Predictable Automatic Memory Management for Embedded Systems

Roger Henriksson

LU-CS-TR:97-189
LUTEDX/(TECS-3077)/1-8/(1997)

Presented at:
OOPSLA'97 Workshop on Garbage Collection and Memory Management,
Atlanta, Georgia, USA, October 5, 1997.

An earlier version of the paper was published in:
Proceedings of SNART'97, Conference of the Swedish National Real-Time Association,
Lund, Sweden, August 21-22, 1997.

Department of Computer Science

Lund Institute of Technology
Lund University

Box 118, S-221 00 Lund, Sweden

Predictable Automatic Memory Management for Embedded Systems

Roger Henriksson

Department of Computer Science, Lund University
Box 118, S-221 00 Lund, Sweden
e-mail: Roger.Henriksson@dna.lth.se

Abstract

The power of dynamic memory management can be used to produce more flexible control applications without compromising the robustness of the applications. It is demonstrated how automatic memory management, or garbage collection (GC), can be used in a system that has to comply with hard real-time demands while still preserving the predictability of the system. A suitable garbage collection algorithm is described together with a strategy for scheduling the work of the algorithm. A method for proving that a given set of processes will always meet their deadlines without interference from the garbage collector is given.

1 Introduction

Traditionally, embedded real-time systems have been implemented using static techniques to ensure predictability. Static task scheduling has been used together with static memory management. The demand for more flexible applications together with improved implementation and analysis techniques have brought an increased use of dynamic process scheduling. However, memory is still in most cases managed statically. Increased application complexity and demands on flexibility make dynamic memory management more and more desirable. Object-oriented techniques have recently begun making their way into the development of embedded systems, which further accentuates the need for dynamic memory management.

The most common technique for dynamic memory management is *manual* memory management. This implies that it is the responsibility of the application, or the application programmer, to decide which objects in memory must be retained (so called *live* object) and which can be deallocated and reused (*dead* objects). Manual memory management does enable a flexible use of the available memory, but it also introduces a set of new problems. In any but toy-size applications, the problem of keeping track of which objects are live and which are dead is very complex. Failure to manage the memory in a correct fashion usually manifests itself in one of two ways, namely memory leaks or dangling pointers. Memory leaks are caused by neglecting to deallocate dead objects. Eventually the application runs out of memory, which cause the application to crash. Dangling pointers are introduced by deallocating objects to early, while they are still live. When the application later tries to access one of the deallocated objects it often results in a crash.

Memory leaks and dangling pointers can be avoided if *automatic* memory management is introduced. Here, it is the responsibility of a special module of the runtime system, the garbage collector, to identify dead objects and reclaim the memory they occupy. This process is called *garbage collection*, or *GC* for short. A lot of error-prone code can thus be removed from the application. Another bonus that GC brings is that it can easily be combined with memory compaction, avoiding memory fragmentation that otherwise is a problem associated with dynamic memory management.

Automatic memory management has a lot of attractive properties that make it desirable in embedded systems. 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 existing GC techniques tend to disrupt the application for too long periods of time.

This paper presents a novel strategy for scheduling the GC work in such a way that the interference with the execution of critical processes is minimized. The worst-case GC overhead for high-priority processes is predictable, strictly bounded, and small in respect to the response time demands of most control systems. This makes it possible to a priori guarantee that the application will meet all its deadlines. A variant of the strategy handling only a single critical process has previously been described in the form of a licentiate thesis [Hen96].

2 Control systems

In control systems, *high-priority processes* perform tasks such as sampling input from the controlled process, executing regulator algorithms (PID algorithms etc.), and adjusting output signals. These processes are executed periodically, often as frequent as 100 times or more 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 period of time, often in the order of 1 millisecond or less. Control theory needs the time-constraints in order to guarantee stability [ÅW84]. To summarize:

- 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 within a time frame that is shorter than the demanded activation time.
- High-priority processes must complete in time. When calculating the worst-case execution time, possible delays add up. It is therefore an advantage if the individual worst-case costs for memory management operations can be kept small enough that the cumulative cost does not add significantly to the worst-case execution time.

