Joint Scheduling of Garbage Collector and Hard Real-Time Tasks for Embedded Applications

Taehyoun Kim,* Naehyuck Chang and Heonshik Shin
School of Computer Science and Engineering, Seoul National University
Seoul 151-742, Korea

Abstract

Programs with complex data structures often require dynamic memory management based on automatic memory reclamation (garbage collection). A major problem in adopting garbage collection for embedded real-time systems is that it often causes unpredictable pauses and that, as a result of such delays, hard real-time tasks may miss their deadlines. In this paper, we propose a new real-time garbage collection technique for embedded applications. In our approach, the system jointly schedules garbage collector and hard real-time tasks using one of the aperiodic server approaches. Our study focuses on reducing memory requirements while guaranteeing the deadlines of hard real-time tasks. To achieve this objective, we model garbage collection requests as aperiodic hard real-time tasks, and schedule them using the *sporadic server*. We also present an effective live-memory analysis to bound the worst-case garbage collection time. Performance analysis shows that the proposed approach considerably reduces the worst-case memory reservation compared with a background policy. The analytic results are verified by simulation based on trace-driven data.

Keywords: garbage collection, embedded systems, scheduling, sporadic server, live-memory analysis

1 Introduction

As information technology permeates into every aspect of our society and industry, distributed real-time computing systems have been implemented in many forms, such as component-based software, software agents, and Java virtual machines with applets. These techniques are characterized by objects, mobility, autonomy, and dynamic behavior. Execution of programs with complex data structures often requires dynamic memory management of a heap to utilize memory efficiently by reclaiming unused space. We usually hold the system, not the programmer, responsible for the memory management so as to achieve improved productivity, robustness, and program integrity. Central to this automatic memory reclamation is the garbage collection process. A garbage collector identifies the data items that will never be used again and then recycles their space for reuse at the system level. In spite of its advantages, garbage collection has not been widely used in embedded real-time applications. For example, few embedded real-time systems have been written in Java™ language, even though it has been popular for years. This is partly because garbage collection may cause the response time of an application to become unpredictable.

For predictable execution of a real-time application, the garbage collector itself must run in *real-time* mode. We summarize and extend the requirements for a real-time garbage collector that were presented by Wilson (Wilson, 1994):

- *Concurrent or incremental execution.* In order to avoid intolerable pauses incurred by stop-and-go reclamation, a real-time garbage collector often interleaves its execution with the execution of an application. Depending on the implementation, the garbage collector runs either concurrently with mutators[1] or incrementally in the context of mutator execution. The system needs to remain consistent or safe while the garbage

*Corresponding author. Tel.: +82-2-880-7298; fax.: +82-2-874-3104; e-mail: thkim@cslab.snu.ac.kr

[1] In incremental collectors, applications may *mutate* the reachability of heap data structure unnoticed by the garbage collector. For this reason, the application is often referred as a "mutator". This paper uses the term "mutator" for the tasks that manipulate dynamically allocated heap objects.

collector identifies all the live objects and processes them. To preserve the consistency of a heap, a real-time collector must have mutators report on any changes that they have made to the liveness of heap objects.

- *Bounded interference with mutator execution.* Garbage collector must not interfere with the schedulability of hard real-time mutators. For this purpose, we need to keep the basic memory operations, such as memory allocation and pointer modification, short and bounded. We must also aim to minimize the synchronization overhead between garbage collector and mutators.

- *Bounded execution time.* In real-time systems, the worst-case behavior of the garbage collector should also be predictable. Real-time systems with garbage collection must meet the deadlines of hard real-time mutators while preventing the application from running out of memory. Unbounded garbage collection times may cause either the deadlines of hard real-time mutators to be missed, or memory shortage.

Considering these properties, we develop a new approach for real-time garbage collection. Our approach integrates garbage collection with task scheduling. For hard real-time periodic tasks, the objective of our approach is to guarantee that deadlines are always met. Another objective is to reduce the memory requirement since heap memory is usually a scarce resource in embedded systems.

To achieve these objectives, our approach first models garbage collection requests as hard real-time aperiodic tasks. The sporadic server (SS) (Sprunt et al., 1989) schedules them with a preset capacity. Our approach determines the server capacity using the worst-case response time analysis for the SS (Ghazalie and Baker, 1995; Bernat and Burns, 1999). In this way, we can guarantee the schedulability of hard real-time mutators. We also provide a worst-case live-memory analysis to bound the worst-case garbage collection time. Live-memory analysis provides a safe and effective bound of live memory. The worst-case response time of the garbage collector and the worst-case live memory requirement are the dominant factors in determining the total memory requirement, which is minimized by our approach. We choose a semi-space copying collector (Baker, 1978; Brooks, 1984; Kim et al., 2000) for the discussion in this paper because it automatically compacts the live-object area, and thus simplifies the estimation of garbage collection time. From now on, the term *garbage collector* will indicate a real-time copying garbage collector unless specified otherwise.

The rest of this paper is organized as follows. In Section 2, we review related work on real-time garbage collection. Section 3 formulates the problem addressed in this paper. The scheduling algorithm for real-time garbage collection is introduced in Section 4; this section also presents the live-memory analysis based on the scheduling approach. We evaluate the effectiveness of the proposed approach in Section 5. Section 6 briefly discusses the design consideration for embedded real-time systems with garbage collection. Section 7 concludes this paper.

2 Related Work

Since basic garbage collection inherently works in a stop-and-go fashion, the pauses that it introduces may become intolerable for applications that require a bounded response time. Incremental and concurrent garbage collection algorithms (Baker, 1978; Brooks, 1984; Yuasa, 1990) have been proposed to distribute and hide the garbage collection pause time throughout the execution of mutators. Although they can reduce intermediate pause delays, few of them bound the worst-case garbage collection delay because they do not seriously address the timing properties of real-time mutators, such as deadline and periodic execution. Therefore, it is not feasible to apply simple incremental garbage collection to hard real-time systems because the deadline misses of hard real-time tasks may result in catastrophic system failure.

Recently, Henriksson (Henriksson, 1998) proposed a scheduling-based approach for real-time garbage collection. This approach classifies tasks into three priority levels. The garbage collector has medium priority and runs in the *background* while high-priority tasks are idle. This approach is meaningful in that it addresses the schedulability of hard real-time mutators; however, it demands that excessive memory is reserved for hard real-time tasks because of the long response time of the garbage collector (Kim et al., 1999).

For the safety of real-time collectors, synchronization between a garbage collector and mutators is also required. Most approaches solve this problem using tricolor abstraction (Dijkstra et al., 1978) and barrier methods (Wilson, 1994). In general, barrier methods are executed by the mutator's activity and their execution must be

indivisible. Traditional barrier methods for copying collectors (Baker, 1978; Brooks, 1984) are accompanied by asynchronous evacuation and additional reference traversal. These overheads may impose intolerable interference with the schedulability of hard real-time mutators. As mentioned in Section 1, a real-time garbage collector should bound and minimize this interference. A real-time copying garbage collector presented in our previous work is able to fulfill this requirement. Hence, we base our discussion on the real-time copying collector presented in (Kim et al., 2000).

We must also be able to estimate the worst-case garbage collection time. The copying garbage collection process, discussed in this paper, consists of four major components: reference traversal, object evacuation, synchronization between garbage collector and mutators, and memory initialization. The overhead of these operations greatly depends on the amount of live memory. Although there have been many studies on the analysis of pointers (Rugina and Rinard, 1999; Yong et al., 1999), they do not focus on worst-case live-memory analysis. A recent study by Persson (Persson, 1999) targets worst-case memory consumption and worst-case live-memory analysis, based on annotation. Although this study provides useful information for those metrics, Persson's approach requires modification of language features and does not consider the periodic execution of mutators.

3 Problem Formulation

3.1 System Model and Assumptions

We consider a real-time system with a set of n periodic priority-ordered mutator tasks, $\mathcal{M} = \{\mathcal{M}_1, \ldots, \mathcal{M}_n\}$ where $\mathcal{M}_n$ is the lowest-priority task. All the tasks follow a fixed-priority scheduling such as rate monotonic scheduling (Liu and Layland, 1973). Our task model includes an additional property, the memory allocation requirement of $\mathcal{M}_i$. A mutator $\mathcal{M}_i$ is characterized by a tuple $\mathcal{M}_i = (C_i, T_i, D_i, A_i)$ (see Table 1 for notations). Further discussion will be based on the following assumptions:

- *Assumption 1:* There are no aperiodic mutator tasks.
- *Assumption 2:* The context switching and task scheduling overhead are negligible.
- *Assumption 3:* There are no precedence relations among the tasks in $\mathcal{M}$. Although many real-time systems actually have precedence relations among tasks, precedence constraints can easily be removed by partitioning tasks into independent sub-tasks or assigning the priorities of tasks properly. For this reason, we make this assumption without loss of generality.
- *Assumption 4:* Any task can be instantly preempted by a higher priority task (i.e., there are no blocking factors).
- *Assumption 5:* C_i, T_i, D_i, and A_i are known *a priori*. The j^{th} instance of $\mathcal{M}_i$, denoted by $\mathcal{M}_{i,j}$, is released and ready at time $(j-1)T_i$. The release jitter of each task is assumed to be zero.

