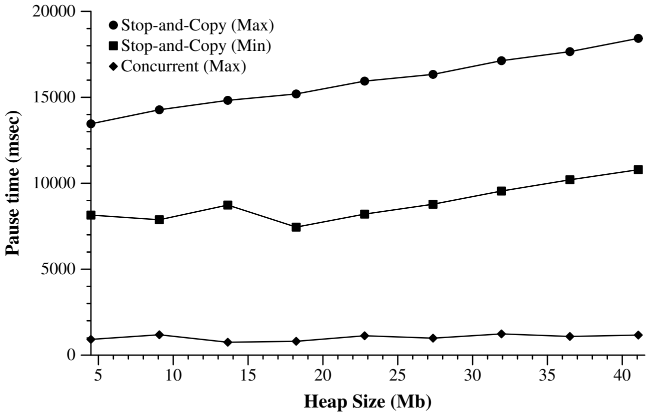

Figure 7: OO1 Collector Pause Times

5.3.1 Heap Size Dependence

We ran the OO1 Engineering Database benchmark in Standard ML with various amounts of live data in the heap. This allowed us to examine the dependence of collector pause duration on heap size. Figure 7 shows the maximum pause times caused by collection of the persistent heap plotted with respect to heap size. The plot also includes the minimum pause times for the stop-and-copy collector. The minimum pause times for the concurrent collector are too small to measure.

Even for the modest heap sizes used here, the pauses created by the stop-and-copy collector are unacceptably long. As expected, pause times increase with increasing heap size. In contrast, the maximum pause created by the concurrent collector is approximately 1 second and is brief enough not to cause a major disruption. Although not shown in this figure, typical concurrent collector pauses for this benchmark were less than 300 milliseconds. These results show that pause times are independent of heap size for the concurrent collector.

We are investigating the cause of the longer pauses and believe they can be avoided. Examining the worst pauses in an earlier implementation was what convinced us that stable to-space should be written using a combination of direct disk writes and RVM. That change produced a five fold reduction in the maximum pause times and eliminated almost all pauses in the 100 to 1000 millisecond range. We believe that further tuning will reduce our maximum pause times to 100 milliseconds or below.

5.3.2 Pause Time Distributions

To explore how the frequency and duration of collection pauses compared to commit pauses, we ran the *Compiler* benchmark using both the concurrent collector and the stop-and-copy collector. We measured pauses caused by persistent garbage collector activity and transaction commit operations. Figures 8, 9, and 10 show the pauses from concurrent collector activity, stop-and-copy collections, and transaction com-