Low-priority processes are used to perform actions such as computation of reference values, supporting presentation and supervision of the state of the controlled process, changing parameters etc. These processes are also time critical, but the time constraints are much weaker, typically in the area of 100 milliseconds or less. Missing a deadline can often be tolerated providing it does not happen too often.

3 A real-time GC algorithm

Baker's algorithm [Bak78] is an incremental copying garbage collection algorithm. By incremental we mean that the garbage collector (*collector* for short) runs interleaved with the application program (also denoted *mutator* since it modifies the object graph). Each time the collector is invoked, a small piece of GC work is performed. The heap is divided into two equally sized semispaces, *fromspace* and *tospace*. Live objects are copied, or *evacuated*, one at the time from an old area (*fromspace*) to a new area (*tospace*), which is also used for allocating new objects, see Figure 1. When tospace runs out of space, all live objects must have been evacuated from fromspace, so it can be reused. At this moment, a *flip* is performed, reversing the role of tospace and fromspace and starting a new GC cycle. The GC work (the copying) must be scheduled often enough to guarantee that all live objects are evacuated before tospace is filled (but from an efficiency point of view preferably just in time). Baker's proposal is to let allocation of new objects trigger the GC work. When a new object is allocated, an amount of live objects relative to the size of the new object is evacuated. This scheme simplifies the arguments that objects are indeed copied at a sufficient rate to guarantee that a flip can be made when tospace is filled. In order to update all pointers to the new location of a moved object, the original formulation uses a *read barrier*, that is, pointers are updated (using forwarding pointers in fromspace) when accessed after the object has been moved. A later improvement of the algorithm [Bro84] shows that a *write barrier* can be used instead, only intercepting pointer updates.

Fine-grained incremental algorithms, such as Baker's algorithm and its variants, constitute a large step towards making garbage collection work in real-time systems. Each memory management operation can potentially trigger GC work, but the amount of work performed at each invocation is small and bounded. The disturbance is small enough for use in low-priority processes due to their relaxed real-time demands and since the possible delay of starting a high-priority process is only one short GC invocation. For high-priority processes on the other hand, the overhead can be too large. The worst-case execution time for a sequence of memory management operations quickly adds up. Some improvements of the algorithm are thus needed in order to make it useful in a system with hard real-time demands.

4 Scheduling principle

The fact that every operation related to memory management may trigger GC work to be performed makes it difficult to guarantee short enough response times for the high-priority processes. We therefore propose that these processes

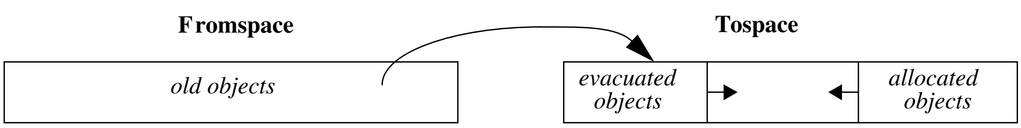

Figure 1 *The heap structure of Baker's algorithm.*

are treated in a different way from a memory management point of view. GC work should be prohibited altogether while a high-priority process executes. This in turn means that the garbage collector will, temporarily, get behind with its work. The missing GC work is performed by a separate GC process as soon as no more high-priority processes are eligible for execution. Low-priority processes, on the other hand, use the traditional strategy to trigger GC work. It should be noted, however, that lengthy (from the perspective of a high-priority process) GC work triggered by a low-priority process must be interruptible in order not to delay an invocation of a high-priority process. We will thus have three main levels of priority:

1. High-priority processes
2. Garbage collection triggered by the high-priority processes
3. Low-priority processes interleaved with triggered GC work

Processes within priority levels 1 and 3 above may of course have different priorities amongst themselves.

The proposed scheduling strategy is quite general in the sense that it is applicable to most fine-grained incremental GC algorithms. We will, however, limit ourselves to studying what implications the strategy has on one such algorithm, namely Brooks' algorithm as formulated by Bengtsson [Ben90].

High-priority processes

Time-critical high-priority processes must be able to guarantee short response times. Therefore, memory management operations performed by such processes must be kept cheap. There are three kinds of memory management operations: pointer dereferencing, pointer assignment, and allocation.

