Garbage Collection for Control Systems

Boris Magnusson and Roger Henriksson

LU-CS-TR:95-154

Also published in proceedings of IWMM'95,
International Workshop on Memory Management,
Kinross, Scotland, September 1995.

Department of Computer Science
Lund University

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

Garbage Collection for Control 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 systems with complex behavior, 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.

The first GC algorithms were designed with a 'stop the world' scheduling policy: The application was executed until a memory allocation request could not be met, then a complete GC cycle was performed after which the allocation could (hopefully) be honored and the application processing continued. In a batch processing environment this was a useful method, but as *interactive* processing came in fashion, and the size of the managed memory grew, the resulting GC pauses (as seen by an interactive user, often seconds, minutes or more) were unsatisfactory. Much development has focused on designing GC schemes for this interactive situation and systems where the GC pauses are typically in the 100 ms range. Such pauses are comparable with other disruptions inflicted by virtual memory or time sharing with other processes on the same workstation etc. and are hardly noticeable for an interactive human user.

In a hard *real-time* environment the response time demands are, however, often 1 ms or smaller. The techniques developed for interactive systems are not sufficient for dealing with such demands — they can not be scaled a factor of 100 to 1000. Furthermore, they are designed to perform well in average cases, but can not guarantee a longest response time. The problem we face is twofold: First, the GC work must be performed in chunks, small enough to be completed quicker than the demanded response time. Second, the GC work must be performed often enough not to stop the application processing due to lack of free memory. At the same time, the GC work must have an overall efficiency that is reasonable.

In this paper we will focus on the scheduling problem, using existing algorithms for incremental GC [Bak78,Bro84,Ben90]. We will focus on the situation in real-time control systems where scheduling is clock-driven rather than event-driven as in interactive systems. Furthermore, there are often many processes with a wide variety of response time demands, but only a few high-priority processes with hard deadlines. We will use these observations to schedule the GC work during the processing of the 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 of the high-priority processes, makes it possible for us to schedule the GC work in order to guarantee the high-priority processes to perform memory allocation operations without any interference from the GC.

The presented work is novel since it not only guarantees 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.

Section 2 provides background on the control systems and the special requirements they put on memory management. In section 3 we analyze existing approaches to improve the performance of garbage collectors and identify what is missing in order to solve the problem for real-time applications. In section 4 and 5 we describe our new approach in detail. In section 6 we elaborate on costs and overheads and in section 7 we relate the practical experience of using our scheduling technique. An outline of our continued work in this area, including a discussion on how to generalize the result to event-driven processing, precedes the conclusions.

2 Control systems

This paper focus mainly on techniques to enable use of GC in control systems. These are computers built into technical equipment, such as airplanes and cars, or used to control industrial plants. The software of these systems is frequently organized as processes executed periodically (rather than event-driven). It is also common that only a fraction of the processing demands short response times (the high-priority processes) while other parts have much lower demands.

The *high-priority processes* are typically doing 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 both on *sample delay* – they should start with very little delay, and on *control delay* – they should be completed within a guaranteed short time period. These demands are often in the order of 1 ms or less. Meeting such deadlines, in particular for the control delay, is essential. The underlying control theory assumes this to ensure stability, high performance, and safety of the resulting control system [AW84]. High-priority processes are written with great care to minimize the control delay by doing as little processing as possible, leaving extensive calculations to lower priority processes. The designer often makes worst-case estimates on processing times in order to make sure that an implementation will complete in the given time frame in all circumstances.

The *low-priority processes* typically performs tasks such as computation of reference values, supporting presentation and supervision of the state of the controlled process, changing parameters, etc. This often involves communication with other computers (or even direct interaction with a human user) to make animations, collect and verify input etc. These processes are thus also time critical, but the time constraints are much weaker, typically in the area of 100 ms rather than 1 ms. The consequences of a missed deadline are usually also much less severe, a bump during an animation or temporarily displaying an outdated figure.

The main requirement to fulfill in order to make automatic garbage collection for control systems viable is to guarantee that high-priority processes start and finish in time. This demand has the following implications:

- 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 vital that the individual worst-case figures for memory management operations can be kept small.

A large difference between average and worst-case behavior means that performance estimates get very pessimistic (costing more powerful hardware). It is therefore a goal that worst-case times for memory management operations are kept as small as possible, but also to be of fixed length. Dependencies on for example object size is problematic since it makes the worst-case figures large.

In order to get the speed required, developers might in practice, when writing the most demanding processes, be willing to refrain from using expensive operations, such as object allocation. This will be so also in a system where GC is available, but it is our goal to make such considerations small and rare. It is a reasonable assumption that the amount of memory allocation performed in high-priority processes is fairly small. This is a consequence of the overall goal to make these processes as small and fast as possible.

Classification of a particular process into low or high priority depends on the scheduling guarantees that can be given to low-priority processes. It is thus a secondary goal to keep the disturbance small also for low-priority processes in order to limit the number of high-priority processes.