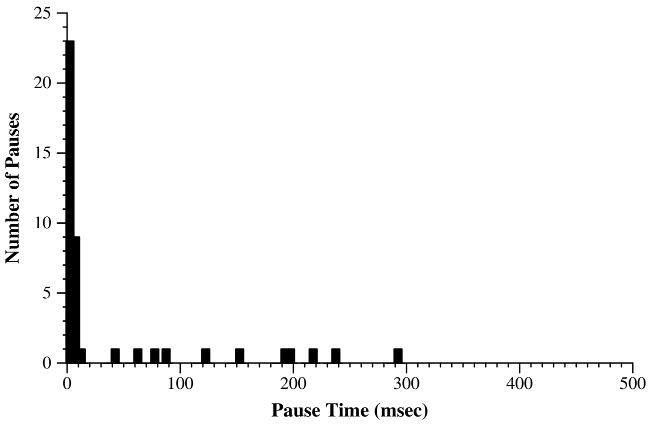

Figure 8: Concurrent Collector Pause Distribution

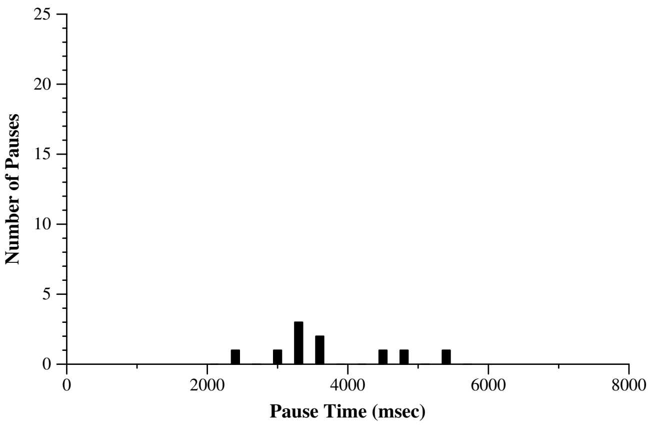

Figure 9: Stop-and-Copy Pause Distribution (Note scale)

mit, respectively. Note that the scale of Figure 8 is smaller than in the other figures because the pauses caused by the concurrent collector are very brief. The commit pauses shown here were produced using the concurrent collector, and are essentially the same as those produced when using the stop-and-copy collector. Stop-and-copy and commit pauses have been grouped into histogram bins 300 milliseconds wide.

As shown in Figure 8, the pauses due to the concurrent collector are mostly very short. The long tail is comprised entirely of pauses during which the collector completed a flip. Even the longest pause is short compared to most commit pauses. The total time spent by the client waiting for the collector was 1760 milliseconds. This is less time than the shortest stop-and-copy pause.

In contrast, the pauses shown in Figure 9 show that all of the stop-and-copy pauses are long enough to be disruptive. Many applications would be unable to use this collector due to the long interruptions. The total time spent in collection was 35 seconds. This is almost twenty times that for the concurrent collector.

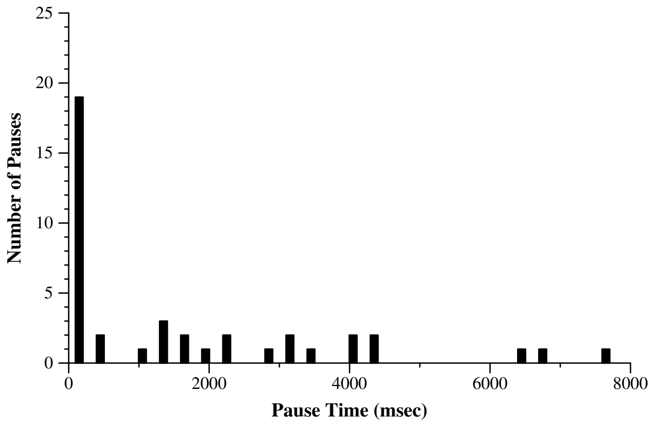

Figure 10: Commit Pause Distribution (Note scale)

5.4 Transaction Throughput

We were also interested in examining the throughput of our system. We wanted to know the answer to two questions. First, could the system provide good performance for simple small transactions? Second, can concurrent replicating collection provide increased throughput for systems that garbage collect? The answer to both of these questions is yes.

5.4.1 Commit Performance

In order to test the fast paths for transaction commit, we executed the TPC-B program using a 12 megabyte database containing bank account records for 1 branch covering 100,000 customers. During this test, our implementation ran 80 transactions per second. The limiting factor was the synchronous disk write required by each transaction. Note that this database size is smaller than the TPC-B guidelines require for a system claiming 80 transactions per second. We believe that we have complied with all other requirements of the benchmark standard. Although we have not done a careful comparison, a similar RVM benchmark [20] also executed approximately 80 transactions per second on our machine.

When the TPC-B benchmark is completing 80 transactions per second, each transaction requires 12.5 milliseconds of processing. We measured our system to determine how those 12.5 milliseconds were being used. Measurements of our system and independent measurements of write calls show that the synchronous write accounts for about 8 milliseconds or 65% of the total time. Other processing by RVM during the end transaction accounts for another 10%. An additional 10% came from a surprising source, an instruction cache flush required because the collector may copy machine code. Traversing the undo/redo log and logging the entries to RVM accounts for an additional 5%. No single factor accounts for the remaining 10% of the time. The time spent executing the benchmark code, which was too small to measure accurately, was certainly less than 0.5% of the total per-transaction time.

