Joint Garbage Collection and Hard Real-Time Scheduling

Maxime Van Assche, Joël Goossens, Raymond Devillers

Université Libre de Bruxelles, Département d'Informatique,
and Mission Critical IT
corresponding author: joel.goossens@ulb.ac.be

Abstract

We analyze the integration of automatic memory management in a real-time context. We focus on integrating a real-time (copying) garbage collector with hard real-time static-priority periodic tasks. This integration is done by considering the copying collector as an aperiodic task served by a polling server. We analyze the schedulability of this system for any polling server parameters. This analysis includes a bound on the memory size that guarantees sufficient memory for the periodic tasks.

Plan

Key works scheduling, garbage collection, sporadic server

1 Introduction

In this paper, we consider the problem of scheduling hard real-time systems composed of a periodic task set and a garbage collector. The advantages of automatic memory reclamation are nowadays somewhat obvious for non real-time as well as real-time systems. These advantages include: the absence of trouble to free memory, the avoidance of dangling pointers (i.e., a pointer referencing some memory freed manually), and no memory leaks. This leads to a better productivity during the implementation and the debugging. Memory leaks, in particular, are often hard to notice and even harder to locate. If a garbage collector is shown correct, then we are certain of never encountering dangling pointers or memory leaks that could result in a system crash. Let us recall however that a garbage collector needs some help from the programmer for yielding the expected behaviour: if a pointer to a structure is left in a variable, the structure will never be reclaimed, even if we know that the variable will never be used again.

A more subtle benefit of a garbage collector arises when sharing data between different entities of a program (e.g., objects, tasks). The manual disposal of such shared structures could create unwanted dependencies between those entities. In fact, these memory structures must be reclaimed by some entity, but only when all entities will not access them anymore.

An increasing number of real-time applications for embedded systems are written using high level programming languages to reduce the development effort. The evolution of processors and compilers, which become faster at lower costs, induced that increase. Automatic memory management in a real-time context is an evolution that is made with real-time Java [2], for instance.

However, automatic memory management and hard real-time systems are usually thought to be incompatible because of the garbage collector's disruptive behavior. It is important to notice that the garbage collection literature includes very good works concerning *incremental* garbage collectors (i.e., the execution of both the garbage collector and the tasks are interleaved). Very often, in that literature and the corresponding research community, these garbage collectors are considered to be real-time as well. But actually, they are not (at least with the common definition used in our real-time community/literature : we shall give more details about this question in the Section 2). To the best of our knowledge, very few works consider truly hard real-time constraints when garbage collection comes into the frame.

In this paper, we show that automatic memory management and hard real-

time systems are not incompatible, at least under our model of computation and our assumptions. We aim at presenting a first formalization of the *joint* scheduling, which consists of scheduling hard periodic tasks and a garbage collection together. Moreover, this work provides a schedulability test for a more specific case: scheduling static-priority periodic tasks with a copying collector (a particular garbage collector technique, introduced in Section 2.2).

Related work While there is some literature on truly real-time garbage collection, only a few works address the problem of scheduling the garbage collector.

Unlike our work, the scheduling approach adopted in most real-time garbage collection research papers is to associate an amount of garbage collection work to heap allocation instructions. For each memory unit allocated, a corresponding amount of garbage collection work is performed. The proportion of work is large enough to ensure that the garbage collection cycle is completed before the system runs out of memory. No matter how fast tasks use up memory, the collector execution time is increased correspondingly. The minimum amount of collection work ensuring sufficient memory is analyzed in [1, 24] for a copying garbage collector. As we justify below, this technique is *inadequate* for the real-time systems we consider.

Siebert [21] improves this scheme by making the amount of work performed at each allocation dependent on the amount of free memory available. Thus, the work to be done by the garbage collector is very small as long as there is sufficient free memory. However, this improvement does not make the technique suitable for hard real-time systems, which need a worst-case analysis.

Associating the collection work to allocation instructions inevitably penalizes tasks that allocate memory in the heap. In fact, a real-time task computation time then depends on the amount of memory it allocates, on the maximum amount of live memory of the whole system and on the heap size. This scheme does not provide the flexibility to distribute the collection overhead independently of allocations. For instance, if a system has a critical task with a high allocation rate and a constraining deadline, we cannot transfer the collection overhead on less critical tasks. This lack of flexibility is too restrictive for practical (real-life) real-time systems.

This collection-at-allocation scheduling technique is also chosen by Nilsen [17, 16], but he focuses on real-time garbage collection using dedicated hardware. By using special hardware circuits placed between the cache and the memory, Nilsen guarantees a worst-case delay for any individual operation. This ap-

proach suffers from the need for dedicated hardware that would be used only for the garbage collector.

Henriksson [5, 8, 15, 6, 7], Robertz [19, 20] and Magnusson [15], concentrate on integrating automatic memory management with high priority processes and low priority processes to increase the real-time robustness. They use a technique called semi-concurrent garbage collection scheduling. Three priority levels are identified: the high priority (hard real-time processes), the garbage collector priority and the low priority (soft real-time processes). The garbage collector never interrupts the high priority processes, but whenever a low priority process allocates memory, enough collection is performed to ensure sufficient memory for the high priority ones. This approach handles adequately the collection scheduling. However, the garbage collector is scheduled as a background task with respect to the high priority processes, which may cause a long response time for the garbage collector's cycle and consequently, the system may require a large heap to ensure sufficient free memory. In this work we generalize the framework, since the garbage collector is served by a polling server whose priority is *not* necessarily the lowest one.

More recently, Robertz and Henriksson [20] have analyzed this semi-concurrent scheduling technique when applied to the earliest deadline first model (EDF). Their approach does not suffer from the problem described previously, since the priorities are dynamic: the garbage collector is not scheduled as a background task anymore. But implementing dynamic priorities is generally harder than static ones, and analysing the worst-case response times is a lot more complex too.

Kim et al. [10, 11, 12] concentrate on scheduling the garbage collector as an aperiodic task using a sporadic server, and provide an implementation of a garbage collector adapted to real time systems. However, their analysis is limited to a sporadic server with the highest priority of the system, which is rather restrictive. The flexibility of the priority level at which the garbage collector is executed is necessary to optimize the heap size while preserving the schedulability of hard real-time tasks.

