Garbage Collection for Hard Real-Time Systems

Boris Magnusson and Roger Henriksson

LU-CS-TR:95-153

Also published in Proceedings of IWOOOS'95,
International Workshop on Object-Oriented Operating Systems,
Lund, Sweden, August 1995, IEEE Computer Society Press.

Department of Computer Science
Lund University

P.O. Box 118
S-221 00 Lund
Sweden

Garbage Collection for Hard Real-Time Systems

Boris Magnusson and Roger Henriksson
Dept of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: {Boris | Roger}@dna.lth.se

Abstract

This paper describes a scheme for garbage collection suitable for hard real-time applications. The approach supports both periodic high-priority processes and low-priority processes. Garbage collection work is done exclusively during execution of low-priority processes. A prototype garbage collector has been implemented for a C++ real-time kernel. The results confirms that high-priority processes can be guaranteed sub-millisecond response times and meet tight deadlines.

1 Introduction

Memory management explicitly programmed into an application is a tricky and very error-prone endeavor. Automatic memory management (Garbage Collection) is therefore a valuable technique for building large complex systems – in particular real-time applications with high demands on reliability. Unfortunately, GC has been little used in systems with strict demands on response times (hard real-time systems). The major reason for this is that common GC techniques tend to disrupt the application for too long periods of time.

Hard real-time systems are often implemented on a bare machine using a runtime kernel providing some of the functions (scheduling, memory management) otherwise provided by a general-purpose operating system.

The GC techniques developed for interactive systems are not sufficient for dealing with hard *real-time* environments where response time demands often are 1 ms or shorter. The problem is twofold: First, the GC work must be performed in chunks, small enough to be completed quicker than the demanded response time. Secondly, the GC work must be performed, scheduled, often enough not to stop the critical application processing due to lack of free memory.

In this paper we will use existing algorithms for incremental GC [Bak78, Bro84, Ben90] and focus on the scheduling problem in clock-driven systems. We use a hard real-time control system as an example of such a case and see event-driven interactive systems as a simpler, less demanding situation.

In our approach the GC work is scheduled during the processing of low-priority processes (or idle time) avoiding to interfere with the high-priority processes. The clock-driven scheduling of the processes, together with information on maximum allocation per activation for the high-priority processes, makes it possible for us to schedule the GC work so the high-priority processes can be guaranteed to perform memory allocation operations without any interference from the GC.

The presented work is novel since it not only guarantee upper bounds on GC pauses, but is removing them from the time-critical, high-priority processes. The presented scheduling mechanism has been implemented and used in a prototype environment achieving sub-millisecond response times on stock hardware.

2 Control systems

In control systems *high-priority processes* are typically performing tasks such as sampling input from the controlled process, executing short time regulator algorithms (PID algorithms etc.), and adjusting output signals. These processes are executed periodically, often as frequent as 100 times per second. The scheduling demands on these processes are that they should both start with very little delay and be completed within a guaranteed short time period, often in the order of 1 ms or less. Control theory needs the time-constraints in order to guarantee stability [AW84]. Only a small fraction of the processing time is consumed by these processes.

The *low-priority processes* are typically performing actions such as computation of reference values, supporting presentation and supervision of the state of the controlled process, changing parameters etc. These processes are thus also time critical, but the time constraints are much weaker, typically in the area of 100ms.

Demands:

- High-priority processes must start on time — they can not afford to wait for extensive GC work to complete. The GC work must be performed in small chunks or be interruptible in a time frame that is shorter than the demanded activation time.
- High-priority processes must complete in time. When calculating the worst case behavior, possible delays add up. It is therefore an advantage if the individual worst case figures for memory management operations can be kept small.

3 GC for real-time systems

Long lived applications such as control systems demand techniques to avoid memory fragmentation. *Compacting algorithms* are one way to solve such problems, while other techniques such as fixed object sizes and pre-fragmentation of large objects can be used to avoid the fragmentation problem [Bak92], [Hen93] and [WJ93]. We have chosen to work with a compacting algorithm since it provides a very clean solution, and solves a more general problem. For an extensive discussion on incremental GC techniques see [Wil92].

3.1 Development for interactive systems

The original batch formulations of the GC algorithms are unsuited for interactive use. The introduction of generation algorithms [LH83,Ung84], based on the copying algorithm changed this picture and solved the problem nicely for so called 'young' garbage. Splitting available memory into smaller areas, garbage collected one at a time, makes a GC cycle usually short enough to go unnoticed for human users. In the worst case there can be a need for GC work in several generations. The inflicted stops are by far too long and using many small generations would not solve the problem due to the bad worst case behavior. The techniques can improve the overall performance and are therefore interesting to combine with techniques invented for dealing with the real-time situations, but they do not alone solve our problem.