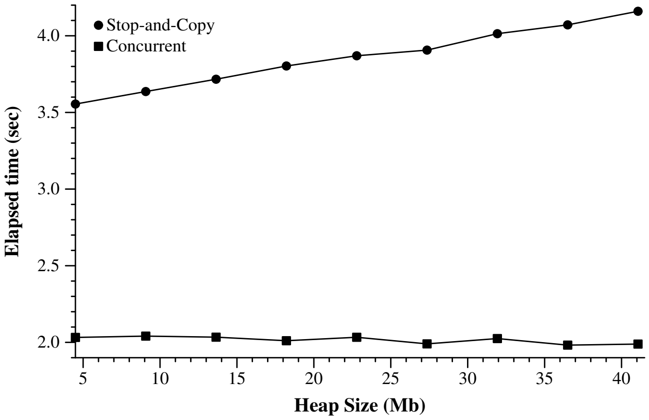

Figure 11: OO1 Elapsed Time per Transaction

5.4.2 Heap Size Dependence

We used our variant of OO1 to study the transaction throughput of our system in the presence of garbage collection. Our version of the benchmark performs twice as many updates to persistent data because each transaction includes one hundred deletions and one hundred insertions. This change to the benchmark causes it to require garbage collection, yet maintain a constant amount of live data in the heap. We measured the elapsed time to perform the standard engineering modification described by the benchmark.

We varied the total heap size by adding various amounts of live data. Figure 11 shows the elapsed time to perform each transaction as a function of heap size. The use of concurrent collection is clearly advantageous. The elapsed time for the concurrent collector is constant with heap size. As expected, the stop-and-copy collector causes the elapsed time to increase linearly with heap size.

5.5 Improving Stable Storage Access

In most transaction systems the critical performance issue is access to stable storage. In this section we consider a variety of performance issues and enhancements. Most of these issues are closely tied to our use of RVM for stable storage.

5.5.1 Transaction Reordering

We ran the compiler benchmark using configurations in which all writes of stable to-space either only used the disk directly or only used RVM. Using only the disk made it hard to synchronize with the client to achieve the flip. The key difficulty was scheduling the flip when no disk write was in progress. This lead to occasional very long pauses, as well as more pauses in the few hundred millisecond range. In the configuration that used only RVM, the extra log traffic from the garbage collector thread introduced substantial additional commit delays.

In the concurrent replicating algorithm, the write traffic from the garbage collector need not slow down the commit traffic from the client because the writes are to different spaces. The transactions that the collector performs on to-space can be reordered with the client transactions on from-space without affecting correctness. A suitable change to RVM would allow us to take better advantage of this fact by allowing some transactions to move through the log independently of others.

5.5.2 Change Record Batching

For large transactions, such as in the compiler benchmark, we examined the detailed components of both commit and collection closely. Our examination shows that a significant fraction of the time is spent processing the log, primarily the cost of logging the modifications through RVM. This suggests several possible enhancements that should improve commit performance or reduce total overhead or both.

Many of the modifications on the log are to the same location. Recognizing these duplicate log entries and eliminating them would reduce the logging cost substantially. A more efficient interface to transfer the log information to RVM would also be advantageous. Currently, we make one call to RVM for each change record on the log and RVM validates each such call independently. An interface that allowed a group of change records to be processed together would allow RVM to reduce its overhead without compromising safety.

5.5.3 Write-Ahead Logging

RVM delays capturing the value of a redo record until its transaction commits. For our purposes it is acceptable to capture the value when the change record is first logged. This would allow RVM to use write-ahead logging aggressively.

We could also take advantage of write-ahead logging by promoting newly persistent objects eagerly. Currently no promotions are done until commit. Earlier promotion would allow the cost to be absorbed in existing collection work and would be especially beneficial for long running transactions like those in the compiler.

