Scheduling Garbage Collector for Embedded Real-Time Systems *

Taehyoun Kim  Naehyuck Chang  Namyun Kim  Heonshik Shin

Department of Computer Engineering, Seoul National University
Seoul 151-742, Korea
Phone: +82-2-880-7298

thkim@comp.snu.ac.kr  naehyuck@snu.ac.kr  nykim@comp.snu.ac.kr  shinh@comp.snu.ac.kr

Abstract

This paper proposes a new scheduling method for multiple mutators and a garbage collector running on embedded real-time systems with a single processor and no virtual memory. The hard real-time tasks should reserve a certain amount of heap memory to prevent memory starvation and/or deadline miss. Since the memory requirement depends on the worst-case response time of a garbage collector, the traditional approach in which garbage collection is performed in the background demands large memory space. The proposed scheduling algorithm is based on an aperiodic scheduling technique, sporadic server. This paper also presents a modified copying garbage collection algorithm with hardware support. In order to minimize the worst-case response time of a garbage collector thus reducing the memory requirement, the garbage collector runs as the highest priority task with a preset bandwidth. This paper also investigates the schedulability of a garbage collector and mutator tasks as well as the worst-case memory requirement. Performance analysis shows that the proposed algorithm can provide a considerable reduction in the worst-case memory requirement compared with the background policy. Simulation results demonstrate that the proposed algorithm can produce the feasible memory requirement comparable to the complex on-line scheduling algorithm such as slack stealing.

1 Introduction

Dynamic behavior of programs with various data structures forces to use dynamic memory management on heap for more efficient memory usage. Dynamic memory management requires careful and consistent memory management scheme to avoid producing memory leaks and/or dangling pointers. Manual memory management using explicit malloc/free operations is prone to result in disaster by mistakes of a programmer. By contrast, well-designed automatic memory management has many advantages over the manual memory management. It improves the productivity, robustness and program integrity by preventing possible errors.

The main functions of automatic memory management are performed by a garbage collector. The garbage collector distinguishes the memory objects that are no longer in use (garbage) from the live objects and reclaims the garbage for future use. An object is live when it is referenced by a set of special referent outside the heap (root set) or other live heap objects.

Many literature have categorized the garbage collection algorithms into two classes: reference counting and tracing. Reference counting requires an additional reference count (RC) field for each object [5, 15]. Whenever a pointer has been changed by a mutator, RC field is also updated. When the RC value drops to zero, the object is reclaimed immediately. The tracing algorithm is classified again into mark-sweep and copying. The mark-sweep collector traverses the pointers to find live objects and marks them. Then, a collector scans the entire heap and reclaims garbage objects that have not been marked. Typically copying collectors maintain two equal-sized spaces called fromspace and tospace. When a garbage collector is triggered, it traverses the pointers and copies the live objects into the new tospace.

The basic tracing garbage collection algorithms are inherently stop-the-world fashion, and sometimes their pause time is intolerable for the applications that require short or bounded response time. Incremental garbage collection algorithms [2, 4, 17] have been proposed to distribute and hide the garbage collection pause time throughout the execution of mutators. This approach, in effect, reduces the intermediate pause delay, but it is difficult to guarantee the schedulability of real-time tasks without cooperation with the scheduling mechanism. Since the deadline miss in the hard real-time systems results in a catastrophic system failure, it is not feasible to apply the simple incremental garbage collection to the hard real-time systems.

Since lots of systems are not for hard real-time applications, the incremental algorithms often show good performance. However, truly real-time garbage collection algorithms are required because real-time embedded systems recently began to adopt automatic memory management due to its advantages. A related work applies a real-time scheduling policy to real-time garbage collection [7]. In order to guarantee the deadlines of hard real-time mutators, the garbage collector should not often interrupt hard real-time mutators. Thus, a certain amount of memory is always reserved for the

*This work is partly supported by S.N.U Research Fund under Grant 982110 and ITEP GITP No. B32-979-3305-03-1-3.

1 Generally, the values in registers, stack and global variables which hold references to heap objects form the root set.
hard real-time mutators. Since the garbage collector runs as a background task in [7], it demands large amount of memory reservation. Although price of memory devices is getting cheaper day after day, it is important to reduce the memory requirement in embedded systems because dedicated systems should have price/performance competitiveness.

To reduce the memory requirement and guarantee the schedulability of hard real-time mutators under automatic memory management scheme, this paper proposes a new scheduling algorithm based on an aperiodic task scheduling technique, sporadic server. This paper also presents a modified copying collector with hardware support. A copying collector is chosen because it automatically compacts live object area and thus simplifies the estimation of garbage collection work. On the contrary, non-compacting mark-sweep collector may cause fragmentation. In many cases, the fragmentation causes severe performance degradation and rules out predictable system design. Since reference counting is too tightly coupled with the mutators [6], it is difficult to apply real-time scheduling policy to the algorithm as well.