The efficiency of the garbage collector, and thus the time spent doing memory management, is a general problem and not particular to real-time applications — other than that efficiency is often a general concern in these applications. However, increasing the efficiency of the GC scheme used can not alone solve the problem of making automatic garbage collection viable in hard real-time applications.

In summary we conclude that, in order to be useful for control systems, a GC must not delay the start of a high-priority process for more than a comparably short time, typically less than a millisecond. Furthermore, the GC must not add to the worst-case behavior of a high-priority process (potentially delay its completion) by more than a similar amount. GC work should thus be very limited during the execution of high-priority processes.

3 GC for real-time systems

Of the many different versions of GC algorithms, we have chosen to present three original and very different ones, and use them as starting point for understanding the further development. We will here limit the discussion to *compacting algorithms*. Long lived applications such as control systems demand techniques to avoid memory fragmentation. Using compacting algorithms is one way to solve such problems, while other techniques such as fixed object sizes and pre-fragmentation of large objects also can be used to avoid the fragmentation problem [Bak92], [Hen93] and [WJ93]. We have chosen to work with a compacting algorithm since it solves a more general and harder problem and also provides a very clean solution to the fragmentation problem. We assume the reader to have a fair understanding of well known GC algorithms and point to the references for details. In this section we will review existing algorithms for their usability in real-time systems. For an extensive discussion on incremental GC techniques see [Wil92].

3.1 The original algorithms

Mark-sweep [McC60,Tho76]. In this formulation the algorithm modifies the data-structures of the application during its work. It must thus run a complete cycle without interference from the application. The GC work is scheduled in the ‘last minute’, i.e. when an allocation request cannot be serviced. A GC cycle typically takes a very long time (depending on total memory available) and can occur at any allocation request. In a real-time system this algorithm would clearly make it impossible to use allocation in a high-priority process since it would potentially trigger a GC. But even worse, memory allocation cannot be used in the low-priority processes either because this may delay the start of a high-priority process for too long.

The copying algorithm [Min63,Fen69], in its original formulation, shares the characteristics with Mark-sweep of having to complete a full cycle and being scheduled in the ‘last minute’. It is thus not useful for our purposes either.

Reference counting [Col60,Deu76] might appear interesting since it is incremental in its original formulation. When an object becomes unaccessible a short burst of GC work is performed. Unfortunately this work, that might occur at any pointer assignment, can in the worst case become extensive since it might generate secondary inaccessible object (e.g. after removing the last pointer to a tree). Even if this problem is solved in an incremental formulation, it can not remove cyclic garbage, which is unacceptable in connection with long lived applications.

3.2 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 short enough to go unnoticed for human users. GC over old objects, typically kept in a larger area, now and then produce longer stops (in an interactive environment this would happen a few times a day, 'during lunch and coffee breaks'). 'Tenuring' policies control when objects are moved to older areas (generations)[UJ88]. The GC scheduling is done in the 'last minute' and tenuring can generate GC cycles in the next older generation and so on. In the worst case there is a snowball effect of GC cycles generated in all generations.

A recent improvement, the 'train algorithm' [HM92,SG95] has presented a technique to organize also old garbage in smaller areas, collected one at the time, limiting GC stops to typically 100 ms. Splitting up memory in a large number of very small areas would potentially be a technique to bring down the time for a GC stop. However, GC in one area includes analyzing and possibly updating pointers from other areas. The involved administration of cross area pointers increase rapidly with the number of areas, and going to 100 times smaller areas is not feasible.

Although useful for interactive systems these algorithms are not sufficient for real-time systems. The inflicted stops are by far too long, and the worst-case situation can not be improved. By the same arguments as above they can not be used in either high-priority or low-priority processes. We should, however, notice that the techniques invented here (such as generations, large objects area) have had a significant improvement on the efficiency of memory management. The techniques are therefore interesting to combine with techniques invented for dealing with the real-time situations, but they do not alone solve our problem.

3.3 Development for the real-time situation

Although there are earlier attempts to construct garbage collectors for real-time applications, mainly focusing on parallel algorithm aspects [Dij78], 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 (but from efficiency point of view preferably 'just in time'). 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 update also 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

Collector

Wait until tospace is filled, then flip memory spaces.
Evacuate the objects referenced by the roots.
S:= The address of the first pointer in the evacuated objects
while not all evacuated objects are traversed
    case: the pointer referenced by S points to:
    1.- an object in fromspace, not yet evacuated.
        Evacuate the object and update the pointer at S.
        Let the forwarding pointers of both the fromspace and tospace copies of the object reference the (new) tospace copy.
    2.- an object in fromspace, already evacuated.
        Update the pointer at S using the forwarding pointer.
    3.- an object in tospace.
        No action.
    S := The address of the next pointer to be examined in the evacuated objects.
end-while
Repeat from start.

