Time-Triggered Garbage Collection

Robust and Adaptive Real-Time GC Scheduling for Embedded Systems

Sven Gestegård Robertz
sven@cs.lth.se

Roger Henriksson
roger@cs.lth.se

Department of Computer Science
Lund University
Box 118, SE-221 00 Lund, Sweden

ABSTRACT

The advent of Java and similar languages on the real-time system scene necessitates the development of efficient strategies for scheduling the work of a garbage collector in a non-intrusive way. We propose a scheduling strategy, *time-triggered garbage collection*, based on assigning the collector a deadline for when it must complete its current cycle.

We show that a time-triggered GC with fixed deadline can have equal or better real-time performance than an allocation-triggered GC, which is the standard approach to real-time GC. Also, by using a deadline-based approach, the GC scheduling and, consequently, real-time performance, is independent of a complex and error-prone GC work metric.

Time-triggered GC allows a more high-level view on GC scheduling; we look at the GC cycle level rather than at the individual work increments. This makes it possible to schedule GC as any other thread. It is also suitable for making the GC auto-tuning by dynamically adjusting its deadline as necessary.

We have implemented our approach in a run-time system for Java and present experimental data to support the practical feasibility of the approach.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Memory management, run-time environments*

General Terms

Reliability, Performance

Keywords

real-time, garbage collection, scheduling, auto-tuning, embedded systems

1. INTRODUCTION

Memory management has traditionally been handled in a very conservative manner in embedded systems. For reasons of safety and predictability, static memory management has often been the technology of choice. As embedded systems grow in complexity, dynamic memory management becomes increasingly desired. Although more flexible than static memory management, manually managed dynamic memory tends to inflict new problems of predictability, robustness, and maintainability — important properties of embedded systems. Many of these problems can be overcome if automatic memory management, or *garbage collection (GC)*, is utilized. With the advent of type-safe languages like Java on the real-time systems scene [7, 10] it becomes increasingly important to develop garbage collectors with the following properties:

1. The garbage collector should be reliable, predictable, non-intrusive, and capable of meeting the memory allocation demands of our applications at all times.
2. The garbage collector should be transparent to the application developer and not require cumbersome manual tuning to be effective on any particular platform or hardware configuration.

This paper describes how a garbage collector can be designed and scheduled to meet these demands.

We propose a concurrent garbage collector, i.e. it is executing as a separate thread. In order to make it possible to preempt the garbage collector thread without long latencies we require that a fine-granular incremental garbage collection algorithm is employed. Any fine-granular incremental algorithm should work provided an upper bound on the amount of GC work required to conclude a GC cycle can be found. Our approach is thus not dependent of any particular GC algorithm. Using either knowledge of the worst-case allocation need of the application, or by using auto-tuning techniques, we calculate a deadline for when the GC thread must complete its cycle and make new memory available for allocation. Establishing a deadline for the GC thread means that we can schedule it using standard scheduling techniques, such as rate monotonic or earliest deadline first scheduling. Thus, we do not have to care about the scheduling of individual GC increments, but leave this to the standard thread scheduler. The thread scheduler ensures that the garbage collector will be allocated enough CPU time provided that the system is schedulable. The schedulability of the system can in turn be decided by the use of standard scheduling analysis techniques, e.g. generalized rate monotonic analysis [18]. Since the elapsed time determines when to run the garbage collector, we call the approach *time-triggered garbage collection*.

The scheduling model we propose has several benefits compared to previous techniques. Since the collector is concurrent and quickly preemptable, the scheduler can quickly suspend it if necessary to allow near-deadline application threads to execute. It also means that application threads are not delayed in connection with memory allocations as they would be if a traditional incremental collector were used. In such a system, a burst of allocation could easily result in a missed deadline due to accumulated GC work. The single scheduling requirement that the garbage collector must finish before its deadline makes it especially suitable for earliest deadline first (EDF) systems, for which we have not seen any similar systems.

Paper outline

The rest of the paper is organized as follows: Section 2 describes garbage collection techniques in general as well as presents some of our previous work which forms a basis for the rest of the paper. Section 3 describes our novel approach to scheduling garbage collection and how it can be applied to hard real-time systems. Section 4 shows how the techniques described in the preceding section can be modified in order to achieve automatic tuning of the GC cycle length. Section 5 reports on our initial experimental validation of the technique. Section 6 relates the work presented here with previous work while our future research plans are outlined in Section 7. Section 8 summarizes the paper and presents our conclusions.

2. BACKGROUND

A *garbage collector* is the part of the runtime system responsible for automatically identifying memory blocks no longer accessible by the *mutator*[1], or application program, and reclaiming them for later reuse. The type of garbage collectors we consider in this paper work by periodically traversing the pointer graph of the mutator in order to identify objects still reachable from the mutator. The memory occupied by objects *not* visited in this way is considered garbage and is reclaimed. For example, *copying* garbage collectors do this by moving, or *evacuating*, all objects encountered during the pointer graph traversal to a separate area in memory. The contents of the previously used memory area can then be considered garbage and reused during the next period. Other algorithms, e.g. mark-sweep algorithms, perform the GC work in multiple passes. The first pass might for example mark all reachable objects while the next pass enters not marked blocks into a free-list.

Each GC period corresponds to a *GC cycle*. A new GC cycle starts each time a complete garbage collection traversal has been completed and all of the reclaimed memory blocks have been made available for allocation (e.g. after all reachable objects have been evacuated by a copying collector and the previously occupied memory has been freed). Early garbage collectors performed all GC work at the end of the GC cycles when no memory remained for allocation, in effect halting the application until the GC cycle ended and the reclaimed memory was made available. This is clearly unsuitable for use in real-time systems.