This research We analyze the integration of a garbage collector in a discrete time real-time system constituted of static-priority periodic tasks with constrained deadlines (for instance, scheduled using the rate or deadline monotonic scheduling [14, 13]). The garbage collector is scheduled as an aperiodic task using a polling server [22]. We provide the different steps necessary to verify the schedulability of the system, including *the minimum memory re-*

quirement analysis for any polling server parameters. This analysis pinpoints an amount of heap memory sufficient to guarantee that the system will never run out of memory. That amount of memory can be reached in worst-case scenarios.

Our contributions can be summarized as follows:

- we present a formalization of the *joint* scheduling problem which consists in scheduling hard real-time periodic tasks and a garbage collection together;
- we propose a schedulability test for a more specific case: scheduling static-priority periodic tasks with a copying collector;
- we characterize the worst case response time of an aperiodic task served by a polling server.

Outline Section 2 asks the question of what a real-time garbage collector is. Requirements on the garbage collector are described in relation to our view of a real-time system. Section 3 explains our technique of integration and the requirements on the garbage collector scheduling. Section 4 models the system and provides a schedulability analysis for this model.

2 Incremental Garbage Collector vs. Garbage Collector for Real-time Systems

Many garbage collectors are proclaimed real-time only because they are incremental and fast on the average. Baker [1] qualifies as real-time a system in which *the programmer would still be assured that each instruction would finish in a reasonable amount of time*. Obviously, this definition is not sufficient for the real-time community.

More specific requirements are defined by Wilson in [24] for a real-time garbage collection: it must be incremental; every pause made to collect garbage must be strictly bounded; it must make significant progress; the pauses must not occur too often: *for any given increment of computation, a minimum amount of the CPU is always available for the running application.*

These requirements are somewhat more specific and detailed, but they are not perfectly adapted to the real-time systems we consider. We separate two aspects of the requirements: the requirements on the garbage collector itself and

the requirements on the way the garbage collector is scheduled within the system. The second aspect is detailed in Section 3.

2.1 Real-time requirements

The systems considered in this work are composed of a set of hard real-time periodic tasks and a garbage collector. The main characteristic of real-time systems is the behavioral *predictability*. Timing constraints have to be met whatever happens in the system, provided its requirements are fulfilled; hence, it is not enough (and sometimes it is superfluous) that the tasks and/or the garbage collector are fast on the average: the worst case is more important (in fact it is the only one which is relevant).

Of course, the garbage collector must be *incremental*, in the sense that it can be preempted by the other tasks, and it must make *significant progress*, which implies that preemptions should not prevent it from collecting. For example, it would be unacceptable that some work has to be redone after each preemption, such that the collector never progresses. Also, the garbage collector must guarantee the *consistency* of the heap at all times.

Finally, to analyze a real-time system with a garbage collector, we must be able to compute or bound the collector's *worst-case computation time*, denoted C_{GC}. This value represents the maximum amount of CPU time necessary for the collector to accomplish one collection cycle. This value depends on the hardware, the run-time environment, but also on the amount of live memory (i.e., accessible memory) of the system when the collection is triggered, on the number of root pointers (i.e. the pointers outside the heap, referencing heap memory) and on other values, which are specific to the garbage collector algorithm.

2.2 Copying collector

Garbage collectors yield a very wide research area, both in theory and in practice. In particular, there are many families of garbage collector techniques (e.g., reference counting, mark and sweep, copying collector, etc.). It is beyond the scope of this manuscript to review the subject. For real-time systems, we believe that the copying collector technique is particularly interesting, because it avoids heap fragmentation (see [9, 1]). For that reason, in the following we will only consider copying collectors. We now introduce briefly this technique (notice that this paper does not contribute to the field of copying collectors but

to the *joint scheduling* of periodic tasks and of a copying collector). For copying collectors, the heap is split into two equal areas: *ToSpace* and *FromSpace*. The reclamation of unused memory is done implicitly by copying (and compacting) only the live memory from one space to the other. *ToSpace* contains the current objects. *FromSpace* contains the garbage from the previous collection. The beginning of a collection cycle starts with a flip, which exchanges the roles of *FromSpace* and *ToSpace*. A cycle ends when all the live memory has been copied in *ToSpace*.

Brooks' copying garbage collector [3] may easily be adapted to fulfill our requirements. This collector is incremental and uses a pointer write barrier to synchronize the mutators (i.e., the user programs) and the collector. Allocation is black, which means that new memory is allocated in *ToSpace*. The details of Brooks' collector is beyond the scope of this paper.

The garbage collectors described in [5, 6, 7, 8, 15] and in [10, 11, 12] are perfect examples of such copying collectors that can be integrated in the real-time system we consider. Different mechanisms are used to fulfill the requirements. The lazy copying technique makes the pointer write instruction bounded by a constant. To have allocation bounded by a constant, the garbage collector is responsible of initializing *FromSpace* after the copying phase.

The analysis we provide also applies for any other copying garbage collector that fulfills our requirements. Notice however that the analysis allowing to determine the worst case execution time for such a collector is not necessarily easy. In particular, the copying collector moves data structures and updates pointers to them, which may result in a modification in the tasks' behaviour with regard to the caches. Moreover, when the garbage collector is preempted, it is usually necessary to redo part of the last structure copying, and this extra work depends on the worst case number of preemptions incurred by one garbage collection cycle. To simplify the presentation, we shall in the following assume that the impact of the caches and of the preemptions on the execution time of the copying collector may be neglected.

3 Predictability

The joint scheduling of the tasks and the garbage collector must be *predictable*. We must be able to predict off-line if the system will be schedulable.

The requirements given by Wilson [24] suggest to choose a maximum CPU utilization rate for the garbage collector to execute. This scheduling technique

is quite simple, but it does not report when to execute the work increments such that the system remains schedulable. As we want the system to be predictable, scheduling the garbage collector through an aperiodic server is an effective solution.

We integrate the garbage collector in a real-time periodic task set by considering it as a real-time aperiodic task. It is aperiodic, because its arrival times depend on the memory utilization of the periodic tasks. It is real-time, because the collection must be over before the tasks run out of memory, hence before the next cycle of the collection is needed. The garbage collector will be serviced by a polling server [22], to interleave its execution with the periodic tasks.