5.5.4 Log Editing

When the RVM log fills with transaction records it must be emptied. RVM does this by applying the log records to the data files and truncating the log. However, if the garbage collector has flipped the stable from-space and the stable to-space, then many log records are obsolete, because they contain changes to the heap that has been reclaimed by the collector. It would be much more efficient to inform the stable log manager (RVM) that these log records can be removed from the log entirely.

In our implementation, this optimization would also be very useful whenever the Copy thread bypasses RVM and writes directly to stable to-space. Currently we must force an RVM log truncation before issuing these writes in order to ensure that RVM is not holding any old log records that apply to stable to-space. It would be much more efficient to discard these log records instead.

Garbage collectors sometimes benefit from similar features in data caches and virtual memory systems [7]. When the garbage collector reclaims a semi-space it is better that cache lines and virtual memory pages that contain reclaimed data be reset to a zero-fill-on-demand status because there is no need for the underlying memory system to preserve their old contents.

Recently the designers of RVM have begun the implementation of an incremental log truncation mechanism [20]. This new feature will subtly change the semantics of RVM. Currently the only changes to virtual memory that RVM applies to stable storage are the changes that the client explicitly logs via the RVM interface. Our collector takes advantage of this property when it overwrites portions of volatile from-space with relocation information containing the new address for each replicated object. Because the collector does not notify RVM of these changes, the original contents of these locations will be restored after a failure. The proposed incremental truncation technique will use pages of the virtual memory to update the disk file instead of using the log entries. For our application this would result in an unacceptable loss of committed data.

5.6 Recovery Performance

Although we have not measured recovery performance, we believe that the cost of recovery is almost entirely attributable to RVM. There are two phases of recovery processing in our implementation. First, the RVM log manager must recover the last committed physical heap image. Second, the transaction manager must undo any uncommitted modifications that are present in the persistent heap. The cost of processing the undo log is small relative to the cost of RVM recovery. RVM recovery must perform disk operations to reconstruct the committed contents of the stable heap from its stable log,

6 Related Work

The basic literature on uniprocessor garbage collection techniques is surveyed by Wilson [21]. Discussions of persistent heaps and language support for transactions appear in work on Persistent Algol [4] and Argus [15]. Garbage collection algorithms based on replication appear in work by Nettles and O'Toole [16, 17, 19], and an instance of the basic technique is also described by Huelsbergen and Larus [13].

We are aware of only one other implementation of a concurrent collector for a persistent heap. Almes [1] designed and implemented a mark-and-sweep collector for use in the Hydra OS for C.mmp. The collector is based on Dijkstra's concurrent mark-and-sweep algorithm [11]. There are two key differences between this work and our own. First, it cannot relocate objects and therefore offers no opportunities for heap compaction or clustering of objects for fast access. Second, although it works in the context of persistent data, it is not designed for use in a system with transaction semantics. Because of the rather unusual environment in which it operated, we cannot make any performance comparison between it and our work.

We have recently learned of ongoing work in the EOS system [12]. The EOS design proposes to combine a marking process and a compaction process. An implementation of EOS is underway but details are not available at this time.

There is a long history of incremental and concurrent copying collectors dating back to Baker [5]. These collectors require the client to access the to-space version of an object during collections and sometimes force objects to be copied so that the client may access them. The technique of Ellis, Li, and Appel [3] enforces this restriction by using virtual memory protection traps to detect certain client accesses and perform required collector work. In contrast, our technique does not constrain the order in which objects are copied nor does it require any special operating system support. We believe that the ability to freely choose the traversal order is especially important in systems that may need to optimize access to the disk.

There are two earlier designs of concurrent copying garbage collectors for persistent heaps, both based on the Ellis, Li, and Appel algorithm. Detlefs [10] described how to apply this algorithm in the transactional environment of Avalon/C++ [9], while Kolodner [14] worked in the context of Argus. In Detlefs's design, the programmer must explicitly manage object persistence at the time of allocation; Kolodner supports orthogonal persistence.

