Automatic Storage Management for Systems with Real Time Constraints

Steven L. Engelstad and James E. Vandendorpe

AT&T Bell Laboratories
Naperville, IL 60566

ABSTRACT

A "garbage collector" programmed in C reclaims objects by executing during the allocation of new objects. After a dynamically scheduled number of object allocations, the application is delayed while lost resources are recovered. The average delay is 6 ms on a Sun3™, 2 ms on a Sun4™/SPARC1, and 0.5 ms on a Sun4/SPARC2 with bounded worst case delays of 10 ms, 5 ms, and 2 ms respectively. We conjecture that this is the tightest real-time bound of its kind.

On the applications we have measured, the percentage of the processor devoted to reclamation averages 7% on a Sun3 and 4% on a Sun4. The system is routinely used to run interactive sessions that last weeks at a time.

To achieve these results we refined a generational garbage collector algorithm published by Carl Hewitt and Henry Lieberman [1][2]. The redesign sought an object-oriented telephony infrastructure that would offer improved audit and field-update capabilities. The work done in changing the Hewitt/Lieberman algorithm involved four parts:

1. The algorithm was made to run as a coroutine resumed by the allocator after every N allocations. When the allocation is sufficiently infrequent and the perceived yield in returned storage is still sufficiently large, the garbage collector runs as a low priority process.
2. The algorithm was divided up into eight phases and fundamental tasks were defined within each phase to bound the amount of real-time each cycle would use before returning to the allocator.
3. The notion of forward progress within a section was defined and controls were established to insure that the algorithm would complete.
4. The real-time implications of scanning the process stack were worked out in detail.

Two additional changes to the Hewitt/Lieberman algorithm were made to facilitate a telephony application:

1. The integral compaction portion of the Hewitt/Lieberman algorithm was separated into an optional real-time process.
2. The garbage collector will arrange for a *finalize* destructor method of an object to be called prior to deallocation that will recover and audit non-memory resources. *Finalize* methods have the opportunity to save a reference to previously unreachable objects that will cause the garbage collector to preserve them.

Our algorithm is implemented within the Calico [5] programming system.

™ Trademark of Sun Microsystems, Inc.

1. Hewitt/Lieberman Algorithm

The H/L algorithm is described in [1] but was originally published in [2]. The H/L algorithm builds on ideas developed by the Baker algorithm [3]. The H/L algorithm reduces the amount of work that needs to be done on the basis of two statistical observations:

1. Young objects are more likely to become unreferenced.
2. Objects mostly reference other objects that are either about the same age or are older.

To take advantage of the first observation, the algorithm divides objects into generations on the basis of the historical period in which they were created. The reclamation effort is focused on the most recent objects since these are the objects most likely to become unreferenced. The algorithm selects a “condemned generation” and all generations younger than this generation are taken as part of a condemned region.

The second observation was used to design an optimization that would minimize the effort spent examining objects older than the oldest condemned generation. Whenever an object in an older generation stores a reference to an object in a younger generation the reference is indirected through an entry table that belongs to the younger generation of objects.

The entry table data optimizes garbage collection because it simplifies the effort needed to search older objects that reference younger objects. The condemned region must be searched in the mark sweep sense, but all references from the older generations into the condemned region can be found in the cells recorded within the entry tables. This is advantageous because there are many more old generations than young generations and because the number of objects that need to be stored within an entry table is very small.

By concentrating effort on the reclamation of objects in the youngest generations, the H/L algorithm expends less effort and reclaims unused objects faster. Objects that are saved are relocated in the Baker fashion into a TO generation. The size of generations are based on the size of the storage area that each generation can hold, a fixed number of bytes representing an arbitrary number of objects. Since generations all have the same size, a TO generation will always accommodate the relocation of a FROM generation even if all of the objects are preserved. As a bonus, a small number of H/L generations means that the TO region is also small (i.e., always the size of the condemned region). In contrast the Baker algorithm’s compaction technique requires 2N storage cells for every N cells actually required.

2. Our Memory Reclamation Algorithm

The H/L algorithm, while efficient, does not provide mechanisms for strict bounds on the real-time consumed by the algorithm and is therefore unsuitable for use in a system with real-time constraints (e.g., electronic telephone switching systems).

Our algorithm refines the H/L algorithm into a hierarchy of tasks. The top level task involves selecting a condemned region and reclaiming all resources associated with unreachable objects within the condemned region. We call this task the *duty cycle*. The duty cycle is segmented into eight phases. Each phase accomplishes a specific function and takes one or more invocations of the garbage collector to complete. Each individual invocation is called a *cycle*. Each cycle is composed of *fundamental* tasks. These fundamental tasks are defined such that

- the memory reclamation algorithm may terminate a cycle after any such task and return to system processing and
- each task takes a known amount of real-time to complete.