If Brooks' algorithm is used, the cost of dereferencing a pointer is already low. Pointer dereferencing is performed using an indirection step (by following a *forwarding pointer* in the head of the objects). This allows objects to be moved without immediately identifying and changing every pointer in the system referring to the object. The overhead for dereferencing a pointer consists thus only of one extra memory access.

In order to guarantee that an assignment to a pointer does not create pointers into fromspace that the garbage collector does not know about, Brooks proposes that a *write barrier* is used. All pointer assignments are watched and those that might jeopardize the integrity of the heap are caught. If the object referenced by the new value of the pointer is located in fromspace it is immediately evacuated and the pointer is updated. Thus, in the worst-case, every pointer assignment would cause an object to be copied. The overhead will furthermore depend on the size of the evacuated objects. The worst-case execution time for pointer assignments adds up quickly, making it difficult to guarantee short response times. Therefore, we employ a *delayed-evacuation* strategy in which we delay the actual evacuation of objects until the high-priority processes have finished executing [Hen96]. The worst-case overhead for a pointer assignment can now be reduced to as little as 12 machine instructions on a processor such as a Motorola 680x0, independent of the object size.

Memory allocation can be a very costly operation in the original version of the algorithm. Whenever an allocation request is made, the garbage collector is started, performing an amount of work. The amount of GC work, and thus the required time, depends on the size of the requested block of memory, the maximum amount of simultaneously live memory, the total heap size and the maximum amount of GC work that may have to be performed during one GC cycle. The overhead for a memory allocation will consequently vary if the size of the heap changes or if the maximum amount of live memory changes (perhaps due to changes in unrelated parts of the program). In order to eliminate the high cost for memory allocation in high-priority processes, we delay the GC work until all high-priority processes have finished executing. A memory allocation will now become a cheap operation for a high-priority process, only involving modifying the allocation pointer and initializing the contents of the new object.

Reserved memory

The copying garbage collector algorithm we use to illustrate our scheduling strategy may experience a deadlock if the evacuation of live objects from fromspace does not keep up with the allocation of new objects. The garbage collector might end up in a situation where some objects remain to be evacuated but there is no space left in tospace to hold them. When low-priority processes allocate memory, they make sure that enough GC work has been performed before they actually allocate the new object, a strategy which guarantees that no such deadlock occurs. High-priority processes, on the other hand, allocate memory before the corresponding GC work has been performed. This is a potentially dangerous situation: If the high-priority processes are invoked shortly before a flip is due, there might not be enough memory left in tospace to hold both the new objects and the live objects that have not yet been evacuated from fromspace. The solution to this problem is to schedule the GC work, and the flip, in such a way that enough memory remains in tospace for evacuation of live objects even if high-priority processes are invoked immediately before the flip. This can viewed as reserving an amount of memory in tospace for allocation made by high-priority processes. We denote this amount M_{HP}.

5 Scheduling analysis

Before a safety-critical control program is used in a real control situation, it is important to convince oneself that the program will meet all its hard deadlines, i.e. the process set is schedulable. Information about the process set, such as worst-case execution times for the various processes, worst-case GC work that has to be performed to clean up after the processes, and worst-case allocation need for each process is used as input to the analysis.

Verifying the schedulability of the GC work is a two-stage process. First, the high-priority processes are studied separately and it is determined whether they are schedulable or not. If not, the system is clearly not schedulable and we are finished. Otherwise, we continue by analyzing the GC work motivated by the actions of the high-priority processes. The findings from the analysis is also used to determine the amount of memory that must be reserved in tospace for the high-priority processes.

High-priority processes

The first part of the analysis, determining the schedulability of the high-priority processes employs standard scheduling analysis techniques. For example, if rate monotonic scheduling [LL73] has been used to assign priorities to the high-priority processes, rate monotonic analysis can be used. This includes the scheduling test of Lui and Layland [LL73], which uses the processor utilization ratio, or the exact analysis originally presented by Joseph and Pandya [JP86] and later enhanced by others to include processes with deadlines shorter than the period of the process, blocking, release jitter etc. [SRL94].

GC interference with a high-priority process manifests itself in two ways: a slightly increased worst-case execution time for the high-priority processes and a slight release jitter in the invocation of the processes. Both types of interference can easily be handled by existing analysis theory.