Mutator - Pointer assignment:

If the referenced object is not evacuated then evacuate it as done in collector case 1. Use the updated pointer for assignment.

Mutator - Pointer dereference:

Use indirection via the forwarding pointer.

Mutator - Allocation

New objects are allocated in tospace (from one end, while evacuated objects are placed in the other end).

Figure 1 *Bakers incremental garbage collector algorithmwith modifications from[Bro84] and [Ben90].*

algorithm [Bro84] shows that a ‘write barrier’ can be used instead, only intercepting pointer updating. See figure 1 for an outline of the algorithm.

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 is depending on the time to move a small number of objects. For high-priority processes the scheme adds considerable time to the worst-case behavior of these operations (allocation, access/update pointers). 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 not allowing pointer manipulation).

Later work has shown that also the mark-sweep algorithm can be formulated as an incremental algorithm, also using forwarding pointers and a write barrier [Wad76,Ben90]. Comparing these two algorithms from efficiency point of view gives that for young generations with a low ratio of live objects, the copying algorithm is more efficient, while in situations with a high ratio of live objects, the mark-sweep version is more efficient [Ben90]. From real-time aspects the incremental mark-sweep algorithm has the same characteristics of being useful for low-priority processes, but problematic for high-priority processes.

An interesting approach has been taken in the Erlang language implementation [ADVW92] where a large number of very small areas are used (100s of bytes). Each area can be collected quickly enough not to disturb medium-hard real-time applications. The language here guarantees that no pointers can cross from one area to another which is the key to why this approach can be used. This is thus an exceptional case, showing the influence of language mechanisms, but unfortunately this technique is not of general use.

A workshop position paper [Wit92] outlines an approach that builds on the idea of giving special treatment of high-priority processes, but it leaves some situations where unbound delays can occur unsolved. The approach is to modify an existing system and much of the discussion is concerned with the interaction with specialized hardware. To the best of our knowledge this work was not completed. Some of the ideas we present in this paper, in particular special treatment of high-priority processes, are similar. We do, however, also solve the remaining problems, and our approach is more generally applicable since it make no assumption on hardware support. We also present some results from a completed implementation.

4 Our solution: integrating 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 incurs minimal disturbance of 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. The remaining time should then be divided 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 to the low-priority processes. We thus get the order of which the scheduler assigns the processor time to the different activities:

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

In order not to have the low-priority processes suffer from starvation, the garbage collector will suspend its work as soon as it can guarantee that the high-priority processes will not run out of memory. Furthermore, a natural optimization is to let the garbage collector work if no other processes want to run, making use of otherwise idle processor time. 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. 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.

For the rest of the paper we make a couple of simplifying assumptions. We assume that the real-time application only has one high-priority process. We do not, however, think that there will be any major obstacles to generalize our strategy to handle several high-priority processes. Furthermore, we use an incremental copying algorithm with a write barrier, such as those presented by Brooks [Bro84] and Bengtsson [Ben90], as the base for our garbage collector.

Scheduling

To decide whether the garbage collector should work or suspend itself in favor of the low-priority processes, we statically calculate a *minimum GC ratio*. We denote the amount of garbage collection work necessary in the worst case to evacuate all live objects from fromspace W_{max}. The minimum amount of memory available in tospace for allocation of new objects is denoted F_{min}, as illustrated by figure 2. The minimum GC ratio, or GCR_{min}, is defined as:

$$GCR_{min} = \frac{W_{max}}{F_{min}}$$

We furthermore define the *current GC ratio*, denoted *GCR*, as the ratio between performed garbage collection work, and the amount of new, allocated, objects in tospace. During the GC cycle, until all live objects have been evacuated from fromspace, the garbage collector makes sure that the present GC ratio is higher than, or equal to, the minimum GC ratio. That is:

$$GCR \geq GCR_{min}$$

In this way we guarantee that fromspace will be empty before tospace is filled, even in the worst-case scenario. This strategy is essentially the same as found in [Bak78] and a more thorough analysis of the strategy can be found in [Hen94].

Allocation

New objects are allocated at the top of tospace. Using Dijkstras terminology [Dij78] new objects are allocated black.

Allocation of memory by *low-priority processes* are checked to guarantee that the present GC ratio doesn’t drop to low, i.e below GCR_{min}. If it threatens to do so, the

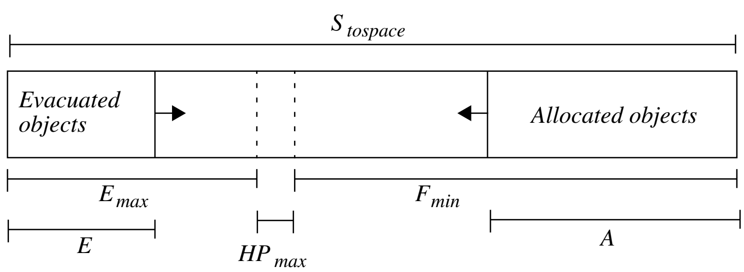

Figure 2 *The structure of tospace. A minimal area, HP_{max}, is kept available for the high-priority process at all times.*

garbage collector is given priority. The actual allocation of the new object is done first when sufficient garbage collection work has been performed. The upper bound on the GC work performed in connection with an allocation will be proportional to the size of the allocated area.