Research within the field of real-time garbage collection has been on-going since the 1970s. The earliest attempts to implement non-intrusive garbage collectors used a technique called incremental GC. Here, the GC work required for a GC cycle is split into a number of very small *work increments* that can be interleaved with the execution of the application. In order to guarantee progress of the GC, a number of GC work increments are performed in connection with each memory allocation request. An example of such an algorithm is Baker's algorithm [5]. Let F_{min} denote the minimum amount of memory available for allocation during a GC cycle, a denote the amount of memory requested, and W_{max} denote the maximum amount of GC work (according to a given metric[2] and corresponding unit) that might be required to complete a GC cycle. Then, the size w of the GC work increment that must be performed in connection with the allocation in order to guarantee that we do not run out of memory before the GC cycle is complete is:

$$w \geq W_{max} \cdot \frac{a}{F_{min}} \tag{1}$$

Incremental GC triggered by allocation requests has at least two major disadvantages. Firstly, even if the overhead incurred by a single GC increment is small, a burst of allocation requests can lead to long accumulated delays. Secondly, in order to keep the cost of each GC increment within a low upper bound we might need a complex GC work metric in order to decide when to end each increment. When a simple metric is used, e.g. based on measuring the number of evacuated objects in a copying garbage collector, an increment which should be short according to the metric can in reality take a long time to perform. For example, we might have to traverse a significant amount of pointers in order to find just one object to evacuate. Thus increasing the performed amount of work according to the metric by one may require an unbounded amount of actual work. Performing GC at the time of allocation does make it easy to prove that the garbage collector will always keep up with the application, but it also means that it suffers from the inherent problem of GC work always being performed when application threads run — thus causing interference.

The problem of GC work always being performed when application threads run can be overcome by making the GC work *concurrent*, i.e., assigning the GC work to a separate GC thread executing in parallel with the application threads. This is a strategy applied by a number of garbage collectors, e.g. the Appel-Ellis-Li collector [3], but it has not been much used in real-time settings. Typically, no provision is made for guaranteeing that the collector keeps up with the allocation demands of the application.

[1]The application program is often denoted *mutator* when discussing memory management since it changes, or *mutates*, the pointer graph constituting the data structures of the application.

[2]We define a *GC metric* as a method of measuring how much GC work has been performed at a given point in time. This can be approximated in several ways, for example by counting the number of pointers that have been traversed by the GC or by the actual time spent performing GC. A GC metric is used in one way or another by all real-time garbage collectors in order to decide how to schedule the GC work.

In order to satisfy the demands of hard real-time systems, a technique must be found to schedule the work of a concurrent GC such that the application is guaranteed to meet all of its hard deadlines. In [9] we have previously presented such a scheduling technique. Here, we assume an embedded system with a number of high-priority (typically periodic) threads which have to meet hard deadlines. It can be observed that in most embedded systems, a relatively small number of such threads exist. Apart from these, low-priority (periodic or background) threads are often executing with more relaxed deadline requirements. This leads us to the following idea: Do not perform any GC work when the high-priority threads are executing. Instead, we assign the work motivated by high-priority allocations to a separate GC thread which is run when no high-priority thread is executing. When invoked, it performs an amount of GC work proportional to the amount of memory allocated by the high-priority threads. Since we may temporarily get behind with the GC work in this way, there must always be an amount of memory reserved for the high-priority threads. Slightly modified generalized rate monotonic analysis [18] can be used both for calculating the amount of memory which need to be reserved and to verify that the garbage collector thread will always keep up with the high-priority threads. GC work motivated by low-priority threads are performed incrementally at allocation time. Since GC work is partly performed concurrently and partly incremental in such a system we call the approach *semi-concurrent scheduling*.

The effect of this scheme is that we can guarantee hard real-time performance for threads that actually require it in a system scheduled by a fixed-priority scheduler. Since GC work is not performed while high-priority threads run we can allow ourselves to use a slightly coarser GC work metric without affecting real-time performance. An imperfect metric will only prevent low-priority threads without hard deadlines to execute as often as they would prefer.

The approach still has some drawbacks, however. One drawback is that it is not immediately suitable for systems with EDF schedulers. Another is that we always have to do an amount of scheduling analysis in order to tune the collector to any target platform.

3. TIME-TRIGGERED GC

Traditionally, incremental garbage collectors have been scheduled based on the allocations of the application — for each unit of allocation, a corresponding amount of garbage collection work is performed. We propose a different approach where we use time, instead of allocation, as the trigger for GC work. That is, garbage collection is scheduled to make the GC cycle finish at a certain time, rather than after a certain amount of allocation.

In [16] the idea of time-based GC scheduling and having a fix GC cycle length was introduced. That made it possible to determine how much memory will be allocated during a cycle or to reserve a certain amount of memory for the next cycle while still making it possible to perform schedulability analysis and give real-time guarantees on the run-time system in a straight-forward manner. In that work, we used a hybrid approach, with time-triggered GC that was scheduled using a traditional work metric in a fixed-priority scheduled system.

This paper presents time-triggered GC scheduling more thoroughly. We show how the GC cycle time can be calculated in order to guarantee sufficient GC progress. We discuss how the process scheduling strategy affects a time-triggered GC scheduler and argue that time-triggered GC makes it possible to schedule the garbage collector as any other thread under both RMS and EDF schedulers. We also show how time-triggered GC can be used to achieve the same objectives under a deadline-based process scheduler as the semi-concurrent scheduling strategy does in a fixed-priority system.

The main areas where time-triggered garbage collection scheduling has impact are:

Concurrent GC in deadline-based systems: In order to schedule GC in a way that we can give real-time guarantees while still disturbing the mutator (application) threads as little as possible in a deadline-based system, we want to be able to schedule the GC just as any other thread. With time-triggered GC, this property is inherent in the model, as the only scheduling parameter is the deadline, and we explicitly specify the deadline of the current garbage collection cycle.

GC work metric concerns: A traditionally scheduled incremental GC relies on some kind of work metric to determine whether it is in sync with the mutator or needs to perform more GC work. Therefore, such a GC relies on the accuracy of the metric and using a poor metric may cause poor real-time performance. With time-triggered GC, the actual scheduling is done by the process scheduler and therefore independent of the work metric. Thus a poor metric does not affect the real-time properties of the run-time system. This allows us to separate the problems of schedulability analysis[3] and run-time scheduling.

Bursty allocation: Applications often show bursty allocation patterns. This means that an allocation-triggered GC would have a bursty execution pattern. Time-triggered GC scheduling does not have this problem as the run-time scheduling of GC work is independent of mutator activity.