Garbage collection work

Given that the high-priority processes of a system have been determined to be schedulable, we need to verify that the GC work motivated by the actions of the high-priority processes is schedulable as well. Consider the worst-case scheduling situation, in which all high-priority processes are released simultaneously. Each time a high-priority process, τ_i, with period T_i, is invoked, it executes for a duration equal to its worst-case execution need, C_i, and performs memory management related actions that requires a worst-case garbage collection work of G_i to be performed. If we can show that this situation is schedulable, all other possible (less demanding) scheduling situations will also be schedulable.

Definition. We define the worst-case response time of the garbage collector, R_{GC}, as the time from the high-priority processes are simultaneously released until no more garbage collection work is left to be performed.

Let C_{GC} denote the worst-case time required for GC work in any interval of time of length R_{GC}, $t..t{+}R_{GC}$. R_{GC} can then be calculated in a similar way to how response times are calculated in the exact rate-monotonic analysis. We assume that N high-priority processes, $\tau_1..\tau_N$, exist: [1]

$$R_{GC} = C_{GC} + \sum_{i=1}^{N}\left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot C_i\right) \qquad (1)$$

The equation contains C_{GC}, which in our case depends on the actions of the high-priority processes. For each invocation of a high-priority process τ_i during R_{GC}, the required garbage collection work amounts to G_i. The total garbage collection work during R_{GC} will therefore be:

$$C_{GC} = \sum_{i=1}^{N}\left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot G_i\right) \qquad (2)$$

Applying (2) to (1) yields:

$$R_{GC} = \sum_{i=1}^{N}\left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot (C_i + G_i)\right) \qquad (3)$$

R_{GC} is found on both the left side and the right side of the equality. The smallest non-zero value of R_{GC} that satisfies (3) can be found using the recursive formula:

$$\left.\begin{aligned} R_{GC}^{0} &= \sum_{i=1}^{N} C_i \\ R_{GC}^{n+1} &= \sum_{i=1}^{N}\left(\left\lceil \frac{R_{GC}^{n}}{T_i} \right\rceil \cdot (C_i + G_i)\right) \end{aligned}\right\} \qquad (4)$$

It should be noted that we cannot use 0 as the first approximation of R_{GC}, as is usually done in

1. We use $\lceil\ \rceil$ to denote the ceiling function, i.e the smallest integer such that it is equal to, or larger than, the function argument.

rate-monotonic analysis when calculating the worst-case response time of a process. This is because 0 is a trivial solution to (3), whereas the solution we want is the first positive, non-zero solution. Clearly, R_{GC} cannot be smaller than the sum of the worst-case execution times for the high-priority processes, since all processes are released simultaneously in the worst case and the garbage collector has lower priority than these processes. Therefore, the sum of the worst-case execution times for the high-priority processes is a suitable initial approximation of R_{GC}.

If the garbage collection work is schedulable, (4) will converge. If the garbage collection work is *not* schedulable, (4) will *not* converge since no solution exists. It is easy to detect that (4) has converged. This happens when two consecutive values of R_{GC} are found to be equal. But how do we detect that the formula does *not* converge? The answer is that it is possible to calculate an upper bound for R_{GC}. If one of the steps in the iterative process of calculating R_{GC} yields a value larger than the maximum possible response time, we can deduce that the iteration will not converge.

Theorem. The maximum possible response time for the garbage collector is the *least common multiple of the periods of the high-priority processes*, denoted $lcm(T_1..T_N)$.

If we, for example, have a system with two high-priority processes with periods of 10 and 14 milliseconds respectively, the response time of the garbage collector must be less than or equal to $lcm(10,14) = 70$ milliseconds for the system to be schedulable.

Proof. Assume that all the high-priority processes are released simultaneously at time t. This is the worst-case scheduling situation. The processes will execute with different periods forming a scheduling pattern. Sooner or later they will all again become ready to run simultaneously, after which the scheduling pattern will repeat itself. This happens at time $t+lcm(T_1..T_N)$. Thus, if there was not enough time in the time slot $t..t+lcm(T_1..T_N)$ to complete the garbage collection work in progress, there will not be enough time in the next time slot either, and so on. The amount of required garbage collection work will continue to accumulate. The response time of the garbage collector must therefore be less than, or equal to, $lcm(T_1..T_N)$.