Although the estimation of A_i is generally an application-specific problem, A_i can be specified by an application programmer or given by a pre-runtime trace-driven analysis.

The target system is designed to adopt dynamic memory allocation with no virtual memory. Figure 1 illustrates our memory model. The garbage collector reclaims unused memory at the rate of $f_r(k,t)$ while a mutator $\mathcal{M}_i$ consumes available memory at the rate of $f_c(\mathcal{M}_i,k,t)$. The garbage collector distinguishes the memory objects that are no longer in use (*garbage*) from live objects and reclaims their space for future use. It runs either as a separate real-time task or in the context of mutator execution, depending on the implementation. This paper treats each garbage collection request as a separate task.

We now consider the real-time garbage collection requests, $\{\mathcal{G}_k,\ k \geq 1\}$. The release time and completion time of $\mathcal{G}_k$ are denoted by t_s^k and t_e^k, respectively. A garbage collection request $\mathcal{G}_k$ is released when the amount of available memory becomes less than a certain threshold (i.e., the cumulative memory consumption exceeds the amount of free memory). The cumulative memory consumption $m_c(\mathcal{M}_i,k,t)$, which is defined for the interval $[t_s^k, t_s^{k+1})$, is a monotonically increasing function. The memory consumption function $f_c(\mathcal{M}_i,k,t)$ can be of various types depending on the scheduling strategy and the nature of an application. However, we can derive upper bounds of $m_c(\mathcal{M}_i,k,t)$ and $m_c(k,t)$ from the worst-case memory requirement of the mutator task $\mathcal{M}_i$, which is the product of

Table 1: Notations

Symbol	Description
n	Number of periodic tasks
$\mathcal{M}_i$, $\mathcal{M}_{i,j}$	Periodic mutator task i and its j^{th} instance
C_i, T_i, D_i, R_i	worst-case execution time, period, deadline, and response time of $\mathcal{M}_i$
A_i	Maximum amount of memory allocated by $\mathcal{M}_i$ during T_i
γ_i	Portion of A_i that is live when $\mathcal{M}_i$ is *inactive*, $0 \leq \gamma_i \leq 1$
C_{GC}, R_{GC}	worst-case execution time and response time of garbage collector
$\mathcal{R}_\mathcal{G}$	worst-case reservation interval
π_i	worst-case number of $\mathcal{M}_{ij}$s during $\mathcal{R}_\mathcal{G}$
$\mathcal{G}_k$	k^{th} garbage collection request
t_s^k, t_e^k	Release time and completion time of $\mathcal{G}_k$ ($k \geq 1$)
R_k^{gc}, D_k^{gc}	Response time and deadline of $\mathcal{G}_k$
$f_c(\mathcal{M}_i, k, t)$	Memory consumption rate of $\mathcal{M}_i$ at t ($t_s^k \leq t < t_s^{k+1}$)
$f_r(k,t)$	Memory reclamation function at t ($t_s^k \leq t < t_e^k$)
$m_c(\mathcal{M}_i, k, t)$, $m_c(k,t)$	Cumulative memory consumption by $\mathcal{M}_i$ and the sum of $m_c(\mathcal{M}_i, k, t)$ over all $\mathcal{M}$
$L_{k,local}$, $L^*_{k,local}$	Amount of local live memory processed by $\mathcal{G}_k$ and its maximum value
$L^*_{k,glob}$	Maximum amount of global live memory processed by $\mathcal{G}_k$
L_k, L_k^*	Amount of live memory processed by $\mathcal{G}_k$ and its maximum value
M_{resv}	Memory reservation for hard real-time tasks
M	System memory requirement (the size of *semispace* for the copying collector is $\frac{M}{2}$)
H	Hyperperiod of all $\mathcal{M}_i$s (=$LCM(T_1, \ldots, T_n)$)
T_s, C_s	Period and capacity of the sporadic server

A_i and the worst-case number of terms $\mathcal{M}_{i,j}$ during a given time interval. Then, the cumulative memory consumption at t', $t_s^k \leq t' < t_s^{k+1}$, is bounded by the following equation.

$$m_c(k,t') \triangleq \sum_{i=1}^{n} m_c(\mathcal{M}_i, k, t') < \sum_{i=1}^{n} \left\{ \left(\left\lceil \frac{t'}{T_i} \right\rceil - \left\lfloor \frac{t_s^k}{T_i} \right\rfloor \right) A_i \right\}. \quad (1)$$

The amount of available memory depends on the reclamation rate of the garbage collector. While mark-and-sweep collectors progressively reclaim the heap area, copying collectors reclaim half the total memory at flip time. Actually, the amount of heap memory reproduced by $\mathcal{G}_k$ depends on the total heap memory size M and the size of the live memory L_k; in the case of copying collectors, it is bounded by $(\frac{M}{2} - L_k)$ (Kim et al., 1999).

3.2 Timing Constraint on the Garbage Collector

We now consider the timing properties of garbage collection requests. A garbage collection request $\mathcal{G}_k$ is characterized as follows.

- $\mathcal{G}_k$ is an aperiodic request because its release time is not known *a priori*. The release time of $\mathcal{G}_k$ depends on the cumulative memory consumption and the amount of available memory.
- $\mathcal{G}_k$ has a hard deadline. The k^{th} garbage collection request $\mathcal{G}_k$ must be completed before $\mathcal{G}_{k+1}$ is released. Let us denote the deadline of $\mathcal{G}_k$ by $D_k^{gc} = t_s^{k+1}$. Then, the condition $t_e^k \leq D_k^{gc}$ must always be satisfied. Suppose that the amount of available memory becomes less than a certain threshold and that the preceding garbage collection request has not yet been completed. In this case, neither the garbage collector nor the mutators can continue their execution. This is because the heap memory is fully occupied by the live objects evacuated by the garbage collector and new objects allocated by the mutators. Figure 2 illustrates this situation. In Figure 2, if the cumulative memory allocation, represented as the dotted line, causes another garbage collection process to be released, the garbage collector can no longer process live objects which have not been evacuated yet (objects A, B, and C in Figure 2).

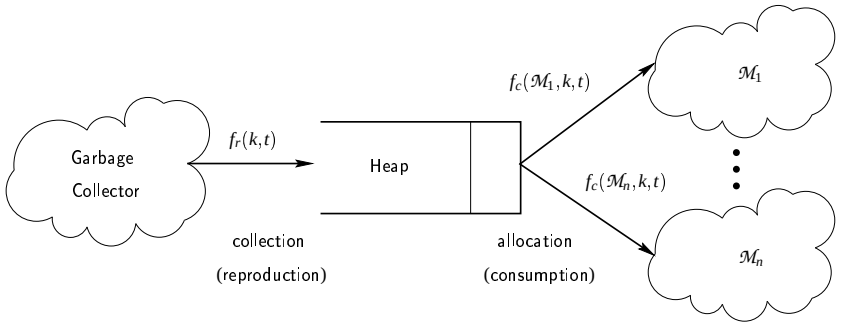


Figure 1: Memory model.

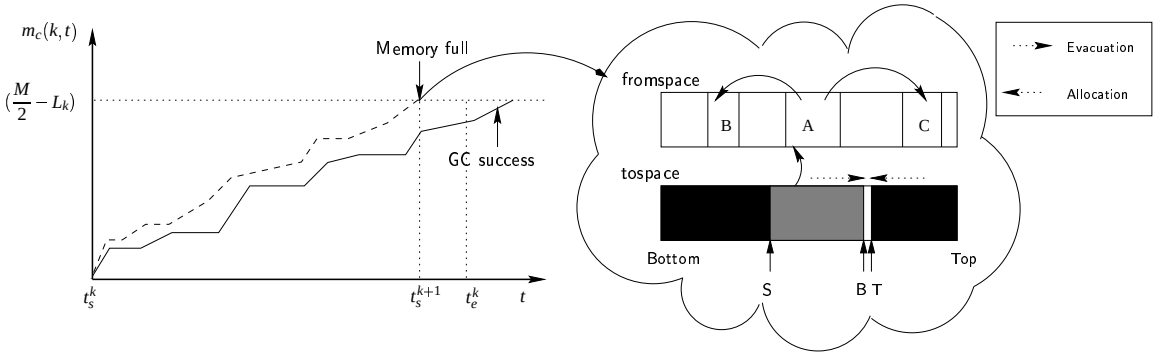


Figure 2: Timing constraints on $\mathcal{G}_k$

.

The principal objective of traditional aperiodic scheduling has been to minimize the average response time of aperiodic tasks while meeting the deadlines of hard real-time periodic tasks. On the other hand, relatively less effort has been applied to the scheduling of hard real-time aperiodic tasks. Ramos-Thuel et. al. proposed an algorithm for the scheduling of hard real-time aperiodic tasks (Ramos-Thuel and Lehoczky, 1993). This algorithm assumes that the worst-case computation time and the deadline of an aperiodic task are known when the task arrives. Unfortunately, the worst-case computation time and the deadline of each $\mathcal{G}_k$ can hardly be known on arrival because the computation time of each $\mathcal{G}_k$ is dependent on the amount of live memory L_k, and D_k^{gc} is determined by the memory requirement of mutator tasks. For this reason, it is necessary to consider carefully the worst-case scenario in order to prevent memory shortage while meeting hard deadlines.