In the following, we define the schedulability of a real-time system with a garbage collector. It contrasts from a classical schedulability definition such as the one in [4, 14], by taking into consideration the memory of the system, since the programmer has no control over the heap in a system with a garbage collector.

Definition 3.1 *A task set with a garbage collector is considered to be schedulable if*

- *each periodic task always completes before its deadline;*
- *periodic tasks never run out of memory;*
- *any allocation request is granted within a constant time.*

The first condition corresponds to the classical time constraints schedulability test of a task set. As programmers do not control the heap, the system itself must guarantee sufficient memory. The last condition states that the system cannot collect garbage at every allocation in order to have sufficient space; however, in order to allow the garbage collector to function correctly when serviced by the polling server, a (small) bounded extra work is generally necessary when allocating or accessing heap memory, which will be incorporated in the worst case execution time of the various real-time tasks. If we would allow to collect garbage at each allocation such that enough space is created in the heap to satisfy the allocation, the worst-case response times of the tasks would generally be too large to allow a feasible scheduling (and we would no longer need an extra server for the garbage collection).

4 Schedulability Analysis

4.1 Model of computation and assumptions

First, we provide a system model designed for real-time copying collectors fulfilling our requirements.

For each periodic task, we assume that its allocation rate, i.e., the maximum amount of memory allocated during one request of that task, is known (expressed in terms of some memory unit), but we make no hypothesis about when this memory is claimed by the task during its requests. Also, the maximum amount of live memory of the whole system is known (expressed with the same memory unit).

Definition 4.1 *A periodic task τ_i is specified by a 5-tuple $(C_i, T_i, D_i, O_i, A_i)$.*

- *C_i is the worst-case **computation** time of each request of τ_i (including the extra work needed to manage the heap in a way compatible with the copying garbage collector).*
- *T_i is the **period** of the task.*
- *D_i is the relative **deadline** of the task (we assume $0 < C_i \leq D_i \leq T_i$: the task has a constrained deadline).*
- *O_i is the **offset** of the task, i.e., the release instant of the task (O_i will be 0 for each i if the system is synchronous).*
- *A_i is the worst-case memory **allocation** during one request of the task.*

Definition 4.2 *A uniprocessor system Φ is specified by $(\Gamma, \Pi, \mathcal{L}, \mathcal{R})$, where $\Gamma = \{\tau_1, \ldots, \tau_n\}$ is a set of n periodic tasks, $\Pi = \{P_1, \ldots, P_n\}$, with $i \neq j \Longrightarrow P_i \neq P_j$, is the set of priorities assigned to the tasks (0 is the highest priority), $\mathcal{L}$ is the maximum amount of live memory of the whole system at any moment and $\mathcal{R} = \cup_{i=0\ldots n}\mathcal{R}_i$ is the the root set, i.e., the variable and stack pointers of each task ($i = 1..n$) or global to the whole system ($i = 0$) allowing to reach the live part of the heap.*

The tasks share a single heap (of maximal size M); the various tasks dynamically allocate memory in it, and the garbage collector will reclaim the unsused parts regularly or when needed. Memory in the heap can (but does not need to) be shared between tasks through pointers in global variables, but such objects

may also be referenced via local variables or stack variables. The root set is therefore composed of n stacks, n local variable sets and global variables. A stack is not necessarily emptied after each request; for instance, it may be used as a return-input between two requests of the same task. The maximum depth of each stack is fixed and known.

In order to simplify the presentation, we shall assume that all the numerical values characterizing the system are natural integers, i.e., we only consider discrete systems.

4.2 Elements of the analysis

This section details the different elements necessary to verify the schedulability of a real-time system with a real-time copying collector serviced by a polling server.

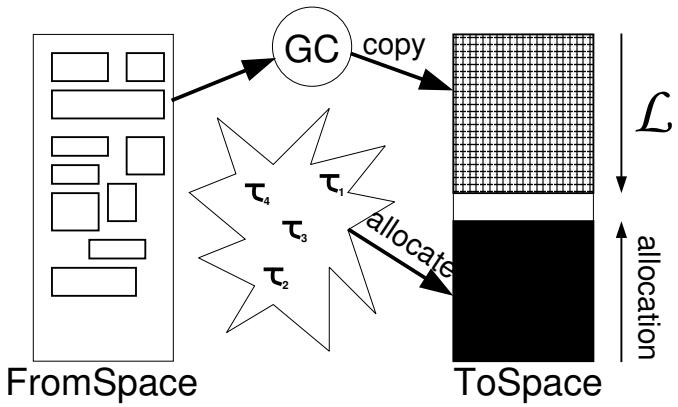
Foremost, a time constraints schedulability analysis must be performed. For each task in the task set (plus the polling server, see definition 4.4), its worst-case response time is computed and compared to its deadline. If for each task, the worst-case response time is smaller or equal to the task deadline, the task set is schedulable with this priority assignment. See [4, 23] for a complete description of such an analysis.

The second element of the schedulability analysis consists in computing the minimum memory requirement so that the tasks never run out of memory, even in the worst-case scenario. The minimum memory required depends on $\mathcal{L}$, on the allocation rates, and on the worst-case time between the beginning of a collection cycle and the earliest time the next collection cycle can start its execution (we shall formalize this notion further as t^{next}). We justify this statement in the following paragraph.

When the garbage collector starts executing to collect garbage, it performs a flip. Therefore the new *ToSpace* is empty at that time. In the worst-case scenario, the garbage collector copies $\mathcal{L}$ units of memory from *FromSpace* to *ToSpace*. As the periodic tasks and the collector share the CPU, the periodic tasks keep on allocating memory during the collection. This newly allocated memory is placed in *ToSpace*. Figure 1 shows a system with the garbage collector copying $\mathcal{L}$ units of memory and the aperiodic tasks allocating memory in *ToSpace*.

When the collection cycle is over, the tasks continue allocating memory in the same semi-space until a new role flip is performed, making the new *ToSpace* empty. Therefore, the size of *ToSpace* (in fact of each semi-space) must be

Figure 1: Minimum memory required for a real-time system with a real-time copying collector.



large enough to hold $\mathcal{L}$ plus the maximum amount of memory allocated during the longest time between the beginning of a collection cycle and the earliest time the next collection cycle can start its execution.