Reserved memory

How do we calculate the amount of memory that has to be reserved in tospace for high-priority allocation, M_{HP}? We can do this by assuming that all high-priority processes are released immediately before a flip is to be performed. We furthermore assume that the flip can not be performed within R_{GC} time units after the invocation of the high-priority processes, i.e. not until the garbage collector have finished the current GC cycle. This is the worst possible situation.

Our assumptions lead to the conclusion that we must reserve enough memory in tospace to hold all objects allocated during R_{GC} time units. This amounts to the sum of the worst-case allocation needs for the high-priority processes during this time. The allocation need for a process can be calculated by multiplying the worst-case allocation need for one invocation, A_i, with the number of times the process might be invoked during a time span of R_{GC}. We get:

$$M_{HP} = \sum_{i=1}^{N} \left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot A_i \qquad (5)$$

Priority inheritance schemes

Priority inversion is a phenomenon where a higher priority process is blocked by a lower priority process holding a semaphore that the higher priority process is attempting to lock. The duration of the blocking is usually short since the semaphore is typically used to protect a short critical region. However, if a third, medium priority, process is released, it will preempt the lower priority process and prevent it from releasing the semaphore. This lead to arbitrary delays of the higher priority process, which is not acceptable in hard real-time systems. The problem of priority inversion was first described by Lampson and Redell [LR80].

To avoid blocking caused by priority inversion, priority inheritance protocols are often employed. All of these protocols involve temporarily raising the priority of a process that has locked a semaphore, which may cause a low-priority process to become a high-priority one until the semaphore is unlocked. This must be taken into consideration when analyzing the schedulability of a system of processes. We will look at how the probably most used priority inheritance protocol, namely the *basic inheritance protocol*, can be incorporated in our scheduling analysis. Two other priority inheritance protocols are the *priority ceiling protocol* and the *immediate inheritance protocol*, both of which can be included in the analysis in similar ways.

The basic inheritance protocol states that whenever a process blocks because the semaphore it attempts to lock is already locked by a process with a lower priority, the process currently possessing the lock will inherit the priority of the blocked process. The priority of a process is thus raised if, and only if, it is blocking a higher priority process.

Blocking a high-priority process and raising the priority of the process holding the semaphore to the priority level of the blocked process can clearly be viewed as being equivalent to the high-priority process performing the work within the critical region of the process with lower priority. We can thus incorporate the basic inheritance protocol in our scheduling analysis by modifying (3) slightly:

$$R_{GC} = \sum_{i=1}^{N} \left(\left\lceil \frac{R_{GC}}{T_i} \right\rceil \cdot (C_i + d_i + G_i + g_i) \right) \quad (6)$$

For each process, τ_i, we have to add the worst-case time it, as we see it, performs work on the behalf of low-priority processes to the worst-case execution time of the process. We denote the additional time d_i. While performing the work of a low-priority process, τ_i might take actions that motivate additional GC work to be performed. The additional worst-case time for GC work is denoted g_i, and must also be taken into account when calculating the response time of the garbage collector. It is worth noticing that it is only low-priority processes that influence d_i and g_i. The execution time and GC need of high-priority processes are already taken into account, even if they do block other high-priority processes.

To analyze a system utilizing the basic inheritance protocol, we must be able to determine the value of d_i and g_i. Any of the following two observations can be used to find an upper bound [SRL90]: First, under the basic inheritance protocol, a process τ_i can be delayed at most once by each process with lower priority which share semaphore with τ_i. Second, if m semaphores exist which can cause τ_i to block, then τ_i can be blocked at most m times, i.e. once by each semaphore. By analyzing the worst-case execution times and allocation need of the corresponding critical regions and adding them up, we can compute d_i and g_i.

6 Implementation

A garbage collector based on Brooks' algorithm, as formulated by Bengtsson [Ben90], and scheduled according to the principles described in this paper has been implemented within an actual real-time kernel [AB91]. The kernel has been developed at the Department of Automatic Control, Lund Institute of Technology, and it is used in various hard real-time applications, including control of industrial robots. The garbage collector was implemented in C with some critical parts in assembly code.

