HOW REAL IS "REAL-TIME" GC?

P. T. Withington
Symbolics, Inc.
Eight New England Executive Park, East
Burlington, MA 01803, U.S.A.
ptw@Jasper.SCRC.Symbolics.COM

ABSTRACT

A group at Symbolics is developing a Lisp runtime kernel, derived from it's Genera[1] operating system, to support real-time control applications. The first candidate application has strict response-time requirements (so strict that it does not permit the use of paged virtual memory). Traditionally, Lisp's automatic storage-management mechanism has made it unsuitable to real-time systems of this nature. A number of garbage collector designs and implementations exist (including the Genera garbage collector) that purport to be "real-time", but which actually have only mitigated the impact of garbage collection sufficiently that it usually goes unnoticed by humans. Unfortunately, electro-mechanical systems are not so forgiving. This paper examines the limitations of existing real-time garbage collectors and describes the avenues that we are exploring in our work to develop a CLOS-based garbage collector that can meet the real-time requirements of *real* real-time systems.

1 INTRODUCTION

There have been many proposals and implementations of "real-time" garbage collectors, where a real-time collector is typically defined as one such that "the execution of the main program never be suspended for the long time that garbage collection usually requires" [Wadler 76], or such that "the programmer would still be assured that each instruction would finish in a reasonable amount of time" [Baker 78]. Under this definition, even generational garbage collectors have recently been touted as being nearly real time. [Ungar 84] In previous work, the primary thrust of the design of real-time garbage collectors has been to show that the garbage collector can do its job in unnoticeable increments and still "keep up" with the running program, without requiring significantly greater total storage than traditional garbage collectors (which have the luxury of stopping the running program while they clean up). This paper considers whether any existing garbage collection scheme can be considered real-time enough to qualify under the original statement of the problem "when stringent upper bounds are placed on the maximum execution time for each List operation performed" [Knuth 69].

[1]Symbolics®, Genera®, and Ivory® are registered trademarks of Symbolics Incorporated.

2 BACKGROUND

One of Lisp's primary features is it's automatic storage management system that frees the programmer from that typically onerous task. Lisp's storage management is implemented using one of any number of techniques that are commonly know as "garbage collection". Garbage collection (or perhaps more contemporarily, recycling) has three primary facets, each of which impacts the overhead associated with and delays introduced by automatic storage management and each with a range of solutions. For the purposes of our real-time system, we are concerned mostly with delays introduced by garbage collection. We are willing to trade-off increased garbage-collection overhead for a guarantee that introduced delays are both bounded and small with respect to the response-time requirements of the application.

2.1 When to Collect

The first facet of garbage collection is a policy to decide *when* to perform garbage collection.

Traditional collectors wait until the *last minute*. This reduces the interference with the running program, except that when it finally must run, there can be a long interruption while the entire address space is scanned looking for recyclable objects.

At the other end of this spectrum, one could garbage collect *continuously* on a separate, parallel processor. Aside from interlocking problems, this approach would seem to guarantee the best performance of the running program with the least possibility of it running out of free storage. Unfortunately, it is also a great waste of processor power.

Somewhere in between are the strategies of current real-time garbage collectors, which try to run *just in time* to avoid overflowing available storage, thus avoiding unnecessary garbage collection overhead. Another criteria that might be used would be *when worthwhile*; if there were some way of knowing that there is a lot of potentially recyclable material at a particular time. or that the application will be idle for some time.

The policy as to when to start garbage collection has little direct impact on garbage-collection induced delays. It has indirect impact, as noted in the case of the last-minute policy, because it can have implications on how the other two aspects of garbage collection are addressed. The direct impact of the policy on when to collect is bounded by any synchronization required with the running program in order to make the decision to collect. Typically, there is no synchronization required to make this decision, as it is based solely on the ratio of used to unused storage.

2.2 What to Reclaim

The second facet of garbage collection is a mechanism to identify reclaimable objects.