Neither of these designs was completely implemented. We believe this is due to the complexity of using the to-space invariant in a transactional setting.

In to-space techniques the flip takes place at the beginning of the collection. Starting at the flip, the client uses objects that are in to-space and so client operations are reflected in the transaction log using to-space values. The log also contains from-space pointers due to transaction activity that occurred prior to the flip. Therefore, the recovery process must be able to reconstruct the relationship between from-space and to-space objects. This relationship depends on the exact sequence of copy operations performed by the garbage collector. In practice, this means that essentially the entire garbage collection process must be recoverable.

There are two primary disadvantages to making the collection recoverable: complexity and cost. Added complexity arises because each step of the collection must be recoverable, thereby greatly increasing the interaction of the collection algorithm with the logging and recovery algorithms. Both Kolodner and Detlefs explain how each step of the collection algorithm is logged and recovered. These arguments are quite detailed and complicated. Added cost arises because each step of the collection adds to the logging burden of the system. Detlefs's design attempts to minimize the amount of log traffic but at the cost of introducing some synchronous writes. Kolodner requires no synchronous writes but has greater log traffic.

Our design avoids the problems described above because there is never any need for the recovery process to deal with to-space values. In our system, the client only uses from-space objects and only the flip need be coordinated with the transaction manager. Our collector can be made recoverable at the cost of added log traffic, but for many applications this may not be necessary.

7 Future Work

We expect to support multiple client threads by using the Venari transaction model [22]. We plan to examine closely the remaining sources of delay in the pauses due to the concurrent collector. We also plan to experiment with a few simple optimizations that may compress the logs. One lesson learned from the performance measurements is that there are several desirable features that are candidates for addition to the RVM log manager.

Our current implementation does not support the restart of a partially completed garbage collection, but the changes required are minimal. Because the client uses only from-space, the recovered state of to-space is of little importance. The state of the replicating garbage collector can be recovered as long as its volatile data structures are periodically checkpointed to stable storage.

We also expect that replicating garbage collection will prove to be valuable for very large persistent heaps that are accessed using swizzled internal and external representations, via a cache, or in a distributed programming environment containing multiple volatile heaps. There are two primary advantages of replicating collection in these configurations: The client and the garbage collector need not be as tightly coupled as in other garbage collection algorithms and therefore system features that depend on knowledge of object modifications and object locations are easier to build.

8 Conclusions

We have implemented a concurrent compacting garbage collector for a transactional persistent heap. Our design is based on replicating garbage collection and uses a log that is shared with the transaction manager. The prototype implementation demonstrates that client activity can continue during the garbage collection of stable data.

The experimental measurements show that concurrent replicating collection offers garbage collection pauses that are much shorter than for stop-and-copy collection. The interrupts suffered by the client are small in comparison to transaction commit latencies and are independent of stable heap size. Transaction performance provided by the prototype is good. The use of garbage collection in the transaction commit processing adds little overhead; commit performance remains dominated by the underlying log manager. Our design offers garbage collection performance that will be useful to real-time operating systems applications that require safe persistent storage.

Acknowledgments

We would like to thank DEC SRC, where replicating garbage collection was conceived during a very pleasant summer. Special thanks goes to Hank Mashburn for his implementation of Recoverable Virtual Memory, to Jeannette Wing for serving as our shepherd, and to Forest Baskett for providing us with SGI processor boards. We also thank Puneet Kumar for help running the Coda RVM benchmark on our machine and Sally McKee for teaching us about jgraph. Finally, we thank our readers: Brad Chen, Mark Day, Sanjay Ghemawat, Robert Gruber, Rich Lethin, Sally McKee, Mark Reinhold, Brian Reistad, Mark Sheldon, Franklyn Turbak, Mary-Claire van Leunen, and several anonymous referees.

References

[1] Guy T. Almes. Garbage Collection in a Object-Oriented System. Technical Report CMU-CS-80-128, Carnegie Mellon University, June 1980.