3.2 Development for the real-time situation

Although there are some earlier attempts we start with discussing Bakers algorithm [Bak78]. This is an incremental formulation of the copying algorithm. Objects are copied one at the time from an old area ('fromspace') to a new area ('tospace') which is at the same time used for allocating fresh objects. The GC work (the copying) must be scheduled often enough to have all objects evacuated before the new area runs out of space. The proposal is to let allocation of new objects drive the GC work by copying enough live objects at the same time, depending on the size of the allocated object. This scheme simplifies the arguments that objects are indeed copied at a sufficient rate to guarantee that a flip can be made when the tospace is filled. In order to also update all pointers to the new location of moved objects, the original formulation uses a 'read barrier', that is, pointers are updated (using forward pointers in fromspace) when accessed after the object has been moved. Later improvements of the algorithm [Bro84] shows that a 'write barrier' can be used instead.

From real-time point of view this algorithm is a great improvement since GC work is carried out in very small chunks, copying a few objects at the time. This work can be triggered when allocating new objects and when accessing (updating) pointers. The disturbance is small enough for use in low-priority processes, since the possible delay of starting a high-priority process depends on the time to move a small number of objects. For high-priority processes the scheme adds considerable time to the worst-case behavior since these operations (allocation, access/update pointers) can be very frequent [Wit91, WJ93]. Depending on the timing constraints this might be enough to prohibit use of these operations in a high-priority process which can be a rather severe restriction (especially forbidding pointer manipulation). Our approach, making the critical operations very short and of fixed length, has also been used by [Nil94] although there it relies on hardware support.

4 Integrated process and GC scheduling

The main idea in this paper is that the garbage collection work should be scheduled in such a way that it inflicts minimal disturbance to the high-priority processes. To do this, we propose a process scheduler that knows about the real-time garbage collector and helps scheduling its work. The scheduler primarily assigns the processor time to the high-priority processes, dividing the remaining time between the garbage collector and the low-priority processes. Since some garbage collection work must be done in order to satisfy the memory demands of the high-priority processes, the garbage collector is given precedence over the low-priority processes. Thus, the order of which the scheduler assigns the processor time to the different activities should be:

1. High-priority processes
2. Garbage collection
3. Low-priority processes

The garbage collector will suspend its work as soon as it can guarantee that the high-priority processes will not run out of memory. The effect of this scheme is that it will appear to the high-priority processes as if the system was equipped with an infinite amount of memory. They are never interrupted by garbage collection, nor will garbage collection keep them from being activated at the expected time. The overhead caused by pointer assignment and pointer dereferencing is bounded by fixed tight constants (twelve and one machine instruction respectively on a Motorola 68040 microprocessor). The worst-case cost for memory allocation is proportional to the number of pointers it contains, since they need all be initialized to null. To the low-priority processes on the other hand, it will appear as if the system had an incremental real-time garbage collector interrupting the application program for short, bounded, periods.

The rest of this section studies the implications of our strategy when applied to a system supporting only one high-priority process. Furthermore, we assume an incremental copying GC algorithm with a write barrier, such as those presented by Brooks [Bro84] and Bengtsson [Ben90]. It should however be possible to adapt our scheduling strategy to other algorithms as well.

Allocation

Allocation of memory by *low-priority processes* are checked to guarantee that the present ratio between performed GC work and allocated memory doesn't drop to low. If it does, the garbage collector is given precedence. The upper bound on the GC work performed in connection with an allocation will be proportional to the size of the allocated area.

When a *high-priority process* allocates memory no garbage collection is performed, making it an extremely cheap operation. This scheme means that we may temporarily get behind with the GC work, but the garbage collector is activated immediately after suspending the high-priority process to perform the missing GC work.

If a high-priority process is activated shortly before a semi-space flip is due, the remaining memory in tospace could potentially be too small to hold both the objects allocated by the high-priority process and the last objects to evacuate from fromspace. We therefore have to reserve an amount of memory in tospace equal to the worst-case allocation need of the high-priority process during one activation, to be allocated by the high-priority process if necessary.

Pointer assignment

Pointer assignments must be checked to make sure that no pointers to objects in fromspace are introduced into objects already scanned by the garbage collector. If such an attempt is made, the referenced object is evacuated to tospace. Pointer assignments by low-priority processes cause immediate evacuation. In order to avoid this potentially costly operation during the high-priority process, we just reserve space for objects to evacuate. The actual evacuation is delayed until the high-priority process is suspended. Pointer assignment is thus a cheap, fixed-length, operation for the high-priority process, which can not suffer from rapidly accumulating worst-case delays.

For all processes, pointer dereferencing is done indirectly through a forwarding pointer in the head of the object, adding an overhead of one machine instruction.

The approach outlined here is more fully described in [MH95] and in [Hen94]. [MH95] describes GC algorithm and scheduler details and further analyzes the costs of our scheme. It also discusses the information about the system which must be provided by the programmer (maximum amount of memory occupied by live objects, and maximum allocation need during one activation of the high-priority process).

5 Experience and future work

An existing real-time kernel [AB91] was enhanced with our garbage collector and used in a control application. The application runs on a computer based on a 25 MHz Motorola 68040 processor. This prototype of the garbage collector was written entirely in C++, with the interaction between the control application and the garbage collector explicitly coded into the application. Our strategy is really intended for a real-time environment with a GC- and process-aware compiler.