The rest of this paper is organized as follows. Section 2 describes the task model considering the relation between a garbage collector and mutator tasks and proposes a modified copying garbage collection algorithm. Section 3 introduces a new scheduling algorithm, and it analyzes the worst-case memory requirement and the schedulability of a garbage collector and mutator tasks. Simulation studies are conducted to prove the efficiency of the proposed algorithm in Section 4. This paper concludes with future work in Section 5.

## 2 Framework of Memory Model

### 2.1 System Model and Assumptions

Consider a real-time system with a set of $n$ periodic tasks, $\mathcal{M} = \{M_1, \ldots, M_n\}$. The task model in this paper is similar to the typical task model and includes an additional property, memory allocation requirement of $M_i$. The nomenclature used in this paper is provided in Table 2.

$M_i$ is characterized by a four tuple and defined as:

\[(C_i, T_i, D_i, A_i)\]

The underlying assumptions are as follows:

- **Assumption 1**: There is no aperiodic mutator task.
- **Assumption 2**: The context switching and task scheduling overhead is negligible.
- **Assumption 3**: There is no precedence relation between $M_i$ s.
- **Assumption 4**: Any task can be instantly preempted by a higher priority task, i.e., there is no blocking factor.
- **Assumption 5**: $C_i$, $T_i$, $D_i$ and $A_i$ are known *a priori*.

Estimation of $A_i$ is generally application-specific problem. However, most real-time periodic tasks repeatedly execute identical functions or methods, and thus useful information can be obtained by straightforward analysis. Unless $A_i$ is given by the programmer, pre-runtime trace-driven analysis can also come up with estimation of $A_i$.

### 2.2 Relation between Garbage Collector and Mutators

A *garbage collection cycle* means each individual invocation of a garbage collector as shown in Figure 2. The condition that triggers the garbage collection is identical to all the tracing garbage collectors; a garbage collector is invoked when the amount of available memory becomes less than a certain threshold. From now on, the term *garbage collector* means a copying garbage collector unless specified otherwise.

#### Example 1

Discrete Kalman filter [8] iteratively estimates the state of a system. A trace-driven analysis of memory usage is applied to C++ implementation of Kalman filter which solves 1-D wave equation [9]. Table 1 shows the results.

<table>
<thead>
<tr>
<th>Size of system vector</th>
<th>$A_i$ (bytes)</th>
<th>$\alpha_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>2413</td>
<td>0.419</td>
</tr>
<tr>
<td>8</td>
<td>5741</td>
<td>0.384</td>
</tr>
<tr>
<td>16</td>
<td>17005</td>
<td>0.363</td>
</tr>
<tr>
<td>32</td>
<td>57965</td>
<td>0.349</td>
</tr>
</tbody>
</table>

Table 1: Estimation of $A_i$ and $\alpha_i$

A garbage collector and mutator tasks share a common resource, *heap*. Figure 1 illustrates the memory model depicted as producer/consumer model. The garbage collector produces memory at the rate of $f_c(k, t)$ while a mutator consumes the available memory at the rate of $f_k(M_i, k, t)$ from $t_k^i$ until a given time instant $t$ ($t_k^i \leq t < t_k^{i+1}$).

![Memory consumption/reclamation model](Figure1)

**Figure 1**: Memory consumption/reclamation model

![Garbage collection cycle for copying collector](Figure2)

**Figure 2**: Garbage collection cycle for copying collector

Let us consider the graph in Figure 3. To prevent the system breakdown due to memory starvation, the cumulative memory consumption should not exceed the amount of free memory at an arbitrary
time instant $t^k_a \leq t < t^{k+1}_a$. That is, $\Phi_{k+1}$ should not begin before the completion of $S_k$. This paper describes the relation between $m_c(k, t)$ and $m_r(k, t)$ later in detail.

The cumulative memory consumption $m_c(M_i, k, t)$ is a monotonic increasing function which is defined during the interval $[t^k_a, t^{k+1}_a]$. The invocation condition of $S_k$ is derived from $m_c(M_i, k, t)$ according to Figure 1. The memory consumption function $f_c(M_i, k, t)$ can be modeled as various types of functions, depending on the scheduling strategy and the nature of an application. In order to cover arbitrary $f_c(M_i, k, t)$, the upper bound of $m_c(M_i, k, t)$ and $m_r(k, t)$ are derived by $A_i$ and $T_i$. The value of $m_c(M_i, k, t)$ at each $t^k_a$ is zero. It depends on the number of task instances executed during $\Phi_k$. An instance $M_{i,j}$ is said to be active for $\Phi_k$ if it has not finished its execution by $t^k_a$, or it is ready at time $t$ such that $t^k_a \leq t < t^k_b$. Because the actual release time of $M_{i,j}$ is determined on-line, all instances which satisfy the above condition are regarded as active. The worst-case memory requirement of $M_i$ during a given time interval is a multiplication of $A_i$ and the worst-case number of active instances of $M_i$. The upper bound of memory consumption by $t'$ ($t^k_a \leq t' < t^{k+1}_a$) is shown below.