Conventional programming languages typically use *explicit* requests by the running program to determine what should be reclaimed. This technique is error-prone, because in a complex program it is possible to accidentally release an object when it is still in use by some other part of the program. Nonetheless, explicit reclamation can have a place for objects that are closely managed but created and destroyed at extremely high rates or live for very brief intervals.

At the other end of this spectrum is reclamation by *discovery*. Many garbage collectors use this technique to find recyclable objects with no effort on the part of the running program. The discovery process must be correct: an object mistakenly classified as recyclable will break the running pro-

gram. On the other hand, the discovery process does not have to be complete: if it misses some recyclable objects, it is less efficient; but this type of sacrifice is often made to reduce the overhead of garbage collection. (Generational collectors use partial discovery in the sense that they limit discovery overhead by only trying to discover recyclable objects in specific subsets of all storage.)

Again, there is a middle ground of *cooperation* between the running program and the garbage collector. It may still appear automatic to the programmer, but there may be code or hardware to force the running program to cooperate with the garbage collector in enumerating recyclable objects. Typical instances of cooperation are "reference counting" and "read and/or write barriers".

The problem of how to identify objects for reclamation has a range of impacts on the delays introduced in the running program, depending on the choice of solution. These impacts are discussed more fully below, under **DESIGN**.

2.3 How to Recycle

The third facet of garbage collection is a mechanism to recycle unused objects.

One possible choice of how to recycle would be to *reuse* objects (the same way bottles used to be returned to be refilled). Reuse requires that the various sizes of containers be kept sorted out, but depending on how the sorting is done, this technique can lead to low overhead (by analogy, you don't have to re-make objects from raw materials). A possible problem is that you might misallocate your supplies and run out of a particularly popular size.

At the other end of this spectrum is *recycling*— collecting all the unused objects and turning them back into raw materials to make new objects from. This solves the problem of misallocated resources, but is potentially subject to a longer lead time (to recover raw material from unused storage, you typically must copy or compact all the pieces still in use).

The choice of how to recycle has minimal impact on garbage-collector introduced delays if done in a reasonable fashion. The only synchronization requirement is at the point where recovered storage is added back to the free pool that the running program is taking from. Methods exist for implementing the recycling process in a discrete fashion, where each step either has a bounded execution time or is interruptible.

3 PROBLEM STATEMENT

We believe that a garbage collection system as good as the existing Genera system will satisfy the needs of our Lisp kernel for the most part. The exception is in routines that are directly controlling electro-mechanical equipment. Our plan is that these routines can be assigned a priority (or run in a high-priority process) that will inform the garbage collector that it must be especially careful not to introduce any undue delays during their operation.

Typically, this will mean that the garbage collector will be prevented from *starting* a collection when one of these critical routines in running. We also believe that the uninterruptible time required to start a collection (in the Genera system, this is the time to "flip", which involves setting up the hardware oldspace registers and copying the root state) can be minimized by careful choice of what to include in the root state and what can be deferred for later scavenging.

Where the problem arises is when a garbage collection is already *in progress* when a critical routine is called into action (an unpredictable event, due to the nature of the application). In this case, the existing Genera garbage collector (and all other non-

reference-counting garbage collectors that we know of) can introduce unbounded delays when the running process and the garbage collector must synchronize in identifying the objects to be reclaimed.

In Genera this unbounded delay comes from the "read barrier", which prevents the running process from creating new references to objects that have been declared candidates for recycling by the garbage collector. When the running process touches an object that the garbage collector has declared is a candidate for recycling, but which it has not yet made a decision on, the running process is (transparently) forced to copy the object to protect it from collection (effectively doing the garbage collector's job for that object immediately). If a high-priority (fixed response-time) task is started immediately after a garbage collection has been started, it may encounter many such objects. While each copy operation is bounded by the size of the object, the total number of objects referenced by the running task can lead to an arbitrary delay. Copying time is proportional to the object size in the current Genera system, although we know of techniques to mitigate this time by doing partial copying of large objects. But, even with the partial copying solution (which typically involves copying at least one virtual-memory page of the object, plus setting up the remaining pages to fault) the overhead of copying is significant with respect to the normal cost of an object reference (a page worth of reads and writes, as opposed to a single read). This problem is the crux of our current research.