A series of test programs were run on a VME control computer equipped with a 25 MHz Motorola 68040 processor. The test programs, the garbage collector, and the real-time kernel were augmented with code producing output on the digital I/O port of the VME computer. The performance of the garbage collector could then be monitored by connecting a logic analyzer to the I/O port. Several performance aspects were studied, such as the cost of individual memory management operations, the amount of work required for garbage collection, and the time a high-priority process can be delayed by garbage collection.

Pointer assignment

Each pointer assignment is guarded by a write barrier in order to catch assignments that might jeopardize the integrity of the heap. The worst-case time required for a pointer assignment was determined to be 10 µs for high-priority processes. The worst-case cost is independent of object size since we do not evacuate objects while high-priority processes are running. For low-priority processes, on the other hand, the worst-case time depends on object size. For example, a pointer assignment involving a pointer to a 36 byte object required 21 µs in the worst case. Larger objects result in higher worst-case delays.

Allocation

No garbage collection is performed in association with memory allocation requests made by high-priority processes, which makes them significantly cheaper than in the original version of Brooks' algorithm. Allocation involves only updating an allocation pointer and initializing the contents of the new object. All memory cells constituting the object is set to zero except for the GC information fields in the object header. This approach has been chosen for reasons of simplicity and safety. A more efficient approach, from a strict memory management point of view, would be to just set the pointers within the new object to zero, not the contents of the entire object. However, initializing scalar values as well as pointers is preferable in safety-critical applications since it reduces the risk of bugs related to the programmer forgetting to initialize scalar fields. The total cost for an allocation request consists of a constant time for administrating the request and a initialization time proportional to the size of the object to allocate. The following table presents some worst-case allocation times for a high-priority process.

object size (bytes)	*worst-case delay (µs)*
100	22
500	37
1000	58

When a low-priority process requests memory, an amount of GC work corresponding to the size of the requested object is performed. The worst-case time delay will in this case therefore be significantly higher than for a high-priority process.

Garbage collection work
Whenever a high-priority process is suspended and no other high-priority process is ready to execute, the garbage collector checks if there is any GC work pending. If so, garbage collection is started. The amount of GC work that is performed depends on the amount of memory allocated by the high-priority processes, the heap size, the maximum amount of simultaneously live memory, and the maximum amount of total GC work that may be required during one GC cycle. The worst-case time required for garbage collection in the presence of a single high-priority process was studied. The high-priority process ran with a frequency of 1000 Hz and required 222 µs to execute in the worst case. The clock interrupt that triggered the process required 85 µs. Each semispace consisted of 50 kilobytes (total heap size 100 kilobytes) and the object size and amount of simultaneously live memory was varied. The worst-case times for garbage collection are given in the following table.

object size (bytes)	*simultaneously live memory (bytes)*	*worst-case time required for GC (µs)*
100	4000	221
100	20000	261
500	20000	470
1000	22000	503

We see that the garbage collector was able to keep up with the application even under heavy load (1 megabyte allocated memory per second in a 100 kilobyte heap with a high ratio of live objects).

It should also be noted that the garbage collector is suspended if a high-priority process becomes ready to run while GC work is performed. High-priority processes are thus not blocked for the time periods given in the table above.

Locking
Manipulating the pointer graph and copying objects cause the GC heap to be temporarily inconsistent. Such operations must therefore be performed atomically. This is achieved by switching off the processor interrupts before an atomic operation is commenced and switching them on again directly after completing the operation. No context switch is possible during this time. A high-priority process becoming ready to run might therefore be delayed until the atomic operation is finished. It is therefore important to keep atomic operations shorter than the maximum latency tolerated by the critical processes.

The first implementation of our garbage collector considered copying an object to be a single atomic operation. Initialization of an object in connection with an allocation request was considered an atomic operation as well. This proved to work well as long as only small objects, less than 100 bytes, resided on the heap, but it did not scale up as the maximum object size grew. For example, a 1000 byte object caused a 177 µs locking delays, which is rather long for systems with high-frequency control loops. For even larger object sizes, which is quite realistic in an embedded system, the problem grows even worse.