Measurements on the control application show that the latency of the high-priority process is about 0.2 ms. The major part of this time is spent by the real-time process scheduler administrating the context switch. The execution time for one invocation of the high-priority process grew from 0.9 ms to 1.0 ms when garbage collection was introduced. This overhead can be attributed to the implementation of the write barrier which causes a number of procedure calls to be performed at each pointer assignment. As mentioned earlier it can, in a more ambitious implementation, be replaced by twelve in-line machine instructions. The measurements show that the prototype garbage collector already supports sampling frequencies up to about 1 kHz.

Future work

Currently, our strategy and prototype implementation handle only one high-priority process. The thing closest at hand is to generalize our strategy to handle *several concurrent high-priority processes*. We are convinced that this will not be too difficult.

We furthermore assume that the high-priority process is clock-driven. This is true for a large class of applications, but we would also like to support applications where the high-priority processes are triggered by *asynchronous events*. Given some knowledge of the physical process generating the events, we believe that these systems too can be incorporated in our model.

We would also like to improve the situation for the *low-priority processes*. We suggest that the GC work should be performed at a *lower pace early in the GC cycle*, using an estimate of the amount of live objects when calculating the GC speed. This approach would give shorter individual pauses in the average case and also improve overall performance since the number of floating objects would be lower. It would however lead to a slightly worse worst case for the low-priority processes.

Finally, it would be interesting to study what *language and operating system support* is needed to make real-time garbage collection transparent to the programmer and well integrated with process handling. Today's languages and library-based real-time kernels are not entirely suitable environments for garbage collection since too much of the collector internals are exposed to the application program.

6 Conclusions

We have presented a new approach to scheduling garbage collection work which enables use of dynamic memory allocation also in situations with very time critical processes. The solution integrates scheduling of application processes with scheduling of garbage collection work. Garbage collection work has lower priority than high-priority processes, but higher than low-priority processes. Garbage collection work thus never take time from the high-priority processes.The scheme adds very little to worst-case execution times for operations performed by the high-priority processes which is important when doing performance evaluations.

Acknowledgments

This work has been supported by NUTEK, the Swedish National Board for Technical Development in the program for Embedded systems. We also want to thank Klas Nilsson and Anders Blomdell, at the department of Automatic Control, for their active cooperation in this work and in particular for providing the means to evaluate the ideas in a control environment.

References

[AB91] L. Andersson, A. Blomdell. A Real-Time Programming Environment and a Real-Time Kernel. *National Swedish Symposium on Real-Time Systems*, 1991.

[AW84] K. J. Åström, B. Wittenmark. *Computer Controlled Systems - Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.

[Bak78] H. G. Baker. List Processing in Real Time on a Serial Computer. *Communications of the ACM*, April 1978.

[Bak92] H. G. Baker. The Treadmill: Real-Time Garbage Collection Without Motion Sickness. *ACM SIGPLAN Notices*, Vol 27, No 3, March 1992.

[Ben90] M. Bengtsson. *Real-Time Compacting Garbage Collection Algorithms*. Licentiate thesis, Department of Computer Science, Lund University, 1990.

[Bro84] R. A. Brooks. Trading Data Space for Reduced Time and Code Space in Real-Time Garbage Collection on Stock Hardware. *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, August 1984.

[Hen93] W. Hennessey. *Real-Time Garbage Collection in a Multimedia programming Language*. OOPSLA'93 Workshop on Memory Management and Garbage Collection. Washington DC, Oct 93.

[Hen94] R. Henriksson. Scheduling Real Time Garbage Collection. *Proceedings of NWPER'94*, Lund, Sweden, June 1994.

[LH83] H. Lieberman, C. Hewitt. A Real-Time Garbage Collector Based on the Lifetime of Objects. *Communications of the ACM*, June 1983.

[MH95] B. Magnusson, R. Henriksson Garbage Collection for Control Systems, *Proceedings of IWMM'95*, Kinross, Scotland, September, 1995.

[Nil94] K. Nilsen, W. J. Schmidt. A High-Performance Hardware-Assisted Real-Time Garbage Collection System. *Journal of Programming Languages*, Chapman & Hall, 1994(2).

[Ung84] D. Ungar. Generation Scavenging: A Non-disruptive High Performance Storage Reclamation Algorithm. *ACM SIGPLAN Notices*, 19(5), May 1984.

[Wit92] P. T. Withington. How Real is "Real-Time" GC?. OOPSLA'91 Workshop: Garbage Collection in Object Oriented Systems, *OOPS Messenger*, October 1992. Position paper.

[Wil92] P. R. Wilson. Uniprocessor Garbage Collection Techniques. *Proceedings of IWMM'92*, St. Malo, France, Sept 1992, LNCS 637.

[WJ93] P. R. Wilson & M. S. Johnstone. *Real-Time Non-Copying Garbage Collection*. OOPSLA'93 Workshop on Memory Management and Garbage Collection. Washington DC, Oct 93.