Unified GC scheduling: GC schedulers based on a traditional GC work metric are tightly coupled to the actual garbage collector implementation. By using a time-based approach to GC scheduling, it would be possible to separate the GC scheduler from the GC algorithm; using time as both the trigger and the GC work metric provides a simple interface between the GC and the scheduler. Also, as time is easy to measure directly, time-based GC scheduling fits very well into a feedback scheduling framework.

3.1 GC cycle time calculation

With time-triggered garbage collection, there is no direct connection between the GC scheduling and the application other than the allocation rate. The GC cycle time is the only parameter that controls the progress of the garbage collector. Thus, a time-triggered GC needs correct (or conservative) cycle time estimates in order to make real-time guarantees as each garbage collection cycle must be completed before the application runs out of memory.

[3]That, of course, still requires worst-case execution time analysis.

This section shows how an upper bound on the GC cycle time, which guarantees that the application never runs out of memory, can be calculated.

The following symbols will be used in this section: period time (T), frequency (f), heapsize (H), total amount of allocated memory on the heap (A), amount of memory allocated during a cycle (a), free memory (F), live objects (L), floating garbage[4] (G), amount of memory reclaimed during a cycle (r), the set of threads ($\mathbb{P}$), and the allocation per period of thread j (a_j).

Lemma 1. *For a set of processes, $\mathbb{P}$, with frequencies f_j, allocation requirements of a_j bytes per period and F bytes of memory available at the start of the GC cycle, an upper bound on the GC cycle time that guarantees that the cycle will be completed before the available memory is exhausted is*

$$T_{GC} \leq \frac{F - \sum_{j \in \mathbb{P}} a_j}{\sum_{j \in \mathbb{P}} f_j \cdot a_j} \tag{2}$$

Proof A GC cycle must finish before the available memory at the start of the cycle has been allocated. That is,

$$a = \sum_{j \in \mathbb{P}} \left\lceil \frac{T_{GC}}{T_j} \right\rceil \cdot a_j \leq F \tag{3}$$

where the ceiling is to cover the worst case schedule. A stronger condition is

$$\sum_{j \in \mathbb{P}} \left(\frac{T_{GC}}{T_j} + 1 \right) \cdot a_j \leq F \tag{4}$$

Substituting $f_j = \frac{1}{T_j}$ we get

$$\begin{aligned} \sum_{j \in \mathbb{P}} (T_{GC} \cdot f_j + 1) \cdot a_j &= \\ \sum_{j \in \mathbb{P}} T_{GC} \cdot f_j \cdot a_j + \sum_{j \in \mathbb{P}} a_j &= \\ T_{GC} \sum_{j \in \mathbb{P}} f_j \cdot a_j + \sum_{j \in \mathbb{P}} a_j &\leq F \end{aligned} \tag{5}$$

$$\therefore \quad T_{GC} \leq \frac{F - \sum_{j \in \mathbb{P}} a_j}{\sum_{j \in \mathbb{P}} f_j \cdot a_j}$$

□

The amount of free memory needs some further discussion. Since any incremental garbage collector suffers from the problem of floating garbage, we must take that into account when calculating the worst case amount of memory available at the start of a GC cycle (F_{min}). Put differently, we may not be able to use all the free memory in the current cycle if we want to be sure that there is also enough memory for the next cycle as the amount of memory that is reclaimed by the garbage collector can vary from one cycle to another due to floating garbage.

We start by examining floating garbage.

[4]Floating garbage is objects that are no longer reachable by the mutator but are still believed to be live by the collector. For example, objects that die shortly after they have been marked will not be reclaimed until in the next GC cycle.

Lemma 2. *Let a^n be the amount of memory that is allocated during the nth GC cycle and L_{max} be the maximum amount of live memory. Then, the sum of live memory and floating garbage at the start of cycle $n+1$ satisfies the inequality*

$$L^{n+1} + G^{n+1} \leq L_{max} + a^n \tag{6}$$

Proof Let δ^n be the net change in live memory during cycle n:

$$L^{n+1} = L^n + \delta^n \tag{7}$$

Let u^n be the amount of memory that becomes unreachable during cycle n. Then,

$$\delta^n = a^n - u^n \implies u^n = a^n - \delta^n \tag{8}$$

which gives

$$\left. \begin{aligned} G^{n+1} &\leq u^n = a^n - \delta^n \\ L^{n+1} &= L_n + \delta^n \end{aligned} \right\} \implies L^{n+1} + G^{n+1} \leq L^n + a^n \tag{9}$$

But $\forall n, L^n \leq L_{max}$, which concludes the proof. □

In order to make hard guarantees, we must determine the maximum amount of memory that can be allocated during a GC cycle without risking that the system runs out of memory due to floating garbage.

Lemma 3. *Let H be the heapsize and L_{max} be the maximum amount of live memory. Then, the maximum amount of memory that can be safely allocated during a GC cycle is*

$$a_{max} = \frac{H - L_{max}}{2} \tag{10}$$

Proof The heap contains allocated and free memory

$$H = A + F = L + G + F \tag{11}$$

and therefore,

$$F = H - (L + G) \tag{12}$$

Applying Lemma 2 to (12) gives that, at the start of any GC cycle,

$$F \geq H - (L_{max} + a_{max}) = F_{min} \tag{13}$$

Thus, the worst case occurs when $L = L_{max}$, and the remainder of the proof makes this assumption. Then the system has to be in steady state[5] and the maximum amount of floating garbage during a worst case cycle is

$$G^{WC}_{max} = a_{max} \tag{14}$$

An upper bound on the amount of memory allocated during a GC cycle must, of course, not be greater than the minimum amount of available memory so the trivial bound is $a_{max} \leq F_{min}$. We will now prove the equality. Objects that are floating garbage at the start of cycle n will have been reclaimed by the start of cycle $n+1$, which means that

$$F^{n+1} \geq G^n \tag{15}$$

[5]This means that for each allocated object, another object becomes unreachable.