4.3 Aperiodic task response time analysis

In order to analyze the required memory for a system to be schedulable, we need a bound on the longest time between the beginning of a collection cycle and the earliest time the next collection cycle can start its execution. Since we consider the garbage collector as an aperiodic task, we make a more general analysis on the execution of an aperiodic task with known worst-case computation time using a polling server. To the best of our knowledge, this analysis is new and this section provides an additional contribution: the worst case response time of an aperiodic task served by a polling server.

Definition 4.3 *An aperiodic task τ_{ap} is characterized by C_{ap}, its worst-case **computation** time.*

Our polling server behaves like a periodic task, and uses its execution time to service aperiodic tasks. If the polling server is executing and no aperiodic task is ready to execute, it waits (but the capacity decreases) for an aperiodic task to execute until it is preempted by a higher priority task ready to execute

(which improves a little bit the responsiveness of the server in comparison with the basic polling server in [22]). During that waiting time, the time slots normally allocated to the polling server may be used to service ready periodic tasks (with lower priorities) or background non-real-time activities; they may be lost if nothing may be executed meanwhile; but in any case, the servicing capacity of the server is decremented as if the time slots were used by the latter. It is assumed that $C_{ap} > 0$.

Definition 4.4 *A polling server is defined by $\tau_S = (P_S, C_S, T_S)$, respectively the server **priority** ($P_S \neq P_i, i = 1 \dots n$), its maximum **capacity** (i.e., the CPU time budget the polling server has per period) and its **period**. It is assumed that $C_S > 0$, that the server is launched at the initialization of the system, hence at time 0, and that adding a periodic task $(C_S, T_S, T_S, 0, 0)$ with priority P_S to the current task set still respects the deadlines.*

We express the capacity of the server at time t by $\kappa(t)$, i.e., the amount of time the server has left at time t to execute until the end of its period. Notice that $\kappa(kT_S) = C_S \; \forall \, k \geq 0$, and if $\kappa(t) > 0$ and the time slot $[t, t+1)$ is attributed to the server, $\kappa(t+1) = \kappa(t) - 1$ whatever happens (i.e., even if the server does not use the slot for itself).

We also need to formalize any execution of an aperiodic task.

Definition 4.5 *The execution of an aperiodic task τ_{ap} by an aperiodic server τ_S in a system Φ is characterized by $(t^{\text{arriv}}, t^{\text{start}}, t^{\text{end}}, t^{\text{next}})$, where:*

- *t^{arriv} is the **arrival** time of the aperiodic task, meaning that it is ready to execute. It is then waiting for tasks with higher priorities than the polling server to complete, and/or for some server capacity.*
- *t^{start} is the time when the aperiodic task **starts its execution**.*
- *t^{end} is the **completion** time.*
- *t^{next} is the next available time, i.e., the earliest time at or after t^{end} when the aperiodic **server capacity is not zero**, and when the **CPU is available** for the aperiodic server (basically, the earliest time when a succeeding aperiodic task would be able to start its execution).*

Notice that $t^{\text{arriv}} \leq t^{\text{start}} < t^{\text{end}} \leq t^{\text{next}}$ and $t^{\text{end}} - t^{\text{start}} \geq C_{ap}$. The response time is the delay $t^{\text{end}} - t^{\text{arriv}}$ between the arrival time and the completion time.

It is easy to see that $t^{\text{next}} - t^{\text{arriv}} \leq (1 + \left\lceil \frac{C_{ap}}{C_S} \right\rceil) T_S$, but we shall try to get better bounds in the following.

Definition 4.6 *For a given system $\Phi = (\Gamma, \Pi, \mathcal{L}, \mathcal{R})$ and an aperiodic server τ_S, the worst-case response time of an aperiodic task τ_{ap} is the maximum response time over all its possible executions*

$$\mathcal{R}^{wc}(\Phi, \tau_S, \tau_{ap}) = \max_{t^{\text{arriv}}} \{t^{\text{end}} - t^{\text{arriv}}\}$$

The worst-case response time is the maximum response time over all executions. But t^{end} depends on τ_S, on Φ, on τ_{ap} and on t^{arriv}. Thus, the worst-case response time is the maximum response time over all arrival times. To simplify the notations, in the rest of this work, we use $\mathcal{R}_{ap}^{wc}$ to denote $\mathcal{R}^{wc}(\Phi, \tau_S, \tau_{ap})$. Notice that we have indeed a maximum, and not only a supremum, since we only work with integer numbers, and the intervals are bounded.

We can also define the longest time $\max_{t^{\text{arriv}}} \{t^{\text{next}} - t^{\text{start}}\}$ between the beginning of a service and the next time where a succeeding aperiodic request may start its execution. The following theorem shows that this value is never larger than $\mathcal{R}_{ap}^{wc}$.

Theorem 4.7 *For a given system Φ, a polling server τ_S and an aperiodic task τ_{ap},*

$$\max_{t^{\text{arriv}}} \{t^{\text{next}} - t^{\text{start}}\} \leq \max_{t^{\text{arriv}}} \{t^{\text{end}} - t^{\text{arriv}}\}$$

Proof: We may observe that, at t^{start} a time slot is used by the server, at t^{next} a time slot is available for the server, and between t^{start} and t^{next} exactly C_{ap} time slots were available (and used) for the server. Hence, if a request occurs at $t_*^{\text{arriv}} = t^{\text{start}} + 1$, its completion time will be $t_*^{\text{end}} = t^{\text{next}} + 1$. As a consequence, $t^{\text{next}} - t^{\text{start}} = (t^{\text{next}} + 1) - (t^{\text{start}} + 1) = t_*^{\text{end}} - t_*^{\text{arriv}} \leq \max_{t^{\text{arriv}}} \{t^{\text{end}} - t^{\text{arriv}}\}$, and $\max_{t^{\text{arriv}}} \{t^{\text{next}} - t^{\text{start}}\} \leq \max_{t^{\text{arriv}}} \{t^{\text{end}} - t^{\text{arriv}}\}$. □

Therefore, if we find a bound for the response time of an aperiodic task, we can also use it to bound any time period $[t^{\text{start}}, t^{\text{next}})$.

In order to demonstrate a bound for the worst-case response time of an aperiodic task using the polling server, we need to define the concept of worst and best-case response time of a periodic task with the same characteristics than the polling server task but with its computation time fixed to $x \leq C_S$.