The worst-case scenario occurs when $\mathcal{G}_k$ should process L_k^* ($\geq L_k$ for $k \geq 1$) live bytes and the available CPU bandwidth for garbage collection at the time when a garbage collection request is released has its smallest non-negative value. In this case, the amount of free memory is minimized while the response time of the garbage collector is maximized. Two problems now arise. First, how can we determine the worst-case garbage collection time? Because it greatly depends on the worst-case live memory L_k^*, we need an effective live-memory prediction technique. Section 4.4 addresses this problem. Second, how can we minimize total memory requirement while meeting the deadline of each $\mathcal{M}_i$ and $\mathcal{G}_k$? In Section 4.3, we propose an efficient algorithm for scheduling garbage collector and mutators together so as to minimize memory requirement while meeting hard deadlines.

4 Scheduling Garbage Collection and Mutators

We will now describe the scheduling of $\mathcal{G}_k$ in the context of traditional task scheduling. The main objective of our algorithm is to minimize the memory requirement while meeting the hard deadlines of mutators. We consider the

real-time copying collector presented in (Kim et al., 2000) to estimate the worst-case garbage collection time.

4.1 Estimating the Garbage Collection Work

We first provide a brief overview of our garbage collection algorithm. Generally, copying garbage collection begins with the flipping of two semi-spaces and scanning of the root-set. After root-set scanning, the garbage collector scans references in the live objects and evacuates reachable objects until all the live objects are evacuated into the new tospace. In doing so, synchronization between the garbage collector and mutators, and memory initialization is also performed.

To make the above operations faster and more predictable, we have introduced some performance enhancement techniques. To prevent unbounded blocking and traversing of *fake* live objects, our algorithm performs root-set scanning according to the longest-period-first scheme (Kim et al., 1999). In addition, we have introduced a new data structure called update entry (u-entry) to minimize asynchronous evacuations and fake live objects caused by the use of a write-barrier. Instead, the garbage collector performs the clean-up process for u-entries at the end of the collection cycle. Memory initialization is also optimized with hardware support.

The copying garbage collection process consists of the following four components: reference traversal (including root-set and live objects), normal object evacuation, the clean-up process for the active u-entries, and memory initialization. The worst-case garbage collection time C_{GC} is calculated as follows:

$$C_{GC} = c_1 \left(\frac{\mathcal{R} + L_k^*}{sizeof(word)} \right) + c_2 L_k^* + c_3 \mathcal{E} + c_4 M, \tag{2}$$

where $\mathcal{R}$, $\mathcal{E}$, and $sizeof(word)$ denote respectively the maximum size of root-set, the maximum number of active u-entries, and the word size supported by the target system. Although all the coefficients in Eq. (2) rely on the properties of a given task-set and run-time environment, they may be specified by programmer or compiler.

Generally, the values in the stack and the global variables which hold references to heap objects form the root-set. The maximum size of the root-set can be known *a priori*. For example, a modern embedded OS enables programmers to specify the maximum stack size at task creation. The Java language also specifies the maximum stack size and the local variable size for each method. Global variables should be registered as belonging to the root-set by a compiler and their maximum size should be specified by the programmer. Actually, the reference traversal (scanning) overhead depends on the number of reference fields in the scanned object. Our estimate uses the number of words instead of the number of references because, in general, the size of a reference field equals $sizeof(word)$ and thus the number of words becomes the upper bound on the number of fields to be scanned. More accurate estimation would require a well-designed off-line analysis. Coefficients c_1, c_2, c_3, and c_4 are dependent on the underlying run-time environment, such as hardware, OS or compiler support. The implementation should concentrate on reducing the values of coefficients as in traditional incremental approaches. For example, hardware support using the SGRAM (Kim et al., 1999) can greatly reduce c_4.

Eq. (2) indicates that L_k^* exerts a great influence on the worst-case garbage collection time C_{GC}, which determines the CPU utilization of the garbage collector as well as the safe memory requirement. For this reason, we need an effective approach for the estimation of L_k^* to achieve two objectives: schedulability and minimum memory requirement.

4.2 Worst-Case Memory Requirement

The system should have free memory available which is not less than the memory requirement of mutator tasks during the interval $[t_s^k, t_e^k)$. This is because the system breaks down if $\mathcal{G}_{k+1}$ is released before $\mathcal{G}_k$ is completed (i.e., $t_e^k > t_s^{k+1}$, as described in Section 3.2). The system may also break down if there is no CPU bandwidth left for garbage collection at t_s^{k+1} even though the condition $t_e^k < t_s^{k+1}$ holds. To solve this problem, we define *a reservation interval* $\mathcal{R}_\mathcal{G}$ as follows.

Definition 1 *The reservation interval $\mathcal{R}_\mathcal{G}$ represents the worst-case time interval $[t_s^k, t_\gamma)$, where t_γ ($\geq t_e^k$) is the earliest time instant at which the CPU bandwidth for garbage collection becomes available.*

Figure 3 clarifies the definition of the reservation interval. In Figure 3 (a), the garbage collection request that is released at time t, where $t_e^k \leq t < t_\gamma$, cannot be serviced because there is no server capacity available. The reservation interval depends on the CPU bandwidth available for garbage collection and the priority of the garbage collector. For this reason, $\mathcal{R}_\mathcal{G}$ is generally monotonic with respect to the worst-case response time of $\mathcal{G}_k$, R_{GC}.

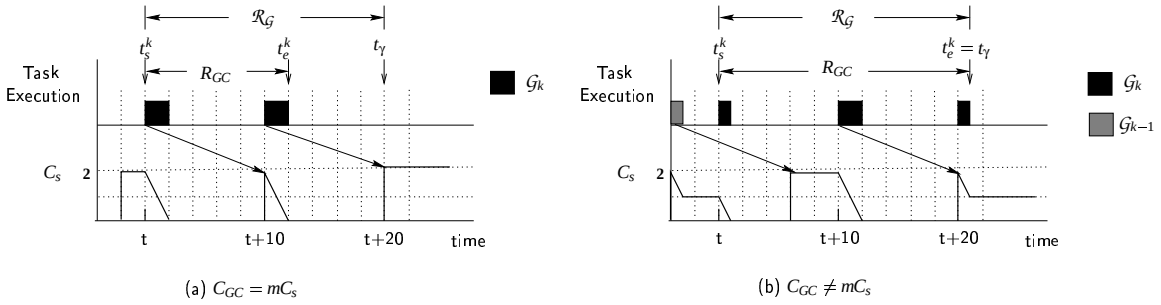


Figure 3: Definition of a reservation interval ($T_s = 10, C_s = 2, C_{GC} = 4$, m is integer)

.

The worst-case memory reservation is the product of the worst-case number of $\mathcal{M}_{i,j}$ during $\mathcal{R}_\mathcal{G}$, denoted by π_i, and A_i. There should also be memory spaces for live objects. As a result, the total memory requirement is twice the sum of the worst-case memory reservation and the worst-case live memory, and is written as:

$$M = 2\left(\sum_{i=1}^{n} \pi_i A_i + L_k^*\right), \tag{3}$$

where n is the number of periodic tasks. The worst-case memory reservation depends on $\mathcal{R}_\mathcal{G}$, which may vary with the scheduling policies adopted. Because $\mathcal{R}_\mathcal{G}$ is monotonic with respect to R_{GC}, we seek a scheduling technique that will minimize the memory reservation by reducing R_{GC}.

4.3 Scheduling using the Sporadic Server

This section introduces a new approach for scheduling the mutators $\mathcal{M}_i$ and the garbage collector $\mathcal{G}_k$. Our approach adopts one of the *aperiodic server* approaches, the sporadic server (SS). The sporadic server is a high-priority task that services aperiodic requests. It preserves its bandwidth (capacity) waiting for the arrival of aperiodic tasks. If an aperiodic task arrives in the meantime it will be executed, provided that server capacity permits; if it cannot finish within one server period, it will resume execution when execution time for the sporadic server is replenished.

There have been several approaches to the problem of jointly scheduling hard real-time periodic tasks (mutators) and aperiodic tasks (garbage collection). The simplest and the least efficient way is to treat the aperiodic tasks as background ones. In this approach, an aperiodic task always has the lowest priority, and it executes only when CPU is idle. A background policy is also the least efficient in terms of memory requirement since it maximizes R_{GC}. A polling server can easily be implemented, but it has a critical drawback. Because aperiodic requests arriving after the polling instant must wait for service until the next polling instant, their response times may increase. The deferrable server (DS) (Lehoczky et al., 1987) is similar to the SS in that it preserves the server capacity during its period. However, it may provide a smaller capacity than the SS if the server periods of the DS and the SS are the same. This is due to the double-blocking problem (Bernat and Burns, 1999). Double-blocking may also increase the interference between periodic tasks. Although a slack-stealing scheme (Lehoczky and Ramos-Thuel, 1992) has been proved to be optimal in terms of average response time for an aperiodic workload, it is difficult to apply this scheme to the scheduling of a garbage collector for several reasons (Kim et al., 1999). In contrast, the SS has many advantages over other aperiodic task scheduling algorithms. First, it has a moderately low run-time overhead. Second, the available time for an aperiodic workload is evenly spread due to the nature of the SS approach. This property facilitates the off-line estimation of R_{GC}. Third, it guarantees that the garbage collector starts immediately at t_s^k by assigning the highest priority to the server and reserving a certain amount of memory for its use.