The amount of available memory at the start of cycle $n+1$ is

$$F^{n+1} = F^n - a^n + r^n \tag{16}$$

Cycle n is a worst case cycle ($F^n = F_{min}$) iff the amount of floating garbage at the start of the cycle is at the maximum ($G^n = G^{WC}_{max}$). In the worst case, $r^n = G^n$, which corresponds to equality in (15). Applying this to Equation (16) gives

$$F^{n+1} = F_{min} - a^n + G^{WC}_{max} = G^{WC}_{max} \quad \Longrightarrow \quad a^n = F_{min} \tag{17}$$

Consequently, we can allocate all available memory during a worst case cycle while still guaranteeing that the amount of available memory at the start the following cycle is no less than F_{min}, i.e.,

$$a_{max} = F_{min} \tag{18}$$

Finally, equations (13) and (18) give

$$a_{max} = \frac{H - L_{max}}{2}$$

□

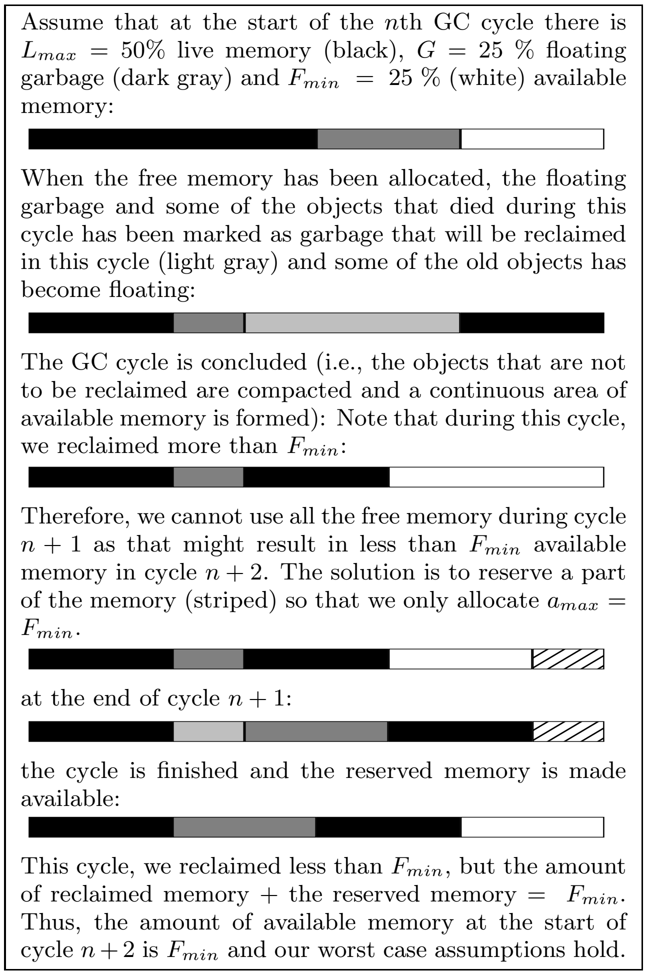
Because the amount of floating garbage may vary, depending on how the execution of the application and the garbage collector are interleaved, the amount of memory reclaimed will also vary from cycle to cycle. Therefore, we cannot always allocate all of the available memory if we want to guarantee that the system never will run out of memory. Consequently, the length of the garbage collection cycles must be calculated based on the worst case amount of available memory.

THEOREM 1. *An upper bound on the GC cycle time that guarantees that we always will have enough memory available for allocation is*

$$T_{GC} \leq \frac{\frac{(H-L_{max})}{2} - \sum_{j\in\mathbb{P}} a_j}{\sum_{j\in\mathbb{P}} f_j \cdot a_j} \tag{19}$$

PROOF The theorem follows from lemmas 1 and 3. □

For an example of how varying amounts of floating garbage affects the amount of available memory, see Figure 1. Note that, somewhat counter-intuitively, the dangerous case is when there is *less* than the worst case amount of floating garbage, as this could lead to a situation where we allocate too much memory if care was not taken to avoid that.



Assume that at the start of the nth GC cycle there is L_{max} = 50% live memory (black), G = 25 % floating garbage (dark gray) and F_{min} = 25 % (white) available memory:

When the free memory has been allocated, the floating garbage and some of the objects that died during this cycle has been marked as garbage that will be reclaimed in this cycle (light gray) and some of the old objects has become floating:

The GC cycle is concluded (i.e., the objects that are not to be reclaimed are compacted and a continuous area of available memory is formed): Note that during this cycle, we reclaimed more than F_{min}:

Therefore, we cannot use all the free memory during cycle $n+1$ as that might result in less than F_{min} available memory in cycle $n+2$. The solution is to reserve a part of the memory (striped) so that we only allocate $a_{max} = F_{min}$.

at the end of cycle $n+1$:

the cycle is finished and the reserved memory is made available:

This cycle, we reclaimed less than F_{min}, but the amount of reclaimed memory + the reserved memory = F_{min}. Thus, the amount of available memory at the start of cycle $n+2$ is F_{min} and our worst case assumptions hold.

Figure 1: Example of a how the amount of floating garbage may vary between cycles and how our reservation strategy guarantees that there always will be at least F_{min} available memory at the start of a cycle.

It may seem that the limit on the amount that may be allocated during a garbage collection cycle may cause unnecessarily low memory utilization but this isn't the case; the limit on the amount of memory that may be allocated during a GC cycle, according to Equation (10), only affects the cycle time calculations. It is true that in the best case (when we have no floating garbage) at most half of the available memory is allocated during a cycle, but this has nothing to do with the total memory utilization. If the GC cycle time is reduced, the amount of allocation per cycle — and, consequently, the maximum amount of floating garbage — is also reduced. This means that if both high allocation rates and high memory utilization is required, the GC cycles will be short, but as long as $L_{max} < H$ and there is enough CPU time to accommodate both application and GC, the system is guaranteed to work.[6]

3.2 Scheduling

This section discusses how time-triggered GC scheduling can be implemented in fixed priority and deadline based systems, respectively and how the general process scheduling policy affects the scheduling of garbage collection. It also relates time-triggered GC scheduling to semi-concurrent scheduling and handling of background tasks.

Based on the cycle time calculations presented in Section 3.1, we can use standard scheduling techniques (e.g., RMS or EDF) and schedule the GC as any other thread since the scheduling of individual GC increments is implicit; the only real requirement is that the GC cycle has ended and enough memory is made available before the application runs out of memory. As the deadline is the sole scheduling parameter, this means that the GC work calculations are only needed for schedulability analysis and not for ensuring

[6]Note that the problems associated with floating garbage are intrinsic to incremental GC and not a consequence of our strategy.