Definition 4.8 *For $x \in \mathbb{N}$ with $x \leq C_S$, $\rho(x)$ is the worst-case response time of the task $\tau_S = (x, T_S, T_S, 0, 0)$.*

Theorem 4.9 *For $x \in \mathbb{N}$ with $x \leq C_S$ and for a synchronous (i.e., $O_i = 0 \; \forall i$) uniprocessor system Φ, $\rho_S(x)$ is the smallest positive solution of the equation*

$$\rho_S(x) = x + \sum_{\{j | P_j < P_S\}} \left\lceil \frac{\rho_S(x)}{T_j} \right\rceil C_j \, .$$

This is an immediate consequence of classical results on fixed priority periodic real-time systems (see for instance [4, 23]). This value may be effectively computed with a (generally fast) iterative procedure. Note that, for an asynchronous task set, the value computed with this theorem is a (non necessarily reached, but often rather good) upper bound of the worst response time.

Definition 4.10 *For $x \in \mathbb{N}$ with $0 \leq x \leq C_S$, $\rho^*_S(x)$ is the steady state best-case response time of the task $\tau_S = (x, T_S, T_S, 0, 0)$.*

"Steady state" means here that, in case of an asynchronous system, we first wait until the system becomes cyclic; for instance, we may consider the system from the time $\max_i\{O_i\} + \sum_i T_i$. $\rho^*_S(x)$ can be lower bounded by x, because it takes at least x time units to execute x computation units, but when we know the characteristics of the higher priority tasks, we can improve this bound by using the best-case response time analysis from [18]. This analysis uses a recurrence equation similar to the one from Theorem 4.9 to find the best-case response time of a periodic task in the periodic part of the schedule. This means that the value found with this analysis results in the best-case response time for any offsets, but only after the largest offset is elapsed. For a synchronous task set, this value is in fact a tight lower bound. If the polling server has the highest priority, we may observe that $\rho^*_S(x) = \rho_S(x) = x$.

The following theorem presents a tight upper bound for the worst-case response time of a given aperiodic task using a polling server $\tau_S = (P_S, C_S, T_S)$, together with a synchronous periodic task set.

Most research papers on aperiodic tasks only consider soft deadlines, so that we did not find any similar result for such an upper bound. While we derived it in the context of a garbage collector, this result could be useful in other aperiodic task analyses.

Theorem 4.11 *Let $\Phi = (\Gamma, \Pi, \mathcal{L}, \mathcal{R})$ be a schedulable uniprocessor system with a polling server τ_S, then, for an aperiodic task τ_{ap}, if $W(x)$ is an upper bound for $\rho_S(x)$ and $B(x)$ is a lower bound for $\rho_S^*(x)$ for any $0 < x \leq C_S$, we have*

$$\mathcal{R}_{ap}^{wc} \leq \left\lceil \frac{C_{ap}}{C_S} \right\rceil T_S + \max_{0 \leq \varphi < C_S} \{W(r + \left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil C_S - \varphi) - \left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil T_S - B(C_S - \varphi)\} \tag{1}$$

with $r \stackrel{\text{def}}{=} C_{ap} - \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right) C_S$.

Proof: We need to prove that the bound holds for any instant when t^{arriv} could occur. First, it should be clear that the worst cases must arrive when all the tasks (with priorities higher than the aperiodic server) have been started and are scheduled periodically; hence we may only consider the periodic (steady state) part of the schedule. We shall distinguish two situations, depending on the remaining capacity of the server when the aperiodic task arrives:

1. If the server capacity is *full* ($\kappa(t^{\text{arriv}}) = C_S$), the worst-case scenario is when t^{arriv} corresponds with the beginning of the aperiodic server period (i.e., $t^{\text{arriv}} = kT_S$ for some $k \in \mathbb{N}$) because delaying t^{arriv} such that $\kappa(t^{\text{arriv}}) = C_S$ can only reduce its response time.

   Therefore, at most $\left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right)$ *full periods* are necessary to execute the first $\left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right) C_S$ units of computation of τ_{ap}. After that, the time necessary to execute the remaining $r \stackrel{\text{def}}{=} C_{ap} - \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right) C_S$ units of computation is at most $\rho_S(r)$, with $0 < r \leq C_{ap}$; it may be observed that $r = C_{ap} \bmod C_S$ if $C_{ap} \bmod C_S \neq 0$, C_S otherwise.

   Therefore, the worst-case response time in this situation is bounded by

   $$\left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right) T_S + \rho_S(r) \tag{2}$$

2. if the server capacity is *not full* ($\kappa(t^{\text{arriv}}) < C_S$), we denote this amount of capacity by ε; thus $\kappa(t^{\text{arriv}}) = \varepsilon$ and $0 \leq \varepsilon < C_S$.

   We first suppose that $C_{ap} > \varepsilon$. This situation has two subcases:

   - $0 \leq \varepsilon < r$: The *earliest* t^{arriv} can occur within one server period such that $\kappa(t^{\text{arriv}}) = \varepsilon$ is $\rho_S^*(C_S - \varepsilon)$ time units after the beginning of the server period. This is justified by the fact that $\rho_S^*(C_S - \varepsilon)$ is the minimum amount of time it takes for the server to reduce its capacity to ε.

     Therefore, the *longest* time between t^{arriv} and the end of the server period such that $\kappa(t^{\text{arriv}}) = \varepsilon$ is $T_S - \rho_S^*(C_S - \varepsilon)$. During that period, ε computation units of τ_{ap} are executed (possibly for other tasks) or lost.