These two properties allow us to bound the real-time taken during each cycle of garbage collector activity. In general the garbage collector executes successive fundamental tasks until either the defined real-time bound has been reached or the current phase has completed.

It is not sufficient for each invocation of the garbage collector to meet a strict real-time bound. The garbage collector must also perform its primary function (recovering unused resources) such that the system remains in a stable state. Each phase of our algorithm employs a set of forward progress measures that are used to determine if memory reclamation is progressing at a sufficient rate. These forward progress measures are a combination of phase dependent and duty cycle dependent heuristics which are evaluated every garbage collector cycle. Forward progress is controlled by altering the frequency at which garbage collector cycles are taken. A cycle is taken every n object allocations where n may vary. The value of n is called the cycle steal rate because cycles are interleaved with application processing. When forward progress is not made, the cycle steal rate is adjusted so that cycles will occur more often. The opposite happens when forward progress becomes excessive. The granularity at which these decisions are made determines how quickly the system responds to changes in application activity. Our experience has been that a fairly simple set of forward progress measures are reasonable and sufficient for most of the applications employing this garbage collector. However, some applications have required extensions to the forward progress measures and/or tuning of parameters associated with the forward progress heuristics.

We replaced the H/L entry table data structures with "remembered set" data structures (as described by Ungar in [4]). A remembered set stores the location of the storage cell in an older generation which may contain a reference to an object in its associated generation. This has the advantage of not indirecting through an entry table when references from younger objects to older objects are traced.

The eight phases within a duty cycle are:

1. *Condemned region selection* - Determine which generations are to be scanned for unreferenced objects during the next duty cycle. This involves selecting a generation which is to be the oldest generation of the condemned region. This selection is table driven and will select a younger generation most frequently, middle aged generations less frequently, and old generations least frequently.
2. *Generation data structure compaction* - Compact garbage collector specific data structures. In particular, those data structures that represent generations and remembered set information.
3. *Remembered set scan* - Use the remembered set data structures to find all objects within the condemned region referenced by objects older than the condemned region.
4. *Condemned region scan* - Determine the transitive closure of all objects referenced from objects older than the condemned region. A second function of this phase is to construct a list of all objects which are candidates for lastwill processing (i.e., they define a *finalize* method and are not currently known to be reachable).
5. *Mark end of condemned region* - Determine the end of the condemned region. This marks the youngest generation as the last generation within the condemned region. Any further objects allocated during this duty cycle will become part of generations younger than the condemned region and will not be subject to garbage collection until the next duty cycle.
6. *Process stack scan* - Find all objects within the condemned region which are only referenced from process stack objects.
7. *Lastwill processing* - Arrange for the *finalize* method to be called on all unreachable objects within the condemned region that have supplied a *finalize* method.
8. *Garbage reclamation* - Reclaim the storage associated with all unreachable objects within the condemned region.

Unlike the H/L algorithm, compaction is not an integral part of our garbage collector and the size of a generation is determined by the fixed number of objects that each generation data structure within a logical generation can hold. A real-time compaction process can be run to pack the active objects (header and data) into one area. This process is orthogonal to memory reclamation.

2.1 Assignment Trapping

Assignments made during the memory reclamation process may affect the garbage collector process in two ways. First, it may remove objects from the set of reachable objects by assigning over existing object references. Second, it may move an object reference from an area not yet scanned by the garbage collector into an area already scanned (the *keep away* problem). The first case where reachable objects become unreachable can be ignored. Such objects may not be reclaimed during the current duty cycle, but will be reclaimed the next time their generation is condemned. However, all objects whose references move during a duty cycle must not be missed. Therefore, the garbage collector requires that all assignments are trapped. Two types of action may need to be taken at every assignment. First, if the referenced object is in a younger generation than the referencing object, then a remembered set entry must be added to the generation of the referenced object. This check needs to be made regardless of the state of the garbage collector process. Second, if the referenced object is within the condemned region and is currently not marked as referenced, then it may need to be saved depending on the current phase of the garbage collector.

New objects can only be added to the set of reachable objects by being pushed on a process stack. Objects created up to the process stack scan phase may be reclaimed during the current duty cycle. Objects created during the process stack scan phase and beyond will fall outside the condemned region and will not be condemned until the next duty cycle.