GC progress at run-time. Hence an error in the GC metric alone cannot cause the GC to run too slowly, which gives a more robust system. If the system is schedulable, the GC will finish on time, without causing any other thread to miss its deadline.

In systems where hard real-time tasks co-exist with background tasks without strict timing requirements, we want hard guarantees that the GC always will make memory available to the real-time tasks on time. We also want to avoid unnecessary disturbance of the background tasks. Conversely, we want to protect the GC from the background tasks in the sense that allocations performed by a background task must not cause the GC to miss its deadline or fail to make enough memory available for the real-time tasks. These problems are addressed by the semi-concurrent GC scheduling strategy.

When implementing a semi-concurrent garbage collector under the aforementioned scheduling policies, the main difference is that in a fixed priority system we must explicitly schedule each GC increment in order to spread the garbage collection overhead evenly across the cycle. That is, each time the garbage collector is invoked, it has to determine how long that increment should be (according to the metric used). When enough work has been performed, the GC must suspend itself until the next increment is triggered. Otherwise, the garbage collector thread might starve low priority[7] threads for long periods of time. In an EDF system, the scheduling of GC increments can be left to the process scheduler, as there are no fixed priorities and, thus, no risk of starvation.

A consequence of the requirement that the garbage collector must determine the length of each increment is that the actual scheduling will depend on both the cycle time and the work metric. In an EDF system, the only scheduling parameter is the deadline, and the garbage collection thread can be scheduled like any other thread. Therefore the run-time scheduling is independent of the GC work metric, worst-case memory usage, and execution time analysis. This is a big advantage in practice, as worst-case estimates are often based on measurements rather than exact analysis.

3.2.1 Fixed priority scheduling

In a fixed priority system, a higher priority thread always get precedence over lower priority threads. Therefore, a semi-concurrent GC must spread the GC work evenly across the whole cycle and not do more work in each increment than absolutely necessary. Otherwise, the low priority threads could be subjected to unnecessary starvation and excessive jitter. Thus, some GC work metric has to be used to determine if the garbage collector has made enough progress.

Naturally, for a given GC cycle time, T_{GC}, all the garbage collection work required to complete a GC cycle has to be performed before T_{GC} seconds have elapsed. In order to ensure sufficient GC progress, the GC scheduler should maintain the invariant

$$\sum w \geq W_{max} \cdot \frac{t - t_{cycle\ start}}{T_{GC}} \tag{20}$$

That is, the fraction of GC work performed should be greater than or equal to the fraction of the cycle time elapsed. This corresponds to Equation (1) with time instead of allocations as the trigger, on the right hand side. Scheduling garbage collection according to this invariant ensures that progress will be made at a well-defined rate regardless of if, and when, the application allocates memory.

3.2.2 EDF scheduling

The first property of semi-concurrent scheduling, non-intrusiveness, is inherent in the EDF model; if the requested CPU utilization is less than 100%, all deadlines will be met. Furthermore, as the period times of real-time threads typically are much shorter than the GC cycle time, the GC is unlikely to interrupt the real-time threads. In systems where low jitter is important, statically scheduled I/O can be used as in e.g. the control server model [8].

The second property of the semi-concurrent model, isolating the high priority threads from the low priority ones, and thus not having to do worst-case analysis on the low-priority threads, can in an EDF system be achieved by using Constant Bandwidth Servers (CBS) [1] with the addition of an importance attribute for the servers. Then, the high-priority and low-priority threads in the semi-concurrent model would correspond to high-priority and low-priority servers.

In such a model, the threads running on high-priority servers would just do allocations without any GC penalty, while the threads on the low-priority servers would do incremental GC at allocation time. When incremental GC is performed due to a low-priority allocation, both the deadline and execution time of the GC thread should be decreased as the memory allocation has reduced the amount of available memory and the incremental GC work has brought the GC cycle closer to its finish. Moving deadlines to an earlier point in time is, however, not allowed in an EDF system in the general case as this causes a temporary increase in the requested CPU utilization and might lead to missed deadlines. This could be solved by temporarily reducing the bandwidth of the low-priority server with a corresponding amount or, if the remaining CPU time in the low-priority server's budget is too low, delaying the allocation that would cause incremental GC work until the next CBS period. In practice, however, this is not a problem as the GC cycles typically are much longer than the period times of the threads and therefore the deadlines and/or server bandwidths can be adjusted at the thread release times when it is safe to do so.

Another way to make sure that the memory management overhead may never cause the critical parts of the application to miss their deadlines was presented in [16]. By introducing priorities for memory allocations, the run-time system is able to automatically prioritize memory allocation requests (i.e., deny non-critical allocations) in order to guarantee that the system will not run out of memory or become unschedulable because of a too high GC workload. In essence, this can be viewed as dividing the application into critical aspects, which are guaranteed to be executed on time and non-critical aspects, which are only executed if it is safe to do so.

4. AUTO-TUNING GC SCHEDULING

As we have seen, the GC cycle length can be calculated at design-time based on the allocation requirements of the high priority threads. If this is not practical for some reason

[7]Here, we use the word "priority" in a sense that corresponds better to the RTSJ notion of "importance" than the RMS sense of the word "priority". In the semi-concurrent model, the low priority threads are background tasks without firm deadlines.

(for instance that the application's execution pattern varies greatly depending on operating mode or that it is to be run on many different platforms and we do not want to do analysis for all possible target platforms or even know which platform it will run on) or if we want the GC scheduler to be completely transparent to the developer we have to use some adaptive technique to measure and control the GC scheduling parameters on-line.

A very simple model is to measure or estimate the allocation rate ($\dot{a}$) of the application. We can then calculate at which time all the currently remaining free memory (F) will have been allocated — the GC cycle's deadline.

$$T_{remaining\ this\ cycle} = \frac{F}{\dot{a}} \tag{21}$$

which, $T_{elapsed}$ seconds into the GC cycle, gives the cycle time

$$T_{GC} = \frac{F}{\dot{a}} + T_{elapsed} \tag{22}$$

This simple model for on-line cycle time calculation performs well if roughly the same amount of memory is reclaimed in each GC cycle but it suffers from the same problems with floating garbage as described in Section 3.1, although the symptoms are a bit different. In the previous case, the system might run out of memory if the GC cycle time was too long. In an adaptive system, the cycle time will be tuned to ensure that this does not happen so the problem in this case is that the system might become unschedulable.