We will therefore adopt the sporadic server for servicing garbage collection requests: it operates as follows. Initially, the server preserves its capacity, waiting for the arrival of a garbage collection request $\mathcal{G}_k$. If a mutator task tries to allocate a certain amount of heap memory and not enough remains to accommodate this request, a garbage collection request is released. Once $\mathcal{G}_k$ arrives, the server starts to service the request as long as capacity permits. If $\mathcal{G}_k$ requires more computation time than is available, then it will resume execution after the server capacity is replenished.

Once the period of the server T_s is determined, the server capacity which can guarantee the schedulability of all the tasks in $\mathcal{M}$ can be obtained. We call this value the *safe* capacity. We select $T_s = T_1$ in order that the server may run with the highest priority. The worst-case response-time analysis for the SS (Ghazalie and Baker, 1995; Bernat and Burns, 1999) is used to compute the *safe* capacity of the server. The worst-case response time of $\mathcal{M}_i$, $R_i = w^{(m)}$, where $w^{(m+1)} = w^{(m)}$, is given by the following recurrence relation:

$$w^{(m)} = C_i + \sum_{j \in hp(i)} \left\lceil \frac{w^{(m-1)}}{T_j} \right\rceil C_j + \left\lceil \frac{w^{(m-1)}}{T_s} \right\rceil C_s, \tag{4}$$

where $hp(i)$ is the set of tasks that have higher priority than $\mathcal{M}_i$. The server capacity C_s for a fixed T_s is the maximum value that satisfies $R_i \leq D_i$ for $\forall i$.

It is necessary to compute R_{GC} in order to determine the worst-case memory requirement. Because the sporadic server services each $\mathcal{G}_k$ with the highest priority, other mutators cannot preempt its execution, which is suspended only when the available server capacity is exhausted. For this reason, we can compute R_{GC} only with C_{GC}, C_s, and T_s.

Theorem 1 *Once C_{GC}, C_s, and T_s are determined, the response time of the garbage collector R_{GC} is bounded by*

$$(T_s - C_s) \left\lceil \frac{C_{GC}}{C_s} \right\rceil + C_{GC},$$

where $R_{GC} \geq R_k^{gc}$ for $k \geq 1$.

Proof. Consider the cases depicted in Figure 4. Let ε (> 0) be the available server capacity when a new garbage collection request is released. If the condition $C_{GC} \leq \varepsilon$ is satisfied, then the garbage collection request $\mathcal{G}_k$ is completely serviced within one server period. Otherwise, additional server periods are required to complete $\mathcal{G}_k$. After the remaining server capacity ε has run out, the remaining garbage collection work must be processed after the server capacity is replenished.

The interval, say Δ, before the first replenishment is at most $(T_s - C_s)$. If the condition $\varepsilon = C_s$ is satisfied, then Δ is equal to $(T_s - C_s)$. This is because the remaining capacity ε equals the full capacity C_s and, as a result, there is no additional replenishment caused by a previous garbage collection request, as shown in Figure 4 (a). On the other hand, if $\varepsilon < C_s$ then $\Delta \leq (T_s - C_s)$. For case (b) in Figure 4, the first replenishment occurs at T_s after the last fragment of the previous garbage collection request has been released and $\Delta < (T_s - C_s)$. We can prove this by contradiction. Let us assume $\Delta \geq (T_s - C_s)$ for case (b). In Figure 4 (b), $T_s - \Delta = C_s + (t_s^k - t_e^{k-1})$. Then, $\Delta = (T_s - C_s) - (t_s^k - t_e^{k-1})$. Because $t_s^k > t_e^{k-1}$ for $\forall k$, the assumption is not valid. For case (c), it is easy to prove that $\Delta = (T_s - C_s)$.

In summary, the response time of $\mathcal{G}_k$, denoted by R_k^{gc}, is the sum of ε, Δ, any additional server periods required for replenishment, and the CPU demand remaining at the end of GC cycle as shown below:

$$\begin{aligned} R_k^{gc} &= \varepsilon + \Delta + \left(\left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil - 1 \right) T_s + \left\{ C_{GC} - \varepsilon - \left(\left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil - 1 \right) C_s \right\} \\ &= \Delta + \left(\left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil - 1 \right) T_s + \left\{ C_{GC} - \left(\left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil - 1 \right) C_s \right\} \\ &= \Delta + (T_s - C_s) \left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil - (T_s - C_s) + C_{GC}. \end{aligned}$$

Consider the situation in Figure 4 (c). This is the worst case. It occurs if t_s^k coincides with t_e^{k-1} and the first replenishment occurs at $\varepsilon + (T_s - C_s)$ time units after $\mathcal{G}_k$ is released. Then, $\Delta = (T_s - C_s)$ and the above equation is reduced to

$$R_k^{gc} = (T_s - C_s) \left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil + C_{GC}.$$

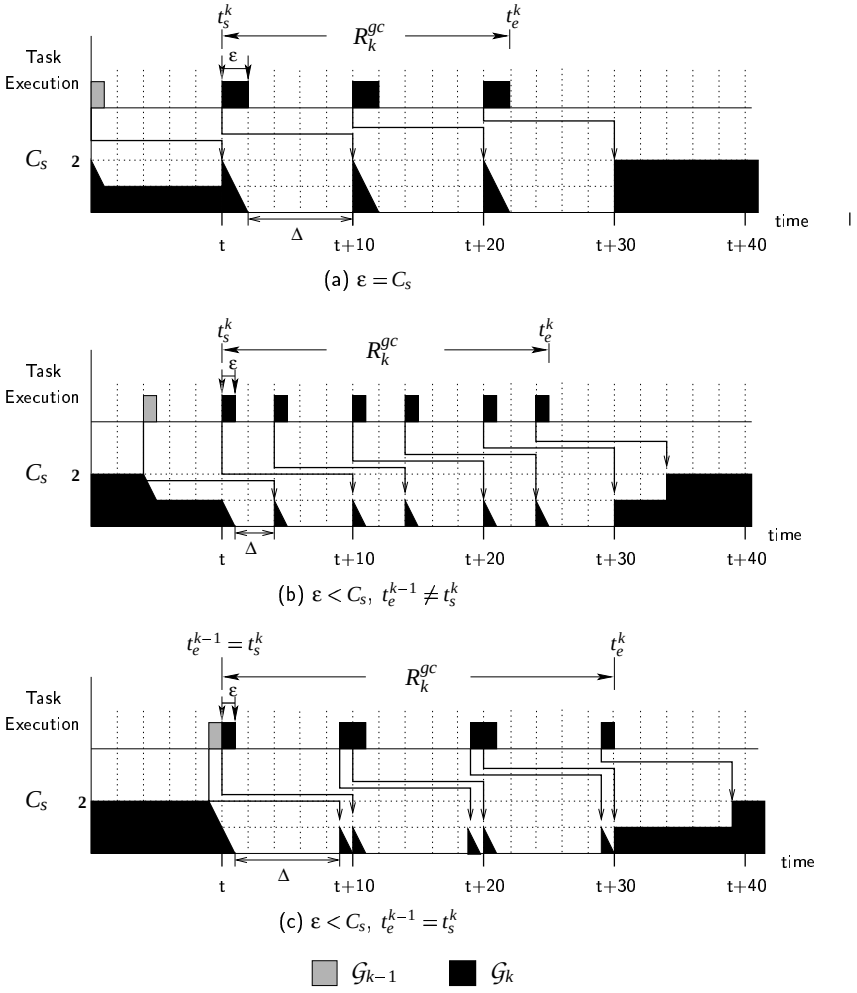


Figure 4: Response time of $\mathcal{G}_k$ ($T_s = 10, C_s = 2, C_{GC} = 6$).

Since $\varepsilon > 0$

$$\left\lceil \frac{C_{GC} - \varepsilon}{C_s} \right\rceil < \left\lceil \frac{C_{GC}}{C_s} \right\rceil .$$

It follows that the response time of garbage collector is bounded by

$$(T_s - C_s) \left\lceil \frac{C_{GC}}{C_s} \right\rceil + C_{GC}.$$

□

Taking into account the replenishment of server capacity, we have

$$\mathcal{R}_{\mathcal{G}} = \begin{cases} R_{GC} + (T_s - C_s) & \text{if } C_{GC} = mC_s, m \text{ is integer} \\ R_{GC} & \text{otherwise.} \end{cases} \tag{5}$$

Let us consider the cases illustrated in Figure 3. If $C_{GC} = mC_s$, then $R_{GC} = mT_s$. In this case, $\mathcal{G}_{k+1}$ must arrive at least $(T_s - C_s)$ time units after t_e^k in order for enough CPU time to be available at flip time. Otherwise, $\mathcal{G}_{k+1}$ can start at t_e^k since the server capacity has not been exhausted.