4 DESIGN

The design of the garbage collector for our Lisp kernel draws heavily on the Genera garbage collector design. Because there is no demand-paging in our system, the garbage collection design is simpler— there is no concern of locality of reference or latency in scanning pages.[2]

New in our design is an object-based implementation of the storage subsystem. Our implementation is in a subset of CLOS (a subset, to avoid circular dependencies). This implementation choice is driven by a desire to experiment with different garbage-collection philosophies, but it also has the benefit of letting us support several different policies by dividing storage into areas and making the policy and mechanism decisions generic on area.

4.1 Deciding when to collect

The decision of *when* to recycle in our system is made using a *just in time* policy based on Baker's algorithm [Baker 78] as described in [Moon 84].[3] The system has adjustable parameters to balance garbage collection with the running program, delaying garbage collection as long as possible while still guaranteeing that the garbage collector can keep up with the running program without running out of storage. A research project is to develop a more effective method for deriving the correct parameter settings other than by trial-and-error. Currently, conservative, static settings are used, guaranteeing sufficient free storage at

[2]Our design includes a provision for "computable" areas: areas of virtual memory where the contents of the page is computed by calling an area-specific function and where the physical storage backing a virtual page is recovered by calling an area-specific function. This mechanism can be used to extend our system to support demand paging by having an area whose compute and release functions are "page-in" and "page-out".

[3]There is both a local policy on generations (by statically set capacities) and a global policy that monitors the total storage consumed and will initiate a collection in time to finish while maintaining an upper bound on overhead. Once a collection is started, the overhead is adaptively adjusted to ensure finishing even if the collection is stalled by high-priority processes.

the expense of the productivity of the running program and the garbage collector. The one modification to the Genera design on when to recycle is factoring in the priority of all currently running processes: when a process with a deadline priority is running (that is, able to run, although potentially being time-sliced), the garbage collector will not initiate a collection. As in Genera, a collection can also be initiated manually, at the behest of the application, to allow a *when worthwhile* policy. (The first application has known periods of inactivity which are ideal times to run garbage collection; but this does not obviate the need for on-going garbage collection during high-activity.)

4.2 Choosing how to recycle

We plan to offer several choices of *how* to recycle, as Genera does. By dividing all of storage into *areas*, different recycling policies can be offered on each area. We plan to have areas that implement a reuse policy, typically for large similarly-sized objects that are used for short periods[4]. A special kind of area will offer efficient recycling with low overhead by being managed as a stack. Most areas, though, will use a copying collector to recycle unused objects. We believe that the overhead of copying live objects can be made arbitrarily small by one of several techniques: really large objects will be segregated into special areas where they can be page-aligned, and copied by remapping using virtual memory hardware (forwarding i implemented by keeping a table of the moved objects). Smaller objects will either be of negligible size or amenable to partial copying, again using virtual memory hardware to catch references to uncopied portions (in this case, the pages associated with the copy are create-on-demand, with either a table or per-object pointer to find the original).

[4]Genera implements explicitly managed objects using a facility called **Resources**. Resources are implemented totally in "user" code, with no support from the storage management system except when the resource manager changes the total number of objects in the resource: objects are added by allocating them in the normal way, and removed by making sure there are no references to them. The objects managed by the resource are not garbage collected, because they are always referenced by the resource. In our real-time kernel, we feel we can do a better job by putting this facility into the storage management system directly. We believe we can offer the benefits of explicitly managed objects and also some of the safety of an automatic system by letting the automatic system know about these explicitly managed objects.

4.3 Discovering what to reclaim

For the *just in time* policy to not introduce large delays, the discovery mechanism must run in parallel with the application. For discovery to do a correct job, there must be some synchronization between the collector and the application: they must synchronize when determining the "live" data (otherwise live data could be mistakenly assumed dead, or *vice versa*— the former will result in a malfunction, the latter in inefficient use of memory).