One example of this that we encountered in our experiments with this simple model is that if there, for some reason, is much floating garbage during one cycle, little memory will be freed during that cycle. Then, the following cycle will have to be very short and we get a memory trace like the one shown in Figure 2. This could cause real-time problems since the required CPU utilization of the GC will be much higher during the short cycles than during the long ones, as the amount of GC work is roughly the same in all cycles[8], but it has to be done in a much shorter time in the short cycles.

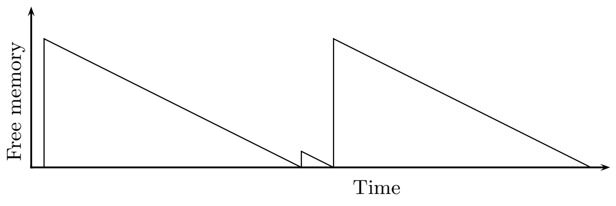


Figure 2: Example of a very short GC cycle caused by large amounts of floating garbage.

In order to handle variations in the amount of floating garbage, we need to reserve memory so that the allocations during the next cycle can be satisfied even if no objects are reclaimed during this cycle. Let $\hat{\dot{a}}$ be the estimated allocation rate and $\dot{a}^{next}$ be the allocation rate for the next cycle. Then $T_{GC} \cdot \dot{a}^{next}$ will be allocated during the next GC cycle and we get

$$\hat{T}_{GC} = \frac{F - \hat{T}_{GC} \cdot \dot{a}^{next}}{\hat{\dot{a}}} + T_{elapsed} \tag{23}$$

$$\implies \quad \hat{T}_{GC} = \frac{F + \hat{\dot{a}} \cdot T_{elapsed}}{\hat{\dot{a}} + \dot{a}^{next}} \tag{24}$$

If we assume that the mutator will continue at the measured allocation rate, i.e., $\dot{a}^{next} = \hat{\dot{a}}$, we get

$$\hat{T}_{GC} = \frac{1}{2}\left(\frac{F}{\hat{\dot{a}}} + T_{elapsed}\right) \tag{25}$$

If the allocation rate is constant, this means that we should reserve half of the available memory at the start of a cycle for the allocations during the next GC cycle. Doing so guarantees[9] that we can handle the worst case, i.e., when all the objects that died during a cycle became floating garbage and will not be reclaimed until at the end of the next GC cycle.

Only allocating at most half of the available memory each GC cycle is the price we pay for incrementality. Note that this only affects the length of the GC cycles and not the overall memory utilization. If, for instance, the amount of live memory is 80% of the heap, the GC cycle length would be set so that 10% of the total memory is reserved for the next cycle.

It should also be noted that a copying collector [5] by design has the property of reserving a part of the available memory for the next cycle so this is only a concern with mark-sweep type collectors and it is not a problem to implement a mark-sweep collector so that it only makes memory available at the GC cycle boundaries. There also exist mark-sweep type algorithms with more than one allocation area that, by design, have this behaviour (e.g., Bengtsson's [6]).

5. EXPERIMENTAL VERIFICATION

A simple control system was implemented to test the proposed technique for auto-tuning GC scheduling. For the experiments, we used a lab process with a ball on a beam. The angular velocity of the beam is controlled in order to roll the ball to a given position on the beam, see Figure 3.

The control was performed by a Java application consisting of three threads: a user interface (low priority), a reference generator (high priority) and a controller (high priority). The UI thread sends setpoints to the reference generator which does rate limiting and sends reference values to the controller thread. In addition to doing the actual control, the controller thread sends log data back to the user interface thread. The reference generator and controller were run at between 20 and 100 Hz. The UI was run at 0.5 Hz.

The experiments were performed using compiled Java [15] on a 350 MHz PowerPC running the STORK [2] real-time kernel. The garbage collector used is an incremental mark-compact collector and the traces were collected by instrumenting the memory manager and the RT-kernel with logging calls at memory operations and context switches. Logging was done to a dedicated memory area and dumped via a serial line after the experiment.

[8]Of course, this depends on the garbage collection algorithm as well as on implementation details. However, the execution time of a garbage collector typically depends on both the amount of retained and reclaimed memory. Even algorithms where there is no explicit *free* operation, like for instance a copying collector, have a fraction of the cost that is proportional to the amount of reclaimed memory if, e.g., the initialization of memory is taken into account.

[9]Given, of course, that the total amount of available memory is sufficient.

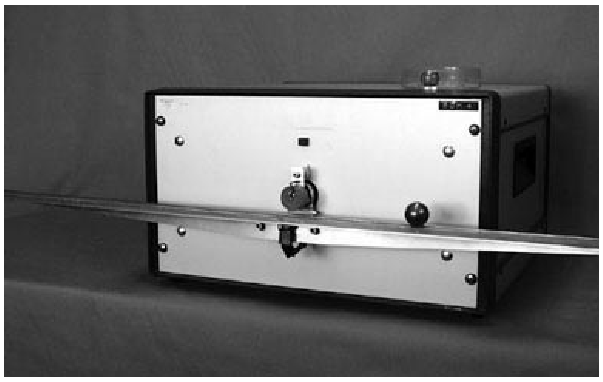


Figure 3: The ball-on-beam process. The beam can be rotated to roll the ball to the desired position.

Figure 4 shows a memory trace of the system with the auto-tuner enabled. The fast threads run at 100 Hz. Figure 5 shows how the auto-tuner reacts to changes in allocation rate. At $t = 10$ s, the frequency of the high priority threads is increased from 20 to 100 Hz and at $t = 20$ s the frequency is lowered to 20 Hz. The GC is scheduled so that it will work even if all the dead objects in one cycle would be floating garbage. I.e., we reserve a part of the available memory for the next GC cycle as described in Section 4.

As memory allocations typically are bursty, the measurement of the allocation rate is filtered in order to keep the deadline estimates more stable and reduce the update frequency for the scheduling parameters. Care must be taken not to underestimate the allocation rate, as this might lead to an out-of-memory situation, so we must react quickly to actual changes in allocation rate while avoiding chatter due to bursty allocations. The rise time in the allocation rate plots are due to such filtering.