We now compute π_i to complete memory reservation. Previous work (Kim et al., 1999) applies the worst-case of each $\mathcal{M}_i$, and thus overestimates the amount of memory required. To remedy this problem, we take the worst case in which all mutators are released immediately before $\mathcal{G}_k$ starts. Hence, we obtain

$$\pi_i = \left\lceil \frac{\mathcal{R}_{\mathcal{G}}}{T_i} \right\rceil. \tag{6}$$

4.4 Live-Memory Analysis

In this section we investigate the worst-case behavior of live memory. Since the previous sections have shown that L_k^* is the dominant factor in determining garbage collection time, an analysis of live memory provides the basis for determining the worst-case garbage collection time. Our analysis is based on the characteristics of object lifetime in embedded real-time systems and the underlying scheduling policy.

We classify heap objects into *global* and *local* objects. While local objects are only reachable by the task that has those allocated to its context, global objects can be referenced by all tasks. As pointed out by Dieckmann et al. (Dieckmann and Hölzle, 1999), the number of global live objects is relatively stable throughout program execution because global objects are significantly longer-lived than local objects. On the other hand, the number of local live objects continues to vary until the time at which the garbage collector is triggered. For this reason, we concentrate on the off-line estimation of the worst-case local live-memory.

According to previous studies (Appel, 1987; Barrett and Zorn, 1993; Wilson, 1994), the average execution time of a copying collector decreases as heap size increases, because of the short lifetime of objects and the characteristics of copying collectors. Long-running non-real-time applications thus produce more garbage as the interval between successive garbage collection instances becomes longer. However, the same is not true for the real-time, periodic tasks dealt with in this paper. The amount of live memory for each task depends not on the heap size but on the state of each task. Although the amount of live memory is a function of A_i and varies during the execution of a task instance, it is stabilized at the end of each instance. Thus, our approach aims to analyze the live memory based on the states of periodic mutator tasks.

Our approach does not entirely depend on traditional pointer analysis techniques because they do not consider the periodic execution of tasks in real-time systems, although they can provide useful information about live memory. Instead, we find the worst-case live-memory requirement by classifying the state of a task instance into two categories: *active* and *inactive*. We regard a task as *active* if the task is running or preempted by higher-priority tasks at time t. Otherwise, the task is regarded as *inactive*. We set the amount of live memory for an active task $\mathcal{M}_i$ to A_i in order to cover an arbitrary live-memory distribution. In contrast, the amount of live memory for an inactive task $\mathcal{M}_j$ converges to $\gamma_j A_j$. Consequently, the sum of local live objects processed by $\mathcal{G}_k$, $L_{k,local}$, amounts to

$$L_{k,local} = \sum_{\mathcal{M}_i \in active(t_s^k)} A_i + \sum_{\mathcal{M}_j \in inactive(t_s^k)} \gamma_j A_j, \tag{7}$$

where *active(t)* and *inactive(t)* denote the set of active tasks and the set of inactive tasks at time t. Its maximum value $L_{k,local}^*$ is given by

$$L_{k,local}^* = max\Big(\sum_{\mathcal{M}_i \in active(t_s^k)} A_i + \sum_{\mathcal{M}_j \in inactive(t_s^k)} \gamma_j A_j \Big). \tag{8}$$

We also assume the amount of global live memory to be constant $L_{k,glob}^*$ because it is known to be relatively stable throughout the execution of the application. Then, L_k^* equals the sum of $L_{k,local}^*$ and $L_{k,glob}^*$.

It is difficult, however, to predict whether a task is actually active at an arbitrary time instant t because the garbage collector may interfere with the periodic tasks. As to the live memory, the worst-case interference occurs when garbage collection starts at the same time as the highest priority task $\mathcal{M}_1$ is running with the other tasks $\mathcal{M}_i$ preempted by the tasks $\mathcal{M}_{i-1}$ ($2 \le i \le n$) (i.e., all the tasks are active). In this case, the *trivial* upper bound of $L_{k,local}$ is equal to $\sum_{i=1}^{n} A_i$, which is too pessimistic.

We introduce a three-step approach in order to find a tighter bound on $L_{k,local}$ in lieu of the trivial upper bound $\sum_{i=1}^{n} A_i$. For simplicity, we put a restriction on the periods of mutators by requiring that $\mathcal{M}_i$ is *harmonic* with respect to $\mathcal{M}_1$ ($\forall\, i,\, 2 \le i \le n$). A task $\mathcal{M}_i$ is harmonic with respect to a task $\mathcal{M}_j$ if T_i is exactly divisible by T_j (Gerber et al., 1994). This harmonicity constraint helps to prune the search space of the following live-memory analysis algorithm.

- **Step 1. Find the active windows**: Let $\mathcal{S}_{i,j}$ and $\mathcal{F}_{i,j}$ denote the earliest start time and the latest finish time of $\mathcal{M}_{i,j}$. We find the *active window* $\mathcal{A}_{i,j}$=[$\mathcal{S}_{i,j}$, $\mathcal{F}_{i,j}$] for $\mathcal{M}_{i,j}$ as follows. First, let $CS_j(i)$ denote the set of tasks $\mathcal{M}_k$ such that their priorities are higher than that of $\mathcal{M}_i$ and $\exists l, (j-1)T_i = lT_k$, for $1 \leq l \leq \frac{H}{T_k}$. Next, let $\mathcal{M}_\kappa$ and R_κ denote the lowest-priority task in $CS_j(i)$ and its worst-case response time, respectively. We will ignore the CPU bandwidth reserved for the garbage collection (i.e., the capacity of the sporadic server) in computing R_κ because no garbage collection is necessary in the best case. R_κ is given by $R_\kappa = w$ where w is the smallest solution to the following recurrence relation:

$$w = C_\kappa + \sum_{l \in (CS_j(i) \cap hp(\kappa))} \left\lceil \frac{w}{T_l} \right\rceil C_l, \tag{9}$$

where $hp(\kappa)$ is the set of tasks whose priorities are higher than that of $\mathcal{M}_\kappa$. $\mathcal{S}_{i,j}$ equals the sum of $(j-1)T_i$ and R_κ because $\mathcal{M}_{i,j}$ can actually start when there are no higher-priority tasks released. We compute $\mathcal{F}_{i,j}$ under the assumption that the total capacity of the SS is used for garbage collection (i.e., the garbage collector behaves like a periodic task with a computation time of C_s and period of T_s). If R_i^s denotes the worst-case response time of $\mathcal{M}_i$ in this case, then $\mathcal{F}_{i,j} = (j-1)T_i + R_i^s$. We compute $R_i^s = w$ using the recurrence relation

$$w = C_i + \sum_{l \in hp(i)} \left\lceil \frac{w}{T_l} \right\rceil C_l, \tag{10}$$

where $hp(i)$ is the set of tasks, including the sporadic server, whose priorities are higher than that of $\mathcal{M}_i$. Finally, we have

$$\begin{aligned} \mathcal{A}_{i,j} &= [\mathcal{S}_{i,j},\ \mathcal{F}_{i,j}] \\ &= [(j-1)T_i + R_\kappa,\ (j-1)T_i + R_i^s], \\ &\quad \text{where } 1 \leq i \leq n,\ 1 \leq j \leq \frac{H}{T_i}. \end{aligned} \tag{11}$$

- **Step 2. Find the transitive preemption windows**: Let $\mathbf{P}_{\mathcal{M}_i \Rrightarrow \mathcal{M}_j} = \{(\psi_s, \psi_f)\}$ denote the set of time intervals in which $\mathcal{M}_i$ *preempts* $\mathcal{M}_j$ $(i < j)$. They are equivalent to the intervals in which the windows $\mathcal{A}_{i,p}$ $(1 \leq p \leq \frac{H}{T_i})$ and $\mathcal{A}_{j,q}$ $(1 \leq q \leq \frac{H}{T_j})$ overlap. Similarly, the intervals which intersect among $\mathcal{A}_{i,p}$ $(1 \leq p \leq \frac{H}{T_i})$, $\mathcal{A}_{j,q}$ $(1 \leq q \leq \frac{H}{T_j})$, and $\mathcal{A}_{k,r}$ $(1 \leq r \leq \frac{H}{T_k})$ form the *transitive* preemption window $\mathbf{P}_{\mathcal{M}_i \Rrightarrow \mathcal{M}_j \Rrightarrow \mathcal{M}_k}$. If $\mathcal{M}_k \in CS_j(i)$, both $\mathcal{M}_k$ and $\mathcal{M}_i$ are released at $(j-1)T_i$, and $\mathcal{M}_k$ immediately preempts $\mathcal{M}_i$. We do not address this *immediate* preemption in finding the preemption window, and so $\mathcal{M}_i$ is treated as *inactive* in this case. Step 2 iterates until all the transitive preemption windows are found. Its computational complexity is $O(n^2)$. The details of Step 2 are shown in Figure 5.