Then, there are $C_{ap} - \varepsilon$ computation units left to execute. The rest of the execution takes the same time as an aperiodic task in situation 1, where t^{arriv} corresponds to the beginning of a server period. We can use the bound from the first situation by replacing C_{ap} by $C_{ap} - \varepsilon$ and r by $C_{ap} - \varepsilon - \left(\left\lceil \frac{C_{ap}-\varepsilon}{C_S} \right\rceil - 1\right) C_S = r - \varepsilon$; indeed, since $0 \leq \varepsilon < r$, $\frac{C_{ap}-\varepsilon}{C_S} > \frac{C_{ap}-r}{C_S} = \left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1$, so that $\left\lceil \frac{C_{ap}-\varepsilon}{C_S} \right\rceil = \left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1 + 1 = \left\lceil \frac{C_{ap}}{C_S} \right\rceil$. Hence, the execution of these $C_{ap} - \varepsilon$ computation units takes at most $\left(\left\lceil \frac{C_{ap}-\varepsilon}{C_S} \right\rceil - 1\right) T_S + \rho_S(r - \varepsilon) = \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right) T_S + \rho_S(r - \varepsilon)$ time units, and the response time of the aperiodic server in this case is at most

$$T_S - \rho_S^*(C_S - \varepsilon) + \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1\right) T_S + \rho_S(r - \varepsilon)$$

$$= \left\lceil \frac{C_{ap}}{C_S} \right\rceil T_S + \rho_S(r - \varepsilon) - \rho_S^*(C_S - \varepsilon)\,.$$

Notice that this bound may be too pessimistic, because the best case ρ^* does not necessarily occur $\left\lceil \frac{C_{ap}-\varepsilon}{C_S} \right\rceil$ periods before the worst case ρ.

Thus, when $\kappa(t^{\text{arriv}}) < r$, the worst-case response time is bounded by

$$\left\lceil \frac{C_{ap}}{C_S} \right\rceil T_S + \max_{0 \leq \varphi < r} \left\{ \rho_S(r - \varphi) - \rho_S^*(C_S - \varphi) \right\}. \tag{3}$$

- $r \leq \varepsilon < C_S$: The *earliest* t^{arriv} can occur within a server period such that $\kappa(t^{\text{arriv}}) = \varepsilon$ is $\rho_S^*(C_S - \varepsilon)$ time units after the beginning of the server period. Therefore, the *longest* time between t^{arriv} and the end of the server period such that $\kappa(t^{\text{arriv}}) = \varepsilon$ is $T_S - \rho_S^*(C_S - \varepsilon)$ time units. During that period, ε computation units of τ_{ap} are executed (for the aperiodic task).

  Then, there are $C_{ap} - \varepsilon$ computation units left to execute. Once again, the rest of the execution takes the same time as an aperiodic task with t^{arriv} corresponding to the end of the server period with its worst-case computation time equal to $C_{ap} - \varepsilon$. Therefore, we can use the bound from the first situation for this extra waiting time:

$$\left(\left\lceil \frac{C_{ap} - \varepsilon}{C_S} \right\rceil - 1\right) T_S + \rho_S\left(C_{ap} - \varepsilon - \left(\left\lceil \frac{C_{ap} - \varepsilon}{C_S} \right\rceil - 1\right) C_S\right)$$

$$= \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 2\right) T_S + \rho_S\left(C_{ap} - \varepsilon - \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 2\right) C_S\right)$$

(because, as $\varepsilon \geq r$, $\left\lceil \frac{C_{ap}-\varepsilon}{C_S} \right\rceil = \left\lceil \frac{C_{ap}}{C_S} \right\rceil - 1$)

$$= \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil - 2\right) T_S + \rho_S(r + C_S - \varepsilon)\,.$$

Thus, when $r \leq \kappa(t^{\text{arriv}}) < C_S$, the worst-case response time is bounded by

$$\left\lceil \frac{C_{ap}}{C_S} \right\rceil T_S + \max_{r \leq \varphi < C_S} \{\rho_S(r + C_S - \varphi) - T_S - \rho_S^*(C_S - \varphi)\} \quad (4)$$

We now look at the specific case when $C_{ap} \leq \varepsilon < C_S$. Clearly, the completion time t^{end} occurs before or at the end of the server period because there is enough capacity at t^{arriv} to execute the entire aperiodic task. More precisely, the response time of the aperiodic task in this case is bounded by the longest time between t^{arriv} and the time to service $C_S - \varepsilon + C_{ap}$ time units of work:

$$\rho_S(C_S - \varepsilon + C_{ap}) - \rho_S^*(C_S - \varepsilon)\,. \quad (5)$$

We now show that the bound in (1) is an upper bound for the formulas (2), (3), (4) and (5). First, we may notice that this bound is trivially an upper bound for (3), since in that case $-1 < \frac{\varphi - r + 1}{C_S} \leq 0$ and $\left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil = 0$; similarly, this bound is trivially an upper bound for (4), since in that case $0 < \frac{\varphi - r + 1}{C_S} \leq 1$ and $\left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil = 1$; moreover, $(2) = \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil \right) T_S + \rho_S(r) - T_S \leq \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil \right) T_S + \rho_S(r) - \rho_S^*(C_S) = \left(\left\lceil \frac{C_{ap}}{C_S} \right\rceil \right) T_S + \rho_S(r - 0) - \rho_S^*(C_S - 0) \leq (3)$, since $\rho_S^*(C_S) \leq T_S$ and 0 is the first φ in $0 \leq \varphi < r$; finally, we may observe that, when $0 < C_{ap} \leq \varepsilon < C_S$, we have $r = C_{ap}$ and $\left\lceil \frac{C_{ap}}{C_S} \right\rceil T_S = T_S$, so that $(5) = \left\lceil \frac{C_{ap}}{C_S} \right\rceil T_S + \Big(\rho_S(r + C_S - \varepsilon) - T_S - \rho_S^*(C_S - \varepsilon) \Big) \leq (4)$ since $r \leq \varepsilon \leq C_S$ and ε is in the right range for φ. □

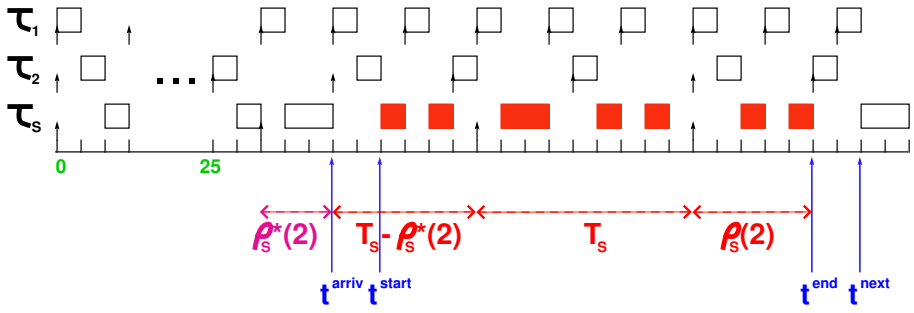
For example, Figure 2 shows the response time of an aperiodic task execution with $C_{ap} = 8$ using a polling server τ_S. Since in this case $r = C_S = 4$, in the formulas above $\varphi < r$, $\left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil = 0$ and only the bound (3) arises. Table 1 summarizes the needed characteristics for the possible values of φ; this table has been obtained by a mere observation of the scheduling of the system during an hyperperiod $[0, 45)$.