When the *high-priority process* allocates memory no garbage collection is performed, making it an extremely cheap operation. This scheme means that the ratio between performed GC work and allocated objects, GCR, might temporarily drop below its lower limit, GCR_{min}, but the garbage collector is invoked immediately after suspending the high-priority process in order 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 reserve an amount of memory in tospace equal to the worst-case allocation need of the high-priority process during one activation, denoted HP_{max}. The minimum amount of memory available in tospace for allocation of new objects, F_{min}, is thus calculated as:

$$F_{min} = S_{tospace} - E_{max} - HP_{max}$$

In this way, we will have a buffer to be used by the high-priority process if necessary. We can further guarantee that the evacuation of fromspace will be finished before tospace is filled up so a semi-space flip can be performed. Normally, the evacuation is finished well before tospace fills up, in which case we delay the flip until the amount of available free memory drops to low. After execution of the high-priority process a flip is performed if the remaining free memory is to small, i.e the following condition is violated:

$$S_{tospace} - E - A > HP_{max}$$

A memory allocation request by a low-priority process triggers a flip before servicing the request if the allocation would violate the condition.

Pointer assignment

Having a write barrier means that pointer assignments have to 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 into tospace and the pointer is updated to point to the new location. Pointer assignments by *low-priority processes* cause immediate evacuation if called for.

Since a pointer assignment may mean evacuating an object, which is a comparably expensive operation, it should be avoided in a high-priority process. The write barrier for *high-priority processes* employs a delayed-evacuation strategy similar to that used by Nilsen and Schmidt in their hardware-assisted garbage collector [NS94], but we are not relying on custom hardware. Assignments do not cause objects to be evacuated directly. The copying is instead delayed until the high-priority process is suspended. At the time of assignment we reserve space for the object in tospace and updates the pointer to point to the reserved memory area. The forwarding pointer of the reserved area is set to point to the actual object in fromspace. We may thus temporarily have forwarding pointers pointing *from an object in tospace to a corresponding copy in fromspace*. In the fromspace copy of the object a flag word used by the garbage collector doubles as a pointer to the reserved area in tospace. In this way subsequent pointer assignments involving the same object can avoid erroneously reserving additional memory in tospace and correctly modify the pointer to point to the reserved area in tospace.

Our garbage collector uses two different implementations of the write barrier depending on the priority of the process. Different methods could be employed to select which implementation to use at each individual pointer assignment. A test could be performed to check for priority or the write barrier could be called indirectly through a pointer modified when switching process. In both cases the overhead would be two machine instructions. These overheads could be eliminated by using a compiler that are aware of the priority of the processes and emits customized code for each type of process. Alternatively, the delayed-evacuation approach could be used also for the low-priority processes. This would, however, add a small amount of total overhead to the garbage collector.

5 System description

In this section we describe the functionality of the memory manager and the corresponding process scheduler. We focus on how the components interact with each other and with the application program.

5.1 Operations on the memory manager

The operations supported by the memory management module are divided into two classes: those called from the application program and those called from the process scheduler.

Application support
Operations used by the application to access and modify the object graph:

- *Allocate memory* – A new object is allocated on the heap. If called by a low-priority process, the collector performs an amount of garbage collection work proportional to the size of the newly allocated object. In the case of a high-priority process invoking the operation, this step is delayed. The garbage collection work is instead performed when the '*Clean up after HP process*' operation is invoked by the process scheduler.
- *Pointer assignment* – Implements the write barrier. The action taken once again depends on whether the calling process has high or low priority. For low-priority processes, actions are taken immediately to ensure that the assignment does not compromise the integrity of the heap. For high-priority processes, the assignment is merely registered to be checked as the process is suspended.

- *Dereference pointer* – The use of a scheduler based on preemption of processes gives rise to the need of keeping the garbage collector from moving an object or updating a pointer while it is being accessed by the application program.

Process scheduler support
In order to schedule the collector work in such a way that the processes of the application program is disturbed as little as possible, the memory manager must cooperate with the process scheduler. The operations enabling this cooperation are given below.

Operations:

- *Suspend GC* - When the running process is preempted in order to assign the processor time to another process with higher priority, the garbage collector can be within a critical region. The scheduler calls this operation to force the garbage collector out of the critical region in order to perform task switching with minimum latency.
- *Clean up after HP process* - When a high-priority process is suspended this operation is called by the scheduler to perform the neglected garbage collection work.