\[
\begin{align*}
    m_c(M_i, k, t') &\triangleq \int_{t^k_a}^{t'} f_c(M_i, k, t) \, dt \\
    &\leq \int_{t^k_a}^{t^k_b} f_c(M_i, k, t) \, dt + \left\{ \left[ \frac{t'}{T_i} \right] - \left[ \frac{t^k_a}{T_i} \right] \right\} A_i \\
    &\quad + \left\{ \left[ \frac{t^k_b}{T_i} \right] - \left[ \frac{t'}{T_i} \right] \right\} f_c(M_i, k, t) \, dt \\
    &\leq \left( \left[ \frac{t'}{T_i} \right] - \left[ \frac{t^k_a}{T_i} \right] \right) A_i \tag{1}
\end{align*}
\]

\[
\begin{align*}
    m_r(k, t') &\triangleq \sum_{i=1}^{n} m_c(M_i, k, t') \\
    &\leq \sum_{i=1}^{n} \left\{ \left[ \frac{t'}{T_i} \right] - \left[ \frac{t^k_a}{T_i} \right] \right\} A_i \tag{2}
\end{align*}
\]

$m_c(k, t)$ can also be modeled using various types of functions according to the type of the garbage collector. For the mark-sweep collector, because a garbage collector progressively reclaims the
heap area, $m_s(k, t)$ is a differentiable monotonic increasing function. On the contrary, copying collector reclaims old tospace entirely at the flip time. Actually, the amount of reclaimed heap memory during $\Phi_{k-1}$ is determined at time $t^*_k$ and derived by $M$ and the size of live objects, $L_h$. Since the reclaimed memory can be allocated by mutators again, $m_s(k, t)$ represents the minimum free memory space that can be allocated during $\Phi_{k+1}$ and given by

$$m_s(k, t') = \int_{t_k}^{t'} f_s(k, t) \, dt$$

Thus, $m_s(k, t)$ can be represented by $m_s(k)$ for a copying collector without loss of generality.

To bound the worst-case execution time of a garbage collector and determine memory requirement, $L_h$ should be estimated in advance. The amount of live memory $L_h$ is defined by the function of $A$. That is, the upper bound of live memory for $M_{i,j}$ is $a_i A_i$. As shown in [1], in general, the average execution time of a copying collector decreases as heap size increases, due to the distribution of object’s lifetime and the nature of copying collector. Many studies show that most objects are short-lived [3, 15]. Thus, for non-real-time long-running applications, the longer the interval between successive garbage collections the more objects becomes garbage thus reducing the amount of live memory and average response time of a garbage collector as well. However, the same is not true for real-time, periodic tasks dealt in this paper. Although periodic tasks give rise to an infinite sequence of jobs, instances, $L_h$ usually tends to approach a certain steady state.

Consequently, $L_h$ varies at the completion of each active $M_{i,j}$, and the upper bound can be derived by assuming all $M_{i}$s have at least one active instance for $\Phi_{k}$. $L_h$ is given by

$$L_h \leq \sum_{j=1}^{n} I_i(k)A_i \text{, where}$$

$$I_i(k) = \begin{cases} 
1 & \text{if } M_{i,j} \text{ is active during } \Phi_{k-1} \\
0 & \text{otherwise.}
\end{cases}$$

The upper bound of live memory $L_{max}$ is also given as follows:

$$L_{max} = \sum_{i=1}^{n} a_i A_i$$

2.3 Garbage Collection Algorithm

This section presents a copying collection algorithm taking the properties of real-time systems into account. The basic object format and evacuation strategy are based on Brooks’ algorithm [4].

2.3.1 Basic Algorithm

In most copying collectors, the heap is divided into two equal-sized semispaces as illustrated in Figure 4. New objects are allocated starting from the top of tospace pointed by $T$. Allocation can proceed until the amount of free space in tospace becomes less than a threshold. The roles of two semispaces are reversed and a new garbage collection cycle starts. All of the live objects are copied from fromspace to the new tospace. The evacuated objects are placed starting from the bottom of tospace pointed by $B$. The scan pointer, denoted by $S$, advances with pointer reachability relation. When $S$ pointer meets $B$ pointer, the garbage collection stops. This evacuation process should finish before the tospace area is filled up.

(a) Object Format (b) Memory Organization

Figure 4: Object format and heap organization

A barrier method preserves the consistency between the views of heap seen by a garbage collector and mutators. In Brooks’ algorithm, every object has an indirection field, called forwarding pointer, as in Figure 4 (a), and objects are always referred to via forwarding pointer. If an object is an obsolete version in fromspace, its forwarding pointer points to the new version.

Several studies [2, 13, 17] have proposed algorithms that can bound the time required to scan the root-set. Those algorithms suggest that root-set scanning should be carefully designed and implemented to reduce and bound the overhead. To achieve this goal, most of them suggest to scan the root-set incrementally. However, they do not consider the characteristics of real-time applications composed of periodic tasks. Usually, the contents of stack may change for each task instance and quite a many instances are executed during RGC. Without careful implementation, aggressive root-set scanning may produce many floating garbage or incur enormous barrier processing.

Because the correctness of a garbage collector is also important, the mutators’ operation and the garbage collection work should be carefully