- **Step 3. Compute the worst-case live memory**: We compute the local live memory by Eq. (8) using the results of Step 2. Suppose that the given task-set has 4 tasks. If the transitive preemption window $\mathbf{P}_{\mathcal{M}_1 \Rrightarrow \mathcal{M}_2 \Rrightarrow \mathcal{M}_4}$ exists, the local live memory $L_{k,local}$ for this combination is $L_{k,local} = A_1 + A_2 + \gamma_3 A_3 + A_4$. In this step, we compute $L_{k,local}$ for all the combinations found in step 2. Finally, we set their maximum value to $L^*_{k,local}$.

An example follows to clarify the proposed approach.

Step 2: Finding all the transitive preemption windows

Input: Active windows for all task instances

for $i := n$ to 2 **do**
  $j := i - 1$
  currentPreemptionWindow $:= \varnothing$
  while $j \geq 1$ **do**
    if *currentPreemptionWindow* $== \varnothing$ **then**
      if $(p * T_i \neq q * T_j)$, and ($\mathcal{A}_{i,p}$ and $\mathcal{A}_{j,q}$ overlap) **then**
        add the overlapping intervals to $\mathbf{P}_{\mathcal{M}_j \Rrightarrow \mathcal{M}_i}$
        currentPreemptionWindow $:= \mathbf{P}_{\mathcal{M}_j \Rrightarrow \mathcal{M}_i}$
        currentTarget $:= j$
      endif
    else
      if $(p * T_{currentTarget} \neq q * T_j)$, and (currentPreemptionWindow and $\mathcal{A}_{j,q}$ overlap) **then**
        update *currentPreemptionWindow* to include the transitive preemption window
        currentTarget $:= j$
      endif
    endif
    $j := j - 1$
  endwhile
  output *currentPreemptionWindow*
endfor

Result: all the combinations $\mathcal{M}_i \Rrightarrow ... \Rrightarrow \mathcal{M}_k$ of which the preemption window $\mathbf{P}_{\mathcal{M}_i \Rrightarrow ... \Rrightarrow \mathcal{M}_k} \neq \varnothing$

Figure 5: Finding the transitive preemption windows.

Example 1 *Consider the task-set with the parameters given in Table 2.*

Table 2: Example task-set: $T_s = 10$, $C_s = 3$

	C_i	T_i	D_i	A_i	γ_i	R_i^s
$\mathcal{M}_1$	2	10	10	988	0.43	5
$\mathcal{M}_2$	4	30	30	1028	0.36	9
$\mathcal{M}_3$	10	60	60	1200	0.38	29
$\mathcal{M}_4$	15	120	200	1696	0.27	106

- **Step 1.** *The active windows of periodic tasks in the example are*

$$\mathcal{A}_{1,j} = [10(j-1),\ 10(j-1)+5],$$
$$\mathcal{A}_{2,j} = [30(j-1)+2,\ 30(j-1)+9],$$
$$\mathcal{A}_{3,j} = [60(j-1)+6,\ 60(j-1)+29],$$
$$\mathcal{A}_{4,j} = [120(j-1)+18,\ 120(j-1)+106], \quad \textit{where } 1 \leq j \leq \tfrac{120}{T_i}.$$

- **Step 2.** *Using the active windows found in Step 1, we can determine the preemption windows for the following combinations:* $\mathcal{M}_1 \Rrightarrow \mathcal{M}_3$, $\mathcal{M}_1 \Rrightarrow \mathcal{M}_4$, $\mathcal{M}_2 \Rrightarrow \mathcal{M}_4$, $\mathcal{M}_3 \Rrightarrow \mathcal{M}_4$, *and* $\mathcal{M}_1 \Rrightarrow \mathcal{M}_3 \Rrightarrow \mathcal{M}_4$.

- **Step 3.** *As a result of Eq. (8),* $\mathcal{M}_1 \Rrightarrow \mathcal{M}_3 \Rrightarrow \mathcal{M}_4$ *is the combination that maximizes the amount of local live memory. In this case,* $L^*_{k,local}$ *is reduced by up to 13% compared with the trivial bound.*

As shown in Example 1, the proposed algorithm provides a bound of $L^*_{k,local}$ which is tighter than the trivial upper bound. It is, however, necessary to prove the safety of our algorithm. Although the actual preemption scenario will be quite different from the analysis because we can not know off-line when $\mathcal{G}_k$ starts, we can nevertheless prove the safety of our algorithm. Once $\mathcal{G}_k$ starts, it behaves like a periodic task with period T_s and computation time C_s until it completes the garbage collection. The proposed algorithm assumes a *critical instant* (Liu and Layland, 1973) in estimating $s_{i,j}$ and $f_{i,j}$. Thus, the active window found in our algorithm completely includes the actual active window for $\mathcal{M}_{i,j}$, at whatever time instant $\mathcal{G}_k$ is triggered.

4.5 Worst-Case Memory Requirement Revisited

The worst-case memory requirement is the sum of the memory reserved for periodic mutator tasks and the worst-case live memory, as shown in Eq. (3). Because the reserved memory depends on C_{GC} and vice versa, we need to calculate the amount of reserved memory, M_{resv}, iteratively. First, we compute L^*_k using the algorithm in Section 4.4. Next, we assign the amount of memory allocated by all $\mathcal{M}_{i,j}$s during H to the initial value of M_{resv}. Thereafter, the recurrence relation performs each m^{th} computation of M_{resv}, $M^{(m)}_{resv}$ successively. The algorithm uses C_{GC} derived from $M^{(m-1)}_{resv}$ to compute $M^{(m)}_{resv}$. If $M^{(m)}_{resv} = M^{(m-1)}_{resv}$, the computation is terminated and, as a result, $M_{resv} = M^{(m)}_{resv}$. Figure 6 summarizes this procedure.

Computation of the system memory requirement

M_{min}: minimum memory requirement
M_{resv}: worst-case reserved memory
M^{prev}_{resv}: worst-case reserved memory computed in the previous iteration

$M_{resv} := \sum_{i=1}^{n} \frac{H}{T_i} A_i$
do
  begin
    $M^{prev}_{resv} := M_{resv}$
    Compute C_{GC} as described in Section 4.1
    Compute R_{GC}, $\mathcal{R}_{\mathcal{G}}$, and π_i
    Compute M_{resv}
  end
while $(M^{prev}_{resv} \neq M_{resv})$
$M_{min} := M_{resv} + L^*_k$;

Figure 6: Algorithm for the computation of memory requirement.

5 Performance Evaluation

In this section, we evaluate the efficiency of our approach. First, we confirm the effectiveness of the live-memory analysis by comparing the results of our analysis with the simulation results and the trivial upper bound. Second, we compare the worst-case memory reservation of our approach with the traditional background policy. Third, we conduct a series of simulations to compare the memory requirement of various scheduling policies.

We wrote five application programs in Java for these experiments. The applications are *LMS*, *ARFREQ*, *RAD-PROC*, *RFAST* and *RTPSE* (Embree, 1995) as listed in Table 3. We partly modified the logging facility of the garbage collector and the JVMPI (Java Virtual Machine Profiler Interface) (JVM, 1999) in PersonalJava™. This modified PersonalJava performs live-memory traversal after every 8 bytes of memory are allocated and logs the results. The following parameters are obtained from the trace-driven analysis of the applications running on our modified PersonalJava.

Table 3: General information about the sample applications.

Sample Applications	Description
LMS	Fixed-point adaptive FIR (Finite Impulse Response) filter where the filter coefficients are updated on each input signal.
ARFREQ	AR (AutoRegressive) frequency estimation using a 35-point Hilbert transform FIR filter.
RADPROC	Real-time radar signal processing using the FFT.
RFAST	Fast convolution using the FFT.
RTPSE	Power spectral estimation using the FFT.

- Worst-case memory allocation requirement for periodic tasks (A_i).
- Stable live factor from A_i for *inactive* tasks (γ_i).
- Maximum global live memory.
- Memory allocation distributions.

We consider three sets of periodic tasks created out of the sample applications in this section. Their specifications are given in the Appendix. The CPU utilization U_p of TS1, TS2 and TS3 are 0.673, 0.738 and 0.792, respectively.

Section 4.4 gives an effective live-memory analysis technique based on the states of tasks. To show the effectiveness of this technique, we analyze the worst-case local live-memory requirement for each task-set and compare them with the simulation results. Table 4 shows the worst-case memory requirement for each task-set. In Table 4, L_{sim}, L_{anal}, L_{lb}, and L_{ub} denote the worst-case local live memory derived from analysis of the simulation, and the lower and upper bounds derived from Eq. (8). These results demonstrate that the analytic bound accords well with the simulation result. The proposed analysis also reduces the worst-case local live memory requirement by up to 18∼35 % compared with L_{ub}. As pointed out in Section 4, the decrease in $L^*_{k,local}$ can reduce the total memory requirement as well as C_{GC}.

Table 4: Live memory of each task-set.