5.2 The process scheduler

The process scheduler provides three operations for the memory manager to call.

Operations:

- *GC active* – Informs the process scheduler that the garbage collector is within a critical region. If a task switch is initiated while within the critical region the kernel calls the *'Suspend GC'* operation on the memory management module. As a result the memory manager will invoke the *'GC inactive'* operation. This operation is preferably implemented by just raising a flag, i.e with a single machine instruction.
- *GC inactive* – The garbage collector is no longer within a critical region. Thus, the scheduler can hereafter preempt the running process without informing the memory manager.
- *Enable/disable task switching* – To guard against other processes causing the garbage collector to move objects as a pointer is de-referenced, task switching is temporarily disabled during the access. Pointer dereferencing is such a cheap operation, just a few machine instructions, that there's no point in allowing task switching during the access.

The state diagram in figure 3 illustrates how the scheduler works together with other components in the system. A few not previously mentioned operations regarding process management are introduced in the diagram.

- *Interrupt* – a hardware interrupt transferring the control to the process scheduler.
- *Resume* – the action of transferring the control to another process. The process gaining control continues executing at the point at which it was previously interrupted. (By an interrupt or a suspend.)
- *Suspend* – transfers the control from the currently running process to the process scheduler.

The initial state is *Idle*. When a process is to be scheduled the scheduler enters the *Scheduling* state. When turning over the control to a process, the scheduler enters one of two states depending on the priority of the process in question. Suspending a high-priority process causes the scheduler to enter the *Cleaning up* state. In this state, the scheduler waits for the memory manager to perform the garbage collection work neglected while running the high-priority process.

Low-priority processes can be suspended in two different ways, voluntarily or by preemption. Normally this means just returning to the *Scheduling* state, but if garbage collection is in progress, it is somewhat more complicated. Before preempting the low-priority process, we must make sure that the memory manager is in a consistent state. When garbage collection is performed, the memory manager forces the scheduler into the *Garbage collecting* state. Interrupts arriving while in this state leads to a transition to the *Schedule?* state. If the process is to be preempted, the scheduler orders the memory manager to leave the critical region, awaiting completion in the *Suspending GC* state.

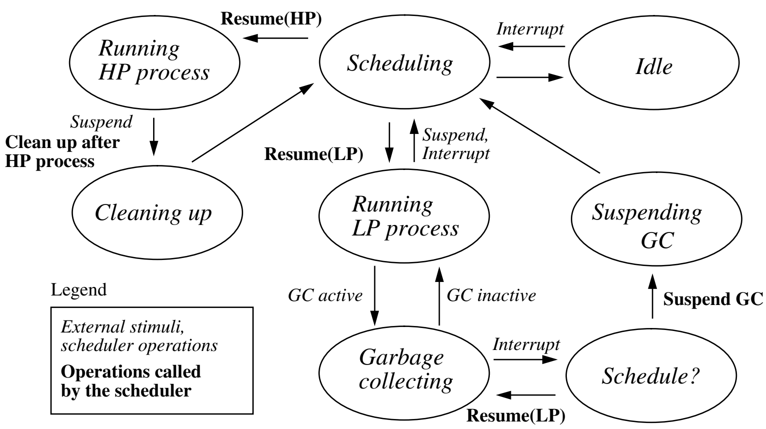

Figure 3 *State diagram for the integrated scheduler handling both processes and garbage collection.*

5.3 A scheduling scenario

To illustrate how the memory manager, the process scheduler, and the application program work together we present a small excerpt from the execution of a typical control application. At the start of the example, a low-priority process is executing and requests a new area of memory from the memory manager. While this request is being serviced, an interrupt triggers the scheduler which decides to restart a high-priority control process. The control process runs for a short duration, it too requesting memory from the memory manager. When the control process is suspended, the scheduler decides to resume the execution of the low-priority process, which is given its requested area of memory.

Figure 4 shows the scenario described above in the form of an interaction diagram. The different modules are represented as vertical lines showing the activity of the module as a function of time. The time is increasing downwards in the diagram. Arrows show how the execution point is transferred between the modules.

Walking through the scenario

Here we walk through the scenario step for step. The encircled numbers in the diagram corresponds to the steps below.