As it occurs, $\max_{0 \leq \varphi < C_S} \{W(r + \left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil C_S - \varphi) - \left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil T_S - B(C_S - \varphi)\}$ reaches its maximum when $\varphi = 1$ or 2.

In Figure 2, the case $\kappa(t^{\text{arriv}}) = 2$ is shown. The earliest t^{arriv} can occur is $\rho_S^*(2) = 3$ time units after the beginning of the server period. Thus, the time between t^{arriv} and the end of the server period is at most $T_S - \rho_S^*(2) = 6$ time units. Then, one full period (9 time units) is necessary to execute the next four computation units. Finally, the two remaining computation units are executed

Figure 2: Example of the execution of an aperiodic task with $C_{ap} = 8$.

Γ	C	T	D	O	Π
τ_1	1	3	3	0	0
τ_2	1	5	5	0	1
τ_S	4	9			2



in $\rho_S(2) = 5$ time units. Hence, the response time of this execution is 20 time units, which exactly corresponds to the evaluation of expression (1).

Table 1: Analysing the hyperperiod $[0, 45)$.

φ	$\left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil$	$\rho_S(r + \left\lceil \frac{\varphi - r + 1}{C_S} \right\rceil C_S - \varphi)$	$\rho^*_S(C_S - \varphi)$	$\rho_S(r-\varphi) - \rho^*_S(C_S - \varphi)$
0	0	9	7	2
1	0	8	4	4
2	0	5	3	4
3	0	3	1	2

4.4 Minimum memory requirement

To verify the second condition of the schedulability definition, we provide an amount of memory guaranteeing that the heap is not submerged in the worst-case scenario. This amount depends on the maximum amount $\mathcal{L}$ of live memory in the system and on the maximum amount of memory allocated by the tasks during the longest time between two flips. It also depends on the collection time $C_{ap} = C_{GC}$, which depends on $\mathcal{L}$, on the size of the root set $\mathcal{R}$, and

possibly on other parameters: we shall assume that we know the worst-case execution time of the garbage collector in our context.

Theorem 4.12 *For a system $\Phi = (\Gamma, \Pi, \mathcal{L}, \mathcal{R})$ with a real-time copying collector τ_{GC} serviced by a polling server τ_S, the minimum size M of the heap such that the system is guaranteed schedulable is upper bounded by*

$$2\left(\mathcal{L} + \sum_{\{i|P_i<P_S\}} \left\lceil \frac{\mathcal{R}_{GC}^{wc}-1}{T_i} \right\rceil A_i + \sum_{\{i|P_i>P_S\}} \left(\left\lceil \frac{\mathcal{R}_{GC}^{wc}-2}{T_i} \right\rceil + 1 \right) A_i \right)$$

Proof: Each semi-space must be large enough to hold the memory copied by the collector during one collection cycle plus the memory allocated by the tasks during $[t^{\text{start}}, t^{\text{next}})$, because a flip is performed at t^{start} and the earliest following flip is at t^{next}.

The collector only copies the live objects. The amount of live memory is at most $\mathcal{L}$.

From Theorem 4.7, the bound $\mathcal{R}_{GC}^{wc}$ also holds for the worst-case period between t^{start} and t^{next}. Therefore, we must count the maximum amount of memory allocated by each task during $[t^{\text{start}}, t^{\text{next}})$, which is at most $\mathcal{R}_{GC}^{wc}$. We should first notice that even if, during the period $[t^{\text{start}}, t^{\text{next}})$, there is only a part of a request executed for a task τ_i, A_i must be fully counted for it because we do not have any information on when the memory is allocated within a request.

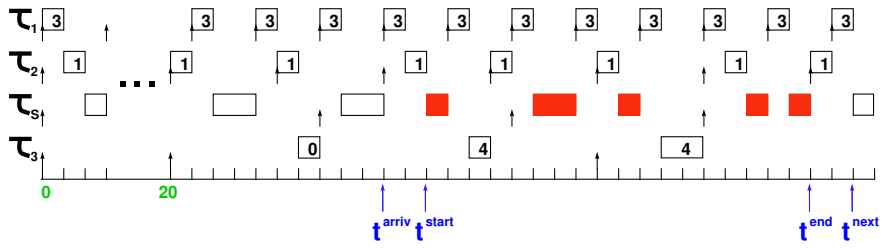
For a periodic task τ_i with priority lower than P_S, we may see that there are at most $\left\lceil \frac{\mathcal{R}_{GC}^{wc}-2}{T_i} \right\rceil + 1$ requests and/or fractions of requests of τ_i executed during $\mathcal{R}_{GC}^{wc}$ time units. Indeed, the worst case occurs when there is a single slot used to terminate a period for τ_i between $t^{\text{start}}+1$ and $t^{\text{start}}+2$; at $t^{\text{start}}+2$, a new cycle starts and there are at most $\mathcal{R}_{GC}^{wc}-2$ time units left till t^{next}, during which $\left\lceil \frac{\mathcal{R}_{GC}^{wc}-2}{T_i} \right\rceil$ cycles of τ_i are completed or started (for the last one); hence the formula for that case.

Now, we count the maximum amount of memory allocated by tasks with higher priorities than P_S. t^{start} cannot occur when a request of task τ_i ($P_i < P_S$) is not completed. In the same way, t^{next} cannot occur when a request of task τ_i is not completed. Therefore, the worst case occurs when a new cycle for τ_i starts at $t^{\text{start}}+1$, and the maximum number of requests of τ_i completed during $[t^{\text{start}}+1, t^{\text{next}})$ is $\left\lceil \frac{\mathcal{R}_{GC}^{wc}-1}{T_i} \right\rceil$; hence the formula for that case. □

Figure 3 illustrates this theorem with an example that actually reaches the bound. Assume that the garbage collector flips the two semi-spaces at time

Figure 3: Example of the memory requirement with $C_{GC} = 6$.