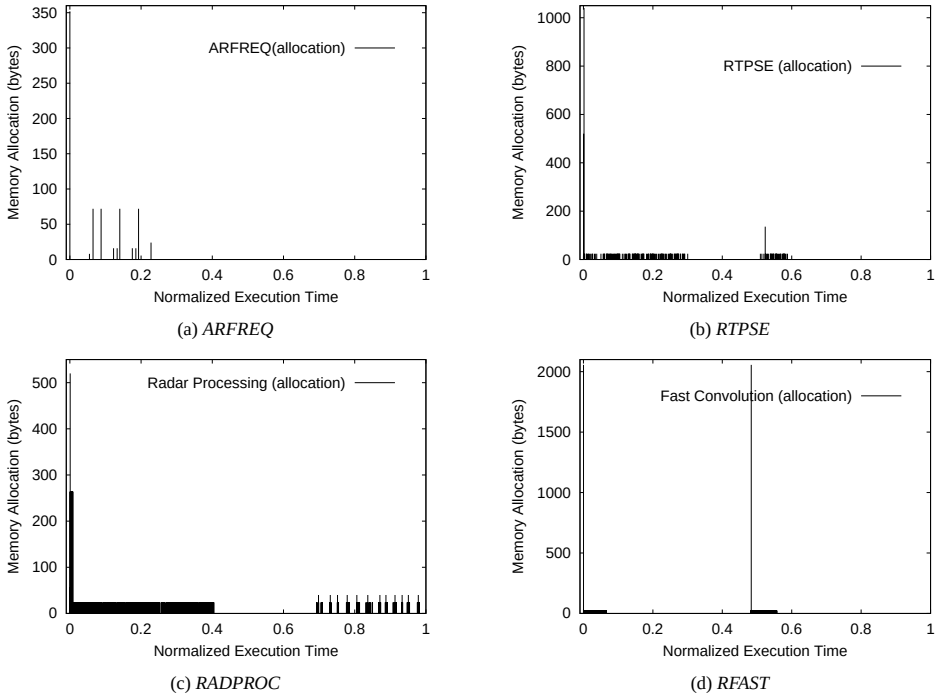
task-set	L_{sim}	L_{anal}	L_{ub}	L_{lb}
TS1	49502	50271	61536	16963
TS2	62665	64784	88646	26144
TS3	75903	77910	121294	36942

We are also interested in the effect of scheduling policy on memory reservation for periodic tasks. We compare the memory reservation, say M_{resv}, of the background policy with that of our policy. Although C_{GC} should be estimated in a different way for the background policy, we assume that both of them are identical. The worst-case garbage collection time for the background policy is determined by the memory allocation requests of the preceding high-priority tasks, and thus is not bounded by L^*_k. Our policy reduces M_{resv} by up to 32∼56 % against the background policy, as shown in Table 5. This is because R_{GC} determines M_{resv} for a given task-set.

Table 5: Memory reservation for the background policy and the proposed policy.

task-set	Background	Proposed
TS1	97332	66370
TS2	156628	102142
TS3	354958	153988

We also conducted a series of simulations to compare the *feasible* memory requirement of our policy with other aperiodic scheduling policies. By feasible memory requirement we mean the amount of heap memory required to avoid memory starvation of hard real-time tasks and deadline misses. The feasible memory requirement is smaller than the worst-case memory requirement because the former is based on a specific memory consumption function while the latter assumes batch allocation. In all simulations, the amount of live memory at t_s^k determines the computation time of $\mathcal{G}_k$ where Eq. (2) is used for the computation. The coefficients in Eq. (2) are derived from a static measurement of the prototype garbage collector running on a 50 MHz MPC860 with SGRAM.



(a) *ARFREQ* (b) *RTPSE*

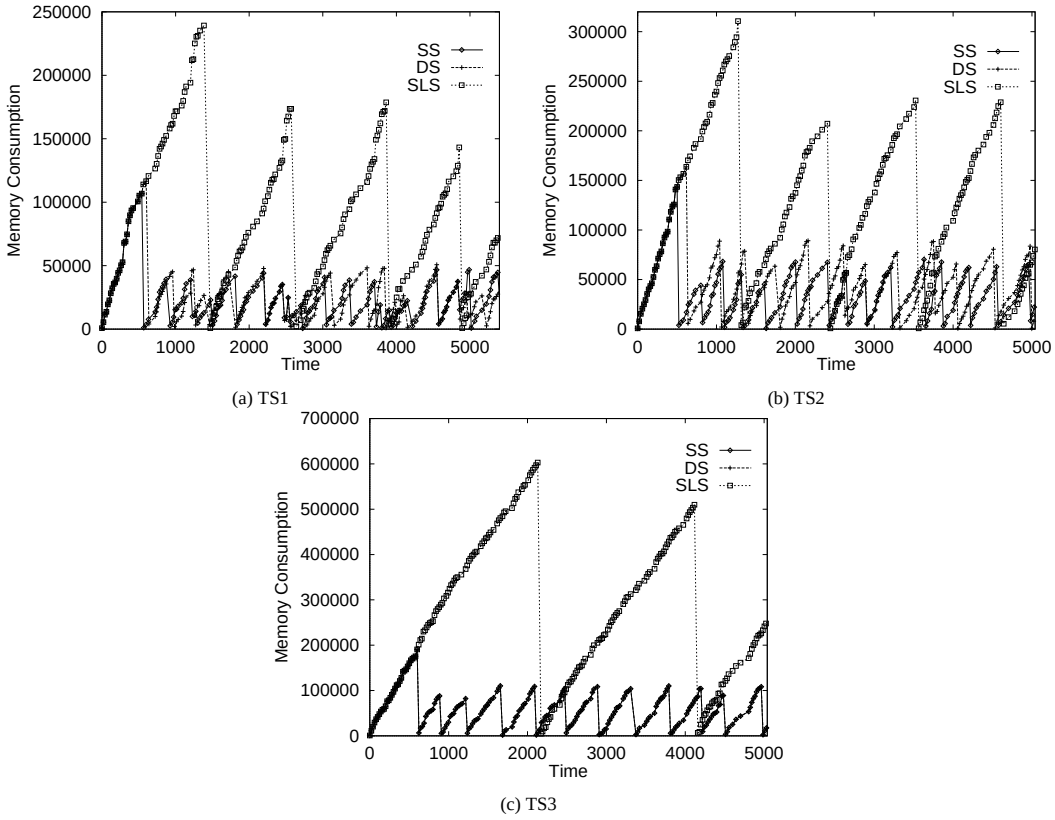
(c) *RADPROC* (d) *RFAST*

Figure 7: Memory allocation distribution of sample applications (total execution time = 1).

Our previous study (Kim et al., 1999) assumed that the memory allocation behavior of each $\mathcal{M}_i$ follows a uniform distribution. It results in noticeable gaps between predicted memory requirements and the simulation results. In order to discover the actual memory consumption behavior, we use a trace-driven memory allocation distribution derived from the sample applications. Figure 7 illustrates the memory allocation distributions for the normalized execution time of sample applications. It shows that most memory allocations occur within the first half of the task execution.

The feasible memory requirement is found by iterative simulation runs. We regard a given memory requirement as feasible if no garbage collection errors or deadline misses of periodic tasks are reported after 100 hyperperiods. The simulator reports a garbage collection error if there is no server capacity or slack time when a garbage collection request arrives. If an error is reported, the simulator increases the heap size by 1000 bytes. This procedure is repeated until the test terminates without any error, and the final heap size is taken to be the feasible memory requirement.

The scheduling policies that we have compared with our approach are the deferrable server (DS) and slack stealing (SLS). We do not consider the background policy or the polling server because they may not provide sufficient CPU time at flip time as pointed out by Kim et al. (Kim et al., 1999). Figure 8 illustrates the memory allocation behavior and the feasible memory requirement of each scheduling policy for given tasks sets. Table 6 summarizes the results.



(a) TS1 (b) TS2

(c) TS3

Figure 8: Feasible memory requirement of given task-sets.

Table 6: Feasible memory requirement of various scheduling policies.

task-set Policies	Our approach (SS)	DS	SLS
TS1	113000	117000	245000
TS2	148000	166000	311000
TS3	200000	200000	604000

The results show that our policy is slightly better than the DS, and greatly outperforms the SLS. We also measured the best, average, and worst-case response times of the garbage collector. For all the task-sets, our approach provides a more stable response-time distribution than the other policies, which results in a reduced memory requirement. The DS provides the shortest best-case response time due to the *double hit* effect (Bernat and Burns, 1999). This effect, however, yields a smaller server capacity than the SS in most cases, thus producing a longer R_{GC}. It eventually results in a 3∼12 % performance degradation for TS1 and TS2. For TS3, the memory requirement of our approach and the DS are the same. This is because the server capacity of the DS equals that of our approach. The noticeable gap between our policy and SLS can be explained as follows: since the SLS uses idle time aggressively, there will be little idle time left over by the end of each hyperperiod, as already pointed out by Kim et al. (Kim et al., 1999). In the experiments, most garbage collection errors, which increased feasible memory requirement, resulted from zero slack occurring near the end of each hyperperiod.