We examined each mechanism we know of for determining recyclable objects. As pointed out above, these mechanisms range from the application doing all the work, to the collector doing all (or nearly all) the work. None of the existing mechanisms was totally satisfactory:

4.3.1 Explicit mechanism

While we will support explicit deallocation for special cases, we dismiss using it in general as not being in the spirit of Lisp.

4.3.2 Reference Counting

We considered reference counting as a mechanism. Reference counting solves the synchronization problem similarly to the explicit mechanism, in that the application does all the work of determining what is recyclable— but it is an automatic mechanism. Reference counting introduces a fixed, bounded delay on each write opera-

tion: incrementing the count on the object for the new reference, and decrementing the count on the previously referenced object, if any. For predictability, reference counting seems ideal. We have shelved it as a choice because we believe we can develop a solution that does not introduce the fixed time overhead on *all* writes, the overhead of a reference-count field on each object, and because of reference-counting's flaw of not being able to recover circular structures.

4.3.3 Stopping

The simplest solution to synchronizing the application and the collector during discovery is to simply stop the application. If we could develop a fast enough algorithm for discovery (or if the hardware were fast enough and the amount of storage to be collected over small enough), this solution would work. This solution is reasonable in a system where storage is partitioned and only partial collections are ever performed. This solution does work for many Lisp systems where the absolute real-time constraint is measured in tenths of a second. This solution is not feasible for our application, both because of our real-time constraints and because of an availability constraint that implies full collections must also be periodically performed without greatly increasing delays.

If the application cannot be stopped for the entire discovery process, we need at least to make a good approximation to stopping it. The problem is to get a consistent "snapshot" of the reachability of all objects while the application is actually changing things out from under us. The trick used by the next two mechanisms is to stop the application while snapshotting the root of the reachability tree and then utilizing hardware virtual-memory support to catch any changes the application makes to reachability "behind the back" of the collector.

4.3.4 Write Barrier

A write barrier can be used to catch new references created by the application in storage the collector has already scanned for live data. Using a write barrier, the collector can scan all of storage completely in parallel with the application except for a final "cleanup" phase when it corrects its conception of the in-use objects by re-scanning those that were modified during the parallel phase.[Boehm 91] Multiple parallel phases can be run to reduce the size of the clean-up phase, but in the end, the clean-up phase must run with the application stopped. We liked this mechanism a lot, but were concerned that we either would not be able to guarantee an upper limit on the interruption of the application caused by the clean-up phase, or, if we ran parallel phases until the cleanup phase were within fixed bounds, there might be situations during periods of high application activity where garbage collection would be delayed beyond *just in time*, leading to storage exhaustion.

4.3.5 Read Barrier

As we have indicated above, Genera uses a read-barrier to ensure synchronization during the discovery phase. Essentially, in Genera, any object the application gets to during a garbage-collection before the collector does, the application copies itself. This gives a smaller bound on the incremental delay introduced by garbage collection, since each object found in this way is copied individually (as opposed to the write barrier solution which eventually must pick a point to deal with all the changed objects *en masse*, and delay the application for the entire process). But, as we have described, we are concerned that the cumulative delays created by the read barrier immediately after a collection is started may exceed the response-time limits imposed by the application.

4.3.6 Proposal

Our proposed solution is an amalgam of the write and read barrier solutions. We propose to implement the Genera mechanism, with the modification that when a critical task is running, the read-barrier will be relaxed to a write barrier. Thus, rather than the critical task being forced to copy objects it references, it will be allowed to proceed, with the write barrier noting if the reference is stored to an object the collector has already examined. When the critical task completes, the collector must re-scan the objects so noted. Because the normal mechanism is a read barrier, we avoid the concern we had with the write-barrier mechanism of not being able to converge during a high-activity period

5 IMPLEMENTATION

Our implementation will utilize the hardware support of the Symbolics Ivory® architecture for it's read and write barriers and will make use of the architecture's invisible pointers to note forwarding. The implementation follows the Genera implementation described in [Moon 84]. The implementation is modified for the case of a high-priority task running using an algorithm similar to that of [Boehm 91], except that the Ivory architecture simplifies the implementation.