[2] A. Appel. Simple Generational Garbage Collection and Fast Allocation. *Software–Practice and Experience*, 19(2):171–183, February 1989.

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time Concurrent Garbage Collection on Stock Multiprocessors. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 11–20, 1988.

[4] M. P. Atkinson, K. J. Chisolm, and W. P. Cockshott. PS-Algol: an Algol with a persistent heap. *SIGPLAN Notices*, 17(7):24–31, July 1982.

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

[6] R. G. G. Cattell. An Engineering Database Benchmark. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 247–281. Morgan-Kaufmann, 1991.

[7] Eric Cooper, Scott Nettles, and Indira Subramanian. Improving the Performance of SML Garbage Collection using Application-Specific Virtual Memory Management. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 43–52, June 1992.

[8] Transactions Processing Council. TPC-B. In Jim Gray, editor, *The Benchmark Handbook for Database and Transaction Processing Systems*, pages 79–114. Morgan-Kaufmann, 1991.

[9] D. L. Detlefs, M. P. Herlihy, and J. M. Wing. Inheritance of Synchronization and Recovery Properties in Avalon/C++. *IEEE Computer*, pages 57–69, December 1988.

[10] David L. Detlefs. Concurrent, atomic garbage collection. Technical Report CMU-CS-90-177, Carnegie Mellon University, October 1990.

[11] E. Dijkstra, L. Lamport, A. Martin, C. Scholten, and E. Steffens. On-the-fly Garbage Collection:An Exercise in Cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.

[12] Olivier Gruber. Eos: An Environment for Persistent and Distributed Applications in a Shared Object Space. host ftp.inria.fr directory INRIA/Projects/RODIN file Olivier.Gruber.phd.ps.Z, December 1992.

[13] Lorenz Huelsbergen and James R. Larus. A Concurrent Copying Garbage Collector for Languages that Distinguish (Im)mutable Data. In *Proceedings of the 1993 ACM Symposiym on Principles and Practice of Parallel Programming*, 1993.

[14] Eliot K. Kolodner. Atomic Incremental Garbage Collection and Recovery for a Large Stable Heap. Technical Report MIT/LCS/TR-534, Massachusetts Institute of Technology, February 1992.

[15] B. Liskov and R. Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems*, 5(3):382–404, July 1983.

[16] Scott M. Nettles and James W. O'Toole. Real-Time Replication Garbage Collection. In *SIGPLAN Symposium on Programming Language Design and Implementation*, pages 217–226. ACM, June 1993.

[17] Scott M. Nettles, James W. O'Toole, David Pierce, and Nicholas Haines. Replication-Based Incremental Copying Collection. In *Proceedings of the SIGPLAN International Workshop on Memory Management*, pages 357–364. ACM, Springer-Verlag, September 1992.

[18] S.M. Nettles and J.M. Wing. Persistance + Undoability = Transactions. In *Proceedings of the 25th Hawaii International Conference on System Sciences*, volume 2, pages 832–843. IEEE, January 1992.

[19] James W. O'Toole and Scott M. Nettles. Concurrent Replication Garbage Collection. Technical Report MIT-LCS-TR-570 and CMU-CS-93-138, Massachusetts Institute of Technology and Carnegie Mellon University, 1993.

[20] M. Satyanarayanan, Henry H. Mashburn, Puneet Kumar, David C. Steere, and James J. Kistler. Lightweight Recoverable Virtual Memory. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, December 1993.

[21] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In *Proceedings of the 1992 SIGPLAN International Workshop on Memory Management*, pages 1–42. ACM, Springer-Verlag, September 1992.

[22] J.M. Wing, M. Faehndrrich, N. Haines, K. Kietzke, D. Kindred, J.G. Morrisett, and S.M. Nettles. Venari/ML Interfaces and Examples. Technical Report CMU-CS-93-123, Carnegie Mellon University, March 1993.

[23] Benjamin Zorn. The Measured Cost of Conservative Garbage Collection. *Software—Practice and Experience*, 23(7):733–756, July 1993.