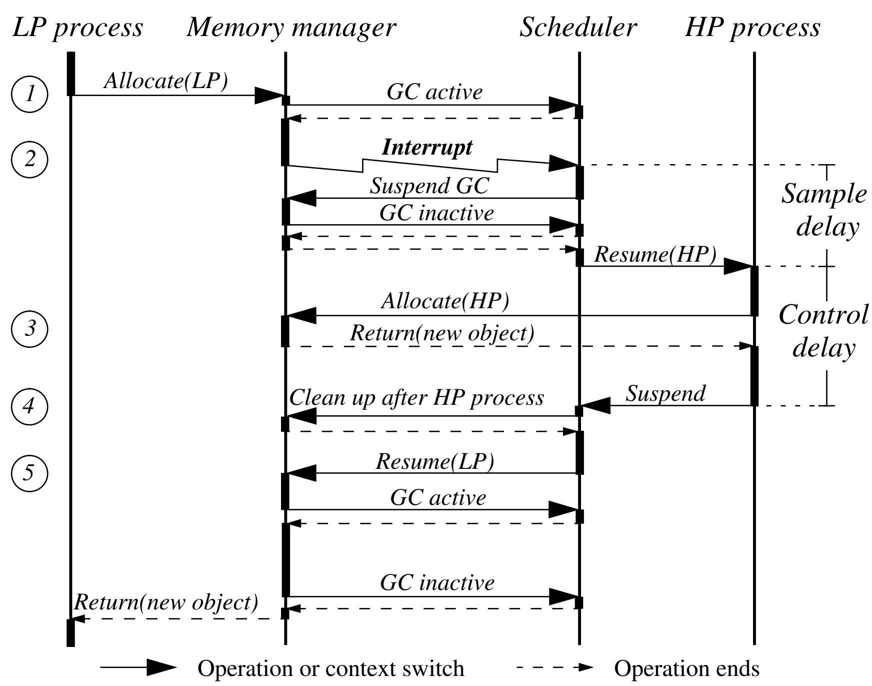

Figure 4 *IComplex scheduling scenario: A high-priority process interrupting a low-priority process during GC.*

1. A low-priority process, denoted *LP process* in the diagram, is running. To allocate memory, it invokes the operation *Allocate* on the memory manager. This will cause the memory manager to perform some garbage collection work. To prohibit the process scheduler from interrupting the memory manager while it is within a critical region, the operation *GC active* is performed on the scheduler before the garbage collection work is begun.
2. While the memory manager is performing garbage collection, the hardware generates an *interrupt* transferring the control to the scheduler. The scheduler decides that it is time to restart a high-priority control process, in the diagram denoted *HP process*. Since the memory manager is within a critical region, the operation *Suspend GC* forces the memory manager out of the critical region. The memory manager acknowledges that it is in a safe state by invoking the *GC inactive* operation on the scheduler. Returning from the *Suspend GC* operation, the scheduler resumes the high-priority process.
3. The high-priority process requests memory from the memory manager using the *Allocate* operation. No GC work is performed.
4. When the high-priority process suspends itself, the scheduler invokes the *Clean up after HP process* operation to perform the missing GC work.
5. The scheduler resumes the interrupted low-priority process. The GC work is continued and the scheduler informed that the memory management is entering a critical region through a *GC active* operation. A pointer to a new block of memory is finally returned to the low-priority process.

The sample delay (at step 2 in figure 4) is made up from the time needed to service the interrupt, selecting which process to resume, suspending the garbage collector, and finally the context switching to the high-priority process. The only additional step introduced in our scheduler, as compared to a traditional scheduler, is to suspend the memory manager. As we will see in the next section, this step is very small compared to the total time needed for a context switch.

6 Costs and worst-case overhead

A programmer trying to prove that a real-time application will meet its deadlines needs to know the worst-case overheads, also those caused by the garbage collection scheme. This section presents the overheads that can be expected when using our garbage collection strategy. When we mention the number of machine instructions required in the worst case we refer to the number of instructions that would have to be executed on a Motorola 68040 microprocessor.

6.1 High-priority process, control delay estimate

The overheads of garbage collection activities for a high-priority process are tight bounded delays for memory allocation, pointer dereferencing, and pointer assignment.

Memory allocation
Our scheme guarantees that memory allocations by the high-priority process can always be performed without GC work. Allocation involves moving a heap pointer and initializing object header information. In most cases there is also a cost to initialize internal pointers of the object to null. The overhead thus consists of some 10 instructions plus approximately 1 store-instruction per word of the object.

Pointer dereferencing
A simple read barrier is required by the underlying garbage collection algorithm: Pointer dereferencing is done indirectly through a forwarding pointer in the head of the object. The overhead is 1 machine instruction.

Pointer assignment
The worst-case overhead of the write barrier implied by our scheme, as seen by a high-priority process, is 12 machine instructions.

The individual worst-case overheads are tight enough to make it possible to guarantee that the high-priority process will meet the deadlines of a control system. The worst-case overheads of individual operations are small and fixed for the frequent operations (pointer manipulation). For memory allocation the overhead is proportional to the size of the allocated object. Calculating the worst-case control delay for the high-priority process our costs are comparable with costs for other operations and can not contribute with an uncontrolled cost.