2.2 Notable Aspects of the Algorithm

2.2.1 Process Stack Scan Phase

Perhaps the most difficult phase of our algorithm to bound (i.e., define fundamental tasks as described earlier) was the process stack scan phase. The creation of references from process stacks to other objects is not trapped. This means that object references can move within a stack undetected and there may exist references from stacks older than the condemned region into the condemned region. We made the following observations about process stacks and object references contained within them:

1. There are relatively few process stacks in the system.
2. The *active* region of a process stack is relatively small.
3. A reference can only move between stacks via an intermediate assignment.
4. The only object references of concern are those in the active portion of the stack when the scan of the stack begins.

Because of observation 1 it is practical for the garbage collector to keep a list of all extant process stack objects. An internal top of stack pointer is maintained which allows easy identification of the active portion of the stack and greatly reduces the amount of stack data which needs to be scanned for object references. Observation 3 is true because there are no virtual machine operations that move an object reference from one process stack to another and there are no methods on process stack objects that cause an object reference to be stored directly into a process stack. The fourth observation says that no new object references can be created on the process stack that aren't already accounted for. This is because new objects will be younger than the condemned region, assignment will catch object references moving between stacks, and any other references pushed onto stacks are reachable via objects other than process stacks (hence already found in previous phases).

The process stack scan phase examines each process stack object on its list and for each such object marked as reachable it scans the process stack object for references. At all times the data remaining to be scanned within the stack is known. If the amount of time remaining in the current garbage collector cycle reaches the time needed to copy the remainder of the stack to a temporary location, then the copy is done and the cycle terminates (in bounded time). The next cycle begins by scanning the temporary copy of the process stack data. No object references will be missed because of observation 4.

2.2.2 Lastwill processing

Our algorithm provides for unique and powerful facilities for the recovery of non-memory resources. If a class of object defines a *finalize* method then its instances will be tagged as lastwill objects. When such an object becomes unreachable, it is scheduled to receive its *finalize* method. An object is scheduled by first assigning it to a special *to-be-finalized* object. This assignment attaches the object (and all objects reachable from it) to the set of reachable objects. The object is no longer tagged as a lastwill object to prevent repetitive finalize action. A process is then scheduled which, when run, sends each object directly referenced by the *to-be-finalized* object a *finalize* method. The reference from the *to-be-finalized* object to the object sent the *finalize* method is then destroyed. Also, if an object during its *finalize* method causes a reference from a reachable object to itself (or some other object it references) to be created it can effectively re-attach a previously unreachable object to the set of reachable objects. If an object does not cause itself to be re-attached to the set of reachable objects or cause itself to be tagged as a lastwill object, it will be garbage collected in a later duty cycle.

2.3 Memory Compaction

The storage allocation and deallocation functions cooperate in maintaining memory in a mostly compact state. Pools are created of the maximum number of each size required. An important category of dynamic memory systems are those that continuously reuse a working set of objects of the same size. For this type of system, continuous compaction is an unnecessary real-time expense. When classes are added, deleted or changed on these systems, however, allocation sizes may change. Compaction may then be required to establish new pools of the appropriate sizes. The compaction algorithm is de-coupled from the garbage collection system and may be run as a separate low-priority process.

3. Conclusion

The memory management algorithm described here has been implemented within the Calico system [5], runs on conventional hardware, and is reliable. The most significant characteristic, however, is in the 0.5-10 ms bounds placed on the collector's cycle times. These real-time bounds are the shortest of any commercially available products known to us.

REFERENCES

[1] H. Lieberman and C. Hewitt, “A Real-Time Garbage Collector Based on the Lifetime of Objects,” *CACM*, Volume 26, Number 6, June 1983.

[2] H. Lieberaman and C. Hewitt, “A Real Time Garbage Collector that can recover Temporary Storage Quickly,” MIT Artificial Intelligence Laboratory, A.I. Memo No. 569, April 1980.

[3] H. G. Baker, “Actor Systems for Real-Time Computation,” MIT Laboratory for Computer Science, MITILCSITR-197, 1978.

[4] D. Ungar, “Generation Scavenging: A Non-disruptive High Perfornance Storage Reclamation Algorithm,” *Software Engineering Notes*, Volume 9, Number 3, May 1984.

[5] S. Engelstad, K. Falck, W. Montgomery, J. Neumann, R.Straubs, J. Vandendorpe, M. Wilde, “A Dynamic C-Based Object-Oriented System for UNIX,” IEEE Software, May 1991, pg. 73 - 85.