The increase of the worst-case locking time with growing object size can be avoided if object copying and object initialization are made interruptible. Then, the worst-case delay will be independent of the maximum object size. Object initialization can easily be made interruptible, but object copying presents some practical problems with process synchronization. If a context switch occurs when the garbage collector is copying an object, the process given control of the processor might attempt to modify the object being copied. In such a case it will modify the old copy, perhaps a part of the old copy that has already been copied. In order to avoid producing an inconsistent version of the object when the copying is resumed, we have chosen to back out of the copying and restart it from scratch. This does lead to a slightly increased worst-case overhead for GC, since every context switch could potentially cause an object copying to be aborted. By making object initialization and copying interruptible we have managed to limit the worst-case locking delay to 38 µs, independent of the maximum object size.

7 Conclusions

We have discussed embedded systems and how they can benefit from dynamic memory management. The drawbacks of manual memory management have been pointed out and we have established that automatic memory management, or garbage collection, is desirable in complex embedded systems. However, previous techniques for garbage collection do not provide sufficient support for hard real-time systems.

A novel strategy for scheduling the work of existing fine-granular incremental garbage collection algorithms was presented. High-priority processes which have to comply with hard real-time demands execute with very little interference from the garbage collector. The interference is small in respect to the response time demands of the high-priority processes and is negligible from a control theory point of view. The remaining processor time is divided between running the garbage collector and low-priority processes. The impact of garbage collection on the low-priority processes is small enough to satisfy soft real-time demands.

A method for analyzing the worst-case behavior of a system of processes and garbage collection was presented. The interference from the garbage collector is strictly bounded and predictable. Overhead for memory management operations, such as pointer assignments and allocation requests is analyzed by including it in the worst-case execution time of the high-priority processes. Process release jitter caused by the garbage collector momentarily switching off the interrupts is also easily handled with standard schedulability analysis techniques. It is thus possible to a priori verify that a set of high-priority processes and the corresponding garbage collection work are schedulable without violating any hard deadlines. It was also demonstrated how the analysis works in the presence of priority inheritance protocols.

The described garbage collection strategy was implemented in an existing real-time kernel and used in an actual control application. Measurements demonstrate that automatic memory management is feasible in hard real-time systems, even in systems with sampling frequencies as high as 1000 Hz. The cost of individual memory management operations was observed to be low for high-priority processes.

Acknowledgments

There are several people that have contributed to the work presented in this paper and which I would like to thank: Boris Magnusson for introducing me to the problem of scheduling garbage collection work and for his support throughout the work. Klas Nilsson and Anders Blomdell for valuable hardware and software support during the implementation of the garbage collector. Anders Ive for helping me to evaluate the performance of the garbage collector. In addition, Klas and Anders provided me with many valuable comments on the draft of this paper. Thanks to the Department of Automatic Control, Lund Institute of Technology for providing the means to evaluate the ideas in an actual control environment.

This work has been supported by NUTEK, the Swedish National Board for Technical Development within their Embedded systems program.

References

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

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

[Ben90] M. Bengtsson. *Real-Time Garbage Collection*. Licentiate Thesis, Dept. 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.

[Hen96] R. Henriksson. *Scheduling Real-Time Garbage Collection*. Licentiate Thesis, Dept. of Computer Science, Lund University, 1996.

[JP86] M. Joseph, P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, Vol. 29, No. 5, 1986.

[LL73] C. L. Liu, J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, Vol. 20, No. 1, 1973.

[LR80] B. W. Lampson, D. D. Redell. Experience with Processes and Monitors in Mesa. *Communications of the ACM*, Vol. 23, No. 2, 1980.

[SRL90] L. Sha, R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*. 39(9), 1990.

[SRL94] L. Sha, R. Rajkumar, J. P. Lehoczky. Generalized Rate-Monotonic Scheduling Theory: A Framework for Developing Real-Time Systems. *Proceedings of the IEEE*, Vol. 82, No. 1, 1994.

[ÅW84] K. J. Åström, B. Wittenmark. *Computer Controlled Systems - Theory and Design*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.