6.2 High-priority process, sample delay estimate

The worst-case sample delay consists of two parts: the time needed by the garbage collector to finish on-going GC work, and the time needed by the process scheduler to perform the actual process switch. There will be an initial cost of about 30 machine instructions. To that will be added the time necessary to copy the object. Most control applications will tolerate such delays without putting a too narrow limit on the maximum object size. If, however, shorter response times are required it is possible to use an alternative approach, letting the GC back out of on-going copying. This will bring down the total cost to around 40-50 instructions, or some 1-2 microseconds. It will, however, slightly affect the worst-case total overhead of the GC scheme since an object might have to be copied twice every time the high-priority process is activated. The time needed by the process scheduler to perform the actual process switch will in any case stand for the major part of the sample delay.

Another issue to consider is how much time will be needed for GC work in between two activations of the high-priority process. The amount of work needed depends on how much memory was allocated the last time the high-priority process executed. The work may involve copying delayed-evacuation objects, but it is *not* required that all these objects are copied before the next execution of the high-priority process.

A flip may also have to be performed between executions of the high-priority process. The flip itself adds very little to the overhead since it does not involve more than changing the meaning of fromspace and tospace. When it is time to perform a flip our strategy guarantees that all live objects have been copied to tospace and that all pointers, including root pointers, have been updated. It can thus not occur when there are delayed-evacuated objects. The flip can be performed by some 10 machine instructions.

6.3 Low-priority processes, overhead

The garbage collector inflicts somewhat different overheads for the soft real-time part of a control system implemented by low-priority processes than for the high-priority process.

Memory allocation
When a low-priority process requests memory the garbage collector is invoked to ensure that the current GC ratio does not drop to low. In the worst case an amount of garbage collection work proportional to the size of the new object is performed. The exact worst-case cost of an allocation operation depends on many parameters such as heap size, maximum amount of live objects, etc.

Pointer dereferencing
Pointer dereferencing is always done indirectly via a forwarding pointer in the object header. This incurs an overhead of 1 machine instruction. Disabling/enabling process switching during dereferencing costs another 2 instructions. Again this cost can be eliminated if a cooperating compiler keeps temporary results in registers known by the garbage collector.

Pointer assignment
The write barrier for low-priority processes evacuates the target object if it is located in fromspace. This means that the worst-case overhead of a pointer assignment is proportional to the size of the target object or roughly equal to the time needed to make a copy of the object.

Also for the low-priority processes the worst-case overhead is small. Memory allocation time is hardest to estimate but should still be small compared with demands from soft real-time processes. The scheme should allow a large group of processes to be handled with low priority.

6.4 Overhead for the programmer

Our garbage collector requires some information about the application program in general and the high-priority process in particular to schedule the GC work such that it does not interrupt the high-priority process. The programmer must specify the *maxi-*

mum allocation need of the high-priority process during one activation, which is used to calculate the size of the memory buffer reserved for use by the high-priority process. The programmer must also specify the *maximum amount of memory* occupied by live objects, which is used to calculate the minimum GC ratio.

7 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++, and in a straightforward way. In a production quality version of the garbage collector, we expect the language and compiler to cooperate by providing information for the memory manager and to emit efficient and transparent inlined code to implement the write barrier etc. Currently, the programmer is expected to provide the runtime type information needed by the garbage collector. He must also make sure that pointer handling is done in compliance with a set of rules, invoking the right operations where expected to.

Little concern was given to optimizing the implementation, the main motivation for the prototype being to prove that very short response times can be achieved, rather than to actually achieve maximum efficiency. For example, the write barrier code consists of a number of procedure calls, but for high-priority processes, this could be replaced by in-lined code consisting of 12 machine instructions. Similar deficiencies are associated with allocation and pointer dereferencing, a very common operation.

Measurements on the control application show that the sample delay of the high-priority process is about 0.2 ms. The major part of this time is spent administrating the context switch. The time needed to suspend on-going garbage collection at the time of the context switch depends on the granularity of which the collector can be interrupted. In our control application, using small objects, about 20 µs is needed to suspend the garbage collection. We have chosen to complete a started object copying, which makes the time needed dependent on the maximum object size. An alternative would be to roll back the copying, especially if the objects are big.

The execution time for one activation of the high-priority process grew from 0.9 ms to 1.0 ms when garbage collection was introduced. The control process allocates an object every time it is activated, fills it with plot data, and inserts it into a linked list to be later taken care of by a separate plotter process. This overhead can be attributed to the inefficient implementation of the read and write barriers. As mentioned earlier these operations can, in a more ambitious implementation, be implemented by 1 and 12 in-lined machine instructions respectively. In such an environment we would expect a total increase of the execution time of the high-priority process of some 10 µs.

The individual longest delays imposed on the low-priority processes by the garbage collector are in the order of 0.5 ms. It should be noted that the existence of processes with higher priority causes a low-priority process to be interrupted for considerably longer periods than that. It it thus reasonable to assume that the garbage collector-induced delays will not significantly affect the execution of the low-priority processes.