When a high-priority task is running and takes a read-barrier trap (because it references an object in oldspace), the read-barrier trap determines if the object has already been copied by the collector. If this is the case, the reference is replaced with the copy and the task continues (this is the normal Genera mechanism for this case). If the object has not yet been copied, read-barrier traps are disabled for the page in which the reference lies (and that page is noted as needing to be scanned for references by the collector when the task exits) and the task is allowed to proceed using the old object. If the old reference is stored into any object that the collector has already examined, a write-barrier trap is taken so that page can be noted as needing to be re-scanned.[5]

When the critical task exits, the collector starts a clean up phase. The cleanup involves simply scanning or re-scanning the pages noted during the high-priority task's execution. Because we use the same mechanism to note pages as we use to mark pages for scanning in the normal case, the clean-up phase is implemented by simply changing the normal scan to first re-scan pages it has already scanned that became "unscanned" due to a write-barrier trap. Once these pages have been cleaned up, the collection can continue normally. Because of the availability of invisible pointers, there is no need of recopying as in Boehm's algorithm. Finally, while we run the cleanup phase with normal-priority processing stopped, we do permit high-priority tasks to execute. We depend on the high priority tasks taking a small enough percentage of the CPU and being limited enough in their effect that the cleanup phase will eventually reach closure and return to the normal read-barrier based collection.

---

[5]Note that a new reference might be created by storing through an old reference or a new reference. In the case of storing through an old reference, the object may or may not have been copied. In the latter case, the collector has not yet scanned the object, hence no write-barrier trap need occur. In the former case, since copied objects are replaced with invisible pointers to the new copy, the hardware will automatically forward the store to the new copy. As a result, we need only enable write-barrier traps on copied objects. The same trap suffices for stores using a new reference.

The write-barrier "Trap" is actually implemented using the hardware modified bit. As the collector scans the root set, it clears the modified bit on pages it has examined (or pages it skips, because they are in new space). The clean-up phase simply involves scanning any such pages that become modified. (If we do implement demand paging, the modified bits will have to be "virtualized" to serve both purposes.)

6 SUMMARY

We have described a problem with current real-time garbage collector mechanisms that prevents their use in applications with strict response time requirements. We have proposed a new hybrid mechanism that reduces the garbage-collection induced delays for critical tasks to a few instructions for each object referenced (as opposed to instructions proportional to the size of the object). We are currently implementing this proposal in a Lisp runtime kernel to support real-time control applications.

REFERENCES

[Baker 78] Baker, H. G., "List Processing in Real Time on a Serial Computer", CACM Vol. 21 No. 4, April 1978, pp. 280-294

[Boehm 91] Boehm, H-J., Demers, A.J., and Shenker, S., "Mostly Parallel Garbage Collection", ACM SIGPLAN '91 Conference on Programming Language Design and Implementation, SIGPLAN Notices Vol. 26 No. 6, June 1991, pp. 157-164

[Knuth 69] Knuth, D. E., *The Art of Computer Programming, Vol. I: Fundamental Algorithms*, Addison-Wesley, Reading, MA, 1969, problem 2.3.5-12 (attributed to M. Minskey).

[Moon 84] Moon, D., "Garbage Collection in Large Lisp Systems", Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, pp. 235-246.

[Ungar 84] Ungar, D., "Generational Scavenging: a non-disruptive high performance storage reclamation algorithm", SIGPLAN Notices Vol. 19 No. 5, May 1984

[Wadler 76] Wadler, P. L., "Analysis of an Algorithm for Real Time Garbage Collection", CACM Vol. 19 No. 9, September 1976, pp. 491-500