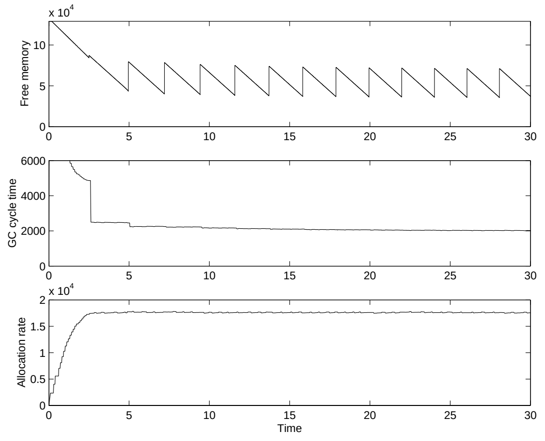


Figure 4: Memory trace of the system with adaptive GC cycle length. The topmost plot shows the amount of available memory (in bytes), the middle plot shows the estimated GC cycle length (in milliseconds) and the bottom plot shows the LP filtered allocation rate measurement (in bytes/second). The time is given in seconds.

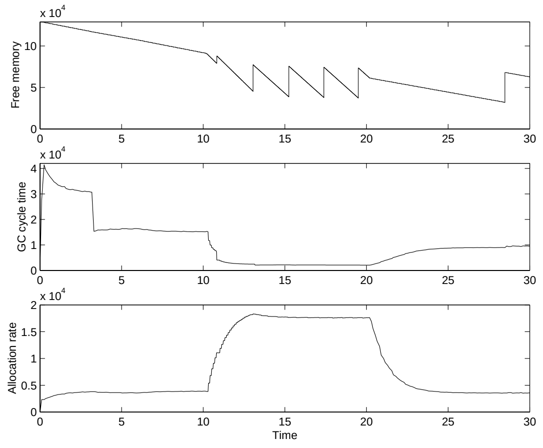


Figure 5: How the GC scheduler reacts to changes in allocation rate; At $t = 10$ s, the frequency of the high priority threads is increased from 20 to 100 Hz and at $t = 20$ s the frequency is lowered to 20 Hz.

6. RELATED WORK

Incremental and concurrent GC

The fields of fine-grained incremental and concurrent GC algorithms are well explored. Examples of an incremental collector is Baker's copying algorithm [5] which we described in Section 2, but several other exist. The problem with algorithms of this type is that they are typically scheduled in connection with allocation requests. That means that GC work will always be performed when application threads run. Even if each increment is made small and predictable in length, it still means intrusion. This is especially noticeable when several allocation requests are clustered together.

An alternative approach is to run the GC as a separate thread. This approach was for example used by Nettles and O'Toole [14] in their replicating GC. Here, most of the memory management overhead can be avoided when executing the mutator (application) threads, but we have not seen any provision for guaranteeing that the collector always keeps up with the mutator threads.

A general issue when using concurrent GC is that it must be possible to quickly interrupt the GC thread. This is important in order to guarantee short latency for the application threads. Therefore, we must ensure that each atomic operation of the GC is very short and that the heap is in a consistent state after each atomic operation. In [9] it is illustrated how a variant of Baker's algorithm can be modified in order to produce a concurrent GC with minimal atomic work increments (order of microseconds).

For a more thorough presentation of the area we refer to [11] and the publications referenced therein.

Time-based GC scheduling

The problems of allocation-triggered GC scheduling in real-time systems, particularly the uneven GC overhead and consequentially, mutator CPU utilization, caused by variances in allocation rate, are addressed by David Bacon et al in a recent paper [4]. To achieve even and predictable mutator utilization, time-based scheduling, where the collector and mutator are interleaved using fixed time quanta, is proposed.

The work of Bacon et al is largely motivated by the same concerns and has much in common with the work presented in this paper. One fundamental feature of time-based GC scheduling common to both approaches is that they turn garbage collection into a periodic activity instead of a sporadic one as allocation-triggered GC does.

The main difference between the model proposed by Bacon et al and the time-triggered GC scheduling model presented in this paper lies in the level at which GC scheduling is considered; the period time of their model is at the quantum level while the period of the time-triggered GC is the GC cycle. Also, the fixed time quanta of [4] explicitly state how the GC work should be scheduled while the time-triggered model specifies a deadline and leaves the actual scheduling decisions to the underlying process scheduler.

The behaviour of the approach of Bacon et al is, at a large time scale, similar to that of a semi-concurrent GC or a time-triggered GC in that the CPU utilization of the mutator is predictable, consistent, and independent of bursty allocation rate of the mutator.[10] However, at a more fine-grained level, the garbage collector may still preempt the mutator as the GC is scheduled to run for one GC quantum after each mutator quantum. Here, the design goals behind their collector differ from the ones driving the work in this paper; they focus on low overhead and consistent utilization while non-intrusiveness and low GC induced latency and jitter are the key issues behind this paper.

Time-based GC was also proposed in [19] as a means to spread GC work more evenly and minimize the number of GC invocations and heap usage when the application's allocation pattern is bursty. The focus on that work is on measuring object lifetimes but they note that similar concerns are relevant in run-time systems for server applications.

Previous object life span studies have used an allocation-triggered approach, calling the GC every n kB of allocation, in order to measure object lifetimes. Qian et al supplement this with a time based approach by periodically performing a GC cycle, e.g., every 100 ms. No effort is made to ensure that the collector keeps up with the mutator since this is not a problem in their application; it is sufficient that the GC cycle time can be manually tuned to suit a particular application.

They also hint that the time-triggered approach can be applicable to embedded systems by using the timing information of the threads to run the GC when the number of live objects is small. The focus is still on efficiency and minimizing the number of GC invocations and they do not address any real-time issues.

7. FUTURE WORK

The auto-tuning presented in this paper uses a black-box approach; we do not require any knowledge of the internals of the garbage collector or mutator — the only quantity that is measured is the amount of available memory. This has the advantage that it is very easy to plug this kind of auto-tuner into an existing system, as very little communication with the memory subsystem, and none at all with the mutator, is required. The drawback is of course that we cannot react to changes in allocation rate until after they actually occur.