Future work

The thing closest at hand is to generalize our strategy to handle *several concurrent high-priority processes*. A naive way to do this would be to treat them as a single process with a period equal to that of the fastest one and with a maximum allocation need equalling the sum of the allocation needs of all the processes. This seems, however, to be a very crude solution. We are convinced that we can devise much better strategies than that.

Currently, we assume that the high-priority processes are 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 can be incorporated in our model. For example, if we know that there is a minimum interval, say 10 ms, between two occurrences of an event, the triggered process can be handled as a clock-driven process with a period of 10 ms. Knowing that the event cannot occur more than a certain number of times within a fixed time interval could also help. In this latter case, we would probably need to reserve a somewhat larger area in tospace for the high-priority processes, enabling them to execute several times in succession without GC work in between.

We would also like to improve the situation for the *low-priority processes*. To guarantee that the GC cycle is finished before we run out of free memory in tospace, we use the worst-case amount of live objects in fromspace as a base for our calculation of the needed GC rate. In practice, this is often a very pessimistic assumption. As a consequence, all the GC work is done early in the GC cycle instead of being evenly distributed over the cycle. 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 rate. If necessary, the GC rate would have to be increased at the end of the cycle to handle the worst case. The pay-off would be twofold: First, the GC work would be more evenly distributed, thus causing shorter individual pauses in the average case, and second, the total amount of GC work will drop somewhat since objects in fromspace will be given more time do die and thus not needing to be evacuated. The price we would have to pay is a slightly worse worst case for the low-priority processes.

Finally, it would be interesting to study what *language support* is needed to make real-time garbage collection transparent to the programmer and well integrated with process handling. Today's 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.

8 Conclusions

We have presented a new approach to scheduling garbage collection work which enables use of dynamic memory 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 garbage collection work is scheduled 'as deadline scheduling' based on memory allocation rather than time. For high-priority processes the worst-case memory need gives a minimal size of free memory that must be available at all times for their use. By using this amount as 'low-water mark' the scheduling policy ensures that high-priority processes are never delayed by the garbage collector. The scheme adds very little to worst-case execution times for operations performed by the high-priority processes which is a valuable property when doing performance evaluations.

The algorithm and scheduling policy has been implemented in a real-time control system (implemented in C++). Experience from using it in a small control application confirms the claims. High-priority processes are scheduled with very little additional delay, much less than the 1 ms clock used, and complete on time. Also the observed disturbance for low-priority processes was very small although it might potentially be larger in a heavily loaded system.

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. We would also like to thank the anonymous referees for many constructive comments.

References

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

[ADVW92] J. L. Armstrong, B. O. Däcker, S. R. Virding, M. C. Williams. Implementing a Functional Language for Highly Parallel Real Time Applications. *Proceedings of SETSS 92*, Florence, Italy, 1992.

[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.

[Col60] G. E. Collins. A Method for Overlapping and Erasure of Lists. *Communications of the ACM*, December 1960.

[Deu76] L. P. Deutsch, D. G. Bobrow. An Efficient, Incremental Automatic Garbage Collector. *Communications of the ACM*, September 1976.

[Dij78] E. W. Dijkstra, L. Lamport, A. J. Martin, C. S. Scholten, E. F. M. Steffens. On-The-Fly Garbage Collection: An Exercise in Cooperation. *Communications of the ACM*, November 1978.

[Fen69] R. Fenichel, J. Yochelson. A Lisp Garbage Collector for Virtual Memory Computer Systems. *Communications of the ACM*, November 1969.

[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.

[HM92] R. L. Hudson, J. Eliot, B. Moss. Incremental Collection of Mature Objects. *Proceedings of IWMM'92*, St. Malo, France, September 1992.

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

[McC60] J. McCarthy. Recursive Functions of Symbolic Expressions and their Computation by Machine, Part I. *Communications of the ACM*, April 1960.

[Min63] M. L. Minsky. *A Lisp Garbage Collector Algorithm Using Serial Secondary Storage*. Memo 58 (rev.) Project Mac, M.I.T., Cambridge, Mass., December 1963.

[NS94] K. Nilsen, W. J. Schmidt. A High-Performance Hardware-Assisted Real-Time Garbage Collection System. *Journal of Programming Languages*, 1994(2).

[SG95] J. Seligmann and S. Grarup. Incremental Mature Garbage Collection Using the Train Algorithm. *Proceedings of ECOOP'95, European Conference on Object-Oriented Programming*, Aarhus, Denmark, August 1995.

[Tho76] L-E. Thorelli. A Fast Compactifying Garbage Collector. *BIT*, 16(4), 1976.

[UJ88] D. Ungar, F. Jackson. Tenuring Policies for Generation-Based Storage Reclamation. *Proceedings of OOPSLA'88*, ACM SIGPLAN Notices 23(11).

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

[Wad76] P. L. Wadler. Analysis of an Algorithm for Real Time Garbage Collection. *Communications of the ACM*, September 1976.

[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.

[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.