Γ	C	T	D	O	A	Π
τ_1	1	3	3	0	3	0
τ_2	1	5	5	0	1	1
τ_3	2	20	20	0	4	3
τ_S	3	9				2



$t^{\text{start}} = 32$ and copies $\mathcal{L}$ memory units during that collection cycle. During that cycle $[t^{\text{start}}, t^{\text{next}}) = \mathcal{R}_{GC}^{wc} = 20$, the periodic tasks allocate $\lceil \frac{19}{3} \rceil 3 + \lceil \frac{19}{5} \rceil 1 + (\lceil \frac{18}{20} \rceil + 1)4 = 33$ memory units in the heap (the amount allocated for each execution unit is provided in the figure).

5 Conclusion

We have shown a predictable method to schedule a real-time copying garbage collector in a real-time system constituted of static-priority periodic tasks. This method considers the garbage collector as an aperiodic task and executes it through a polling server.

In addition to a time constraints schedulability test, the schedulability analysis of the real-time system model requires the analysis of the minimum heap size required to guarantee sufficient memory for the periodic tasks.

We have derived a bound for the worst-case response time of an aperiodic task for the polling server with any parameters. This bound allows us to determine a bound on the longest garbage collection cycle, necessary to bound the heap size required.

The schedulability analysis of this system is possible under the assumption that we can compute or bound the memory utilization information included in the system model: $\mathcal{L}$, $\mathcal{R}$, $\{A_1, \ldots, A_n\}$ and $\{C_1, \ldots, C_n\}$.

Our contribution can be summarize as follows:

- we present a formalization of the *joint* scheduling problem which consists to schedule hard periodic tasks and a garbage collection together;
- we propose a schedulability test for a more specific case: scheduling static-priority periodic tasks with a copying collector;
- we characterize the worst case response time of an aperiodic task served by a polling server.

Extensions of the present analysis could encompass the case where many polling servers, including a background one, are used to service the garbage collector, and/or where many processors share a common heap memory. The analysis could also be extended to continuous time systems.

Acknowledgments

Plenty of thanks go to Dr Philip Holman from the University of North Carolina for carefully reading a draft of this paper and for his helpful suggestions.

References

[1] BAKER, H. G. List processing in real time on a serial computer. *Communications of the ACM 27*, 4 (April 1978), 280–294.

[2] BOLLELLA, G., AND GOSLING, J. The real-time specification for Java. *Computer 33*, 6 (2000), 47–54.

[3] BROOKS, R. A. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on Lisp and Functional Programming* (Austin, TX, August 1984), ACM Press, pp. 256–262.

[4] GOOSSENS, J. *Scheduling of Hard Real-Time Periodic Systems with Various Kinds of Deadline and Offset Constraints*. PhD thesis, Université Libre de Bruxelles, 1999.

[5] HENRIKSSON, R. Scheduling real time garbage collection. In *Proceedings of NWPER'94* (1994).

[6] HENRIKSSON, R. Adaptive scheduling of incremental copying garbage collection for interactive applications. Tech. Rep. 96–174, Lund University, Sweden, 1996.

[7] HENRIKSSON, R. Predictable automatic memory management for embedded systems. In *OOPSLA '97 Workshop on Garbage Collection and Memory Management* (October 1997), P. Dickman and P. R. Wilson, Eds.

[8] HENRIKSSON, R. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund Institute of Technology, 1998.

[9] JONES, R., AND LINS, R. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, 1996.

[10] KIM, T., CHANG, N., KIM, N., AND SHIN, H. Scheduling garbage collector for embedded real-time systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop in Languages, Compilers and Tools for Embedded Systems* (May 1999), pp. 55–64.

[11] KIM, T., CHANG, N., AND SHIN, H. Bounding worst case garbage collection time for embedded real-time systems. In *Proceedings of The 6th IEEE Real-Time Technology and Applications Symposium* (June 2000), pp. 46–55.

[12] KIM, T., CHANG, N., AND SHIN, H. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software 58*, 3 (September 2001), 247–260.

[13] LEUNG, J. Y.-T., AND WHITEHEAD, J. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation 2* (1982), 237–250.

[14] LIU, C. L., AND LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM 20*, 1 (1973), 46–61.

[15] MAGNUSSON, B., AND HENRIKSSON, R. Garbage collection for control systems. In *Proceedings of International Workshop on Memory Management* (Lund University, Sweden, September 1995), H. Baker, Ed., vol. 986 of *Lecture Notes in Computer Science*, Springer-Verlag.

[16] NILSEN, K. High-level dynamic memory management for object oriented real-time systems. In *Workshop on Object-Oriented Real-Time Systems* (San Antonio, Tx., October 1995).

[17] NILSEN, K. D., AND SCHMIDT, W. J. Hardware-assisted general-purpose garbage collection for hard real-time systems. Tech. Rep. ISU TR92-15, Iowa State University, Department of Computer Science, October 1992.

[18] REDELL, O., AND SANFRIDSON, M. Exact best-case response time analysis of fixed priority scheduled tasks. In *Proceedings of 14th Euromicro Conference on Real-Time Systems* (June 2002), pp. 165–172.

[19] ROBERTZ, S. G. Applying priorities to memory allocation. In *Proceedings of the 2002 International Symposium on Memory Management* (Berlin, Germany, June 2002).

[20] ROBERTZ, S. G., AND HENRIKSSON, R. Time-triggered garbage collection: robust and adaptive real-time gc scheduling for embedded systems. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems* (June 2003), ACM Press, pp. 93–102.

[21] SIEBERT, F. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In *Proceedings of the first international symposium on Memory management* (1998), ACM Press, pp. 130–137.

[22] SPRUNT, B. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, August 1990.

[23] TINDELL, K. Using offset information to analyse static priority preemptively scheduled task sets. Tech. Rep. YCS 182, University of York, Department of Computer Science, August 1992.

[24] WILSON, P. R. Uniprocessor garbage collection techniques. In *Proc. Int. Workshop on Memory Management* (Saint-Malo, France, 1992), no. 637 in Lecture Notes in Computer Science, Springer-Verlag.