A more sophisticated model would take e.g., information that some threads are periodic (i.e., the knowledge that each thread does a certain amount of allocation during each invocation and then is idle until its next invocation) into account. That would make it possible to measure and estimate how much each thread allocates during each invocation which might even further mitigate the problems with apparently bursty and random allocation patterns. It would also allow us to use information about the execution patterns of the threads, for instance for feed-forward of changed sampling periods etc.

In this paper, we have shown how to find one of the GC's scheduling parameters, the deadline. The other important parameter, the execution time, could also be estimated by using automatic system identification techniques as discussed in [17]. This would make it feasible to fully incorporate the GC scheduling into a feedback scheduling system, where both the deadline and execution time is used since the scheduler does on-line schedulability analysis and dynamically changes the period times of the threads in order to keep the system's total CPU utilization below a certain level. Another interesting approach is to use Equation (5) or (19) to find safe allocation rates and, hence, period times, for each thread by finding a suitable set $\{a_j\}$ given T_{GC} and the CPU bandwidth available for GC and background threads.

The experiments presented in this paper are of a preliminary nature and the performance requirements of the application were modest. We currently work on implementing and evaluating the real-time performance of our time-triggered GC prototype under more challenging conditions in a high-performance robotics application.

Another area where the presented techniques may have impact are temporally predictable distributed systems. In a distributed system, the nodes can be seen as components and the whole system as being constructed by composition of node components. When designing such systems, one important factor is the ease of composing systems out of components, *composability* . The time-triggered architecture [12, 13] addresses the composability problem and important features of that model are time-triggered communication and temporal firewalls — interfaces between the components specifying what data should be available or communicated at what time. Such interfaces makes it possible to guarantee that if the individual components conform to their specified interfaces, the resulting system will work as intended. They also solve problems of safety critical systems like, for instance, maintaining a global timebase and determining data validity.

In order to utilize automatic memory management in such temporally predictable components, it seems it would be helpful, if not necessary, to be able to guarantee that also the memory manager is temporally predictable. As time-triggered GC scheduling has the property that it has an explicit deadline and therefore makes it possible to guarantee that a GC cycle finishes and makes a certain amount of memory available at a certain time, it would be interesting to study the impact of time-triggered GC in this field of application.

[10]The interleaving of GC and background processes in the semi-concurrent model may be almost identical also on a fine-grained level; quantization effects due to atomic GC primitives make a GC scheduled according to Equation 20 behave as a time-based GC with small GC and mutator quanta.

8. CONCLUSIONS

We have presented a strategy, based on time, for scheduling concurrent garbage collection in a hard real-time environment. Since GC work is triggered by elapsed time, as opposed to triggered at allocations, we avoid the problem of constructing a GC work metric that accurately models the temporal behaviour of the collector. This also makes it straight-forward to use a concurrent GC in hard real-time, EDF scheduled systems, which is a big advantage over allocation-triggered concurrent GC.

We have also shown how the scheduling strategy lends itself well to to adaptively tuning the GC speed according to the requirements of the individual application. Experimental verification has shown that the adaptive GC scheduling is capable of handling significant load changes without violating the real-time demands of the application. The benefits of an adaptive GC scheduler is twofold: It relieves the developer from having to manually tune the performance of the system, both in the initial development phase and during maintenance. It also makes an application more portable, as there is no need for calculating new scheduling parameters when the application is moved to another platform or the CPU load or memory availability is changed.

9. ACKNOWLEDGMENTS

This work has been financially supported by ARTES (A network for Real-Time research and graduate Education in Sweden), SSF (the Swedish Foundation for Strategic Research), and VINNOVA (the Swedish Agency for Innovation Systems).

10. REFERENCES

[1] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of the 1998 IEEE Real-Time Systems Symposium*, Madrid, Spain, December 1998.

[2] Leif Andersson and Anders Blomdell. A real-time programming environment and a real-time kernel. In Lars Asplund, editor, *National Swedish Symposium on Real-Time Systems*, Technical Report No 30 1991-06-21. Dept. of Computer Systems, Uppsala University, Uppsala, Sweden, 1991.

[3] Andrew W. Appel, John R. Ellis, and Kai Li. Real-time concurrent collection on stock multiprocessors. In *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, Atlanta, Georgia, June 1988.

[4] David F. Bacon, Perry Cheng, and V. T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of POPL'03*, New Orleans, Louisiana, USA, January 2003.

[5] Henry G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294, April 1978.

[6] Mats Bengtsson. *Real-Time Compacting Garbage Collection Algorithms*. Lic. eng. thesis, Department of Computer Science, Lund University, 1990.

[7] Greg Bollella et al. *The Real-Time Specification for Java*. Addison-Wesley, 2001.

[8] Anton Cervin and Johan Eker. The Control Server: A computational model for real-time control tasks. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, Porto, Portugal, July 2003. To appear.

[9] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[10] J-Consortium. *Real-time Core Extensions for the Java Platform*. International J Consortium Specification, 2000.

[11] Richard Jones and Raphael Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1996.

[12] Hermann Kopetz. Time-triggered real-time computing. *IFAC World Congress, Barcelona, July 2002, IFAC Press*, July 2002.

[13] Hermann Kopetz and Günther Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112 – 126, January 2003.

[14] Scott Nettles and James O'Toole. Real-time replication garbage collection. *SIGPLAN Notices*, 28(6):217–226, 1993.

[15] Anders Nilsson, Torbjörn Ekman, and Klas Nilsson. Real Java for real time – gain and pain. In *Proceedings of CASES-2002*, pages 304–311. ACM Press, October 2002.

[16] Sven Gestegård Robertz. Applying priorities to memory allocation. In *Proceedings of the 2002 International Symposium on Memory Management (ISMM'02)*, Berlin, Germany, June 2002. ACM Press.

[17] Sven Gestegård Robertz. *Flexible automatic memory management for real-time and embedded systems*. Lic. eng. thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2003.

[18] Lui Sha, Ragunathan Rajkumar, and John. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.

[19] Qian Yang, Witawas Srisa-an, Therapon Skotiniotis, and J. Morris Chang. Java virtual machine timing probes – a study of object life span and GC. In *Proceedings of 21th IEEE International Performance, Computing and Communications Conference (IPCCC)*, Phoenix, Arizona, April 2002.