6 Design Considerations

The drawback of our approach is that aperiodic tasks cannot use the capacity of the sporadic server when the garbage collection is not in progress. Because it is too tight a constraint, we have attempted to find a smaller C_s, while minimizing the memory requirement. Table 7 shows the memory requirement of TS1 for various values of C_s. The results show that although a smaller C_s may produce lower $L^*_{k,local}$, it may not reduce the total memory requirement accordingly. This is because a smaller C_s results in a longer R_{GC}, thus increasing the memory reservation M_{resv}.

Table 7: Memory requirement of TS1 (T_s=30) for varying values of C_s

.

C_s	$L^*_{k,local}$	M_{resv}	$\frac{M}{2}$
7	50271	66370	165745
6	50271	66370	165745
5	50274	70444	169819
4	49502	71228	169834
3	49502	75278	173884

We now suggest how C_s be determined depending on which resource is the primary consideration, CPU or memory resource. This paper determines the memory requirement for the maximum C_s, which is in turn determined by the schedulability test of periodic tasks. We can also use the proposed algorithm to find the minimum C_s for a given heap size when CPU resource is the primary consideration. For the example in Table 7, if we have 5 % more memory in the system, we can reduce C_s by up to 50 %. That also leads to about 10 % reduction of total CPU utilization. Although the relation between CPU and memory resource depends on the characteristics of given task-sets, it delivers useful information for the design of embedded real-time systems with garbage collection.

7 Conclusions and Future Work

This paper has focused on the integration of garbage collection with task scheduling to guarantee the schedulability of hard real-time tasks while minimizing memory requirement. Because the worst-case live memory is an essential metric for scheduling-integrated garbage collection, we have developed an effective live-memory analysis based on the states of periodic tasks. By scheduling garbage collection requests using the sporadic server, our approach can considerably reduce the memory requirement compared with the traditional background approach while meeting the deadlines of hard real-time mutators. Performance evaluation shows that the proposed policy yields minimum feasible memory requirement among various aperiodic scheduling policies.

This paper also provides a design metric for embedded real-time systems with limited memory resource. Given a heap memory and a task-set, system designers can use the proposed algorithm to determine the CPU budget for garbage collection.

For simplicity, we have assumed that there are no blocking factors. Future work must consider the blocking factors caused by synchronization of shared resources, including synchronization of heap accesses such as the pointer-writes.

Acknowledgement

The authors would like to thank the anonymous referees for their valuable comments and suggestions. The authors also wish to thank Namyun Kim and Taewoong Kim for the useful discussions about the scheduling approach.

This work was supported in part by S.N.U Research Fund under Grant 982110, in part by ITEP GITP under Grant B32-979-3305-03-1-3, and in part by the Brain Korea 21 project.

References

Appel, A. W. Garbage collection can be faster than stack allocation. *Information Processing Letters*, 25(4):275–279, 1987.

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

Barrett, D. A. and Zorn, B. G. Using lifetime predictors to improve memory allocation performance. In *Proceedings of SIGPLAN'93 Conference on Programming Languages Design and Implementation*, pp. 187–196, June 1993.

Bernat, G. and Burns, A. New results on fixed priority aperiodic servers. In *Proceedings of Real-Time Systems Symposium*, pp. 68–78, December 1999.

Brooks, R. A. Trading data space for reduced time and code space in real-time collection on stock hardware. In *Conference Record of the 1984 ACM Symposium on LISP and Functional Programming*, pp. 256–262, 1984.

Dieckmann, S. and Hölzle, U. A study of the allocation behavior of the specjvm98 java benchmarks. In *Proceedings of the 13th European Conference on Object-Oriented Programming*, June 1999.

Dijkstra, E. W., Lamport, L., Martin, A. J., Scholten, C. S., and Steffens, E. F. M. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):965–975, November 1978.

Embree, P. M. *C Algorithms for Real-Time DSP*. Prentice Hall PTR, 1995.

Gerber, R., Hong, S., and Saksena, M. Guaranteeing end-to-end timing constraints by calibrating intermediate processes. In *Proceedings of Real-Time Systems Symposium*, pp. 192–203, December 1994.

Ghazalie, T. M. and Baker, T. P. Aperiodic servers in a deadline scheduling environment. *Journal of Real-Time Systems*, 9(1):31–67, July 1995.

Henriksson, R. *Scheduling Garbage Collection in Embedded Systems*. Ph.D. Thesis, Lund University, Sweden, July 1998.

Java Virtual Machine Profiler Interface (JVMPI). http://java.sun.com/products/jdk/1.2/docs/guide/jvmpi/jvmpi.html, 1999.

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 on Languages, Compilers and Tools for Embedded Systems*, pp. 55–64, May 1999.

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*, pp. 46–55, June 2000.

Lehoczky, J. P. and Ramos-Thuel, S. An optimal algorithm for scheduling soft-aperiodic tasks in fixed-priority preemptive systems. In *Proceedings of the Real-Time Systems Symposium*, pp. 110–123, December 1992.

Lehoczky, J. P., Sha, L., and Strosnider, J. K. Enhanced aperiodic responsiveness in hard real-time environment. In *Proceedings of the Real-Time Systems Symposium*, pp. 261–270, December 1987.

Liu, C. L. and Layland, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

Persson, P. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers and Tools for Embedded Systems*, pp. 45–54, May 1999.

Ramos-Thuel, S. and Lehoczky, J. P. On-line scheduling of hard deadline aperiodic tasks in fixed-priority systems. In *Proceedings of the Real-Time Systems Symposium*, pp. 160–171, December 1993.

Rugina, R. and Rinard, M. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 77–90, May 1999.

Sprunt, B., Sha, L., and Lehoczky, J. P. Aperiodic task scheduling for hard-real-time systems. *Journal of Real-Time Systems*, 1:27–60, 1989.

Wilson, P. R. Uniprocessor garbage collection techniques. Technical report, University of Texas, Jan 1994.

Yong, S. H., Horwitz, S., and Reps, T. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN '99 Conference on Programming Language Design and Implementation*, pp. 91–103, May 1999.

Yuasa, T. Real-time garbage collection on general-purpose machines. *Journal of Software and Systems*, 11(3): 181–198, 1990.

Appendix

Table 8: Specifications of task sets.

Task set	Task	C_i	T_i	A_i	γ_i
TS1	$\mathcal{M}_1$	3	30	784	0.377
	$\mathcal{M}_2$	4	60	1058	0.280
	$\mathcal{M}_3$	6	90	1424	0.339
	$\mathcal{M}_4$	11	120	2232	0.314
	$\mathcal{M}_5$	13	360	3032	0.233
	$\mathcal{M}_6$	25	480	3564	0.238
	$\mathcal{M}_7$	25	540	5768	0.288
	$\mathcal{M}_8$	74	750	9288	0.349
	$\mathcal{M}_9$	65	900	5564	0.274
	$\mathcal{M}_{10}$	79	1200	28832	0.252
TS2	$\mathcal{M}_1$	3	20	784	0.377
	$\mathcal{M}_2$	2	40	1058	0.280
	$\mathcal{M}_3$	3	60	784	0.377
	$\mathcal{M}_4$	5	100	1424	0.339
	$\mathcal{M}_5$	8	200	3022	0.233
	$\mathcal{M}_6$	4	300	3022	0.233
	$\mathcal{M}_7$	10	400	1424	0.339
	$\mathcal{M}_8$	20	500	5768	0.288
	$\mathcal{M}_9$	20	600	3332	0.314
	$\mathcal{M}_{10}$	22	700	5768	0.288
	$\mathcal{M}_{11}$	40	800	5564	0.274
	$\mathcal{M}_{12}$	20	900	9288	0.349
	$\mathcal{M}_{13}$	41	960	9288	0.349
	$\mathcal{M}_{14}$	65	1000	9288	0.349
	$\mathcal{M}_{15}$	54	1200	28832	0.252
TS3	$\mathcal{M}_1$	2	20	784	0.377
	$\mathcal{M}_2$	2	40	1058	0.280
	$\mathcal{M}_3$	3	60	1424	0.339
	$\mathcal{M}_4$	5	120	1424	0.339
	$\mathcal{M}_5$	8	200	5768	0.288
	$\mathcal{M}_6$	4	300	1424	0.368
	$\mathcal{M}_7$	4	360	1058	0.280
	$\mathcal{M}_8$	10	400	5768	0.288
	$\mathcal{M}_9$	4	480	1424	0.339
	$\mathcal{M}_{10}$	20	500	5768	0.298
	$\mathcal{M}_{11}$	30	600	9288	0.349
	$\mathcal{M}_{12}$	22	700	5564	0.274
	$\mathcal{M}_{13}$	40	720	3022	0.233
	$\mathcal{M}_{14}$	40	800	9288	0.349
	$\mathcal{M}_{15}$	30	840	5768	0.298
	$\mathcal{M}_{16}$	20	900	9288	0.349
	$\mathcal{M}_{17}$	32	960	5768	0.298
	$\mathcal{M}_{18}$	50	1000	9288	0.349
	$\mathcal{M}_{19}$	65	1200	9288	0.349
	$\mathcal{M}_{20}$	60	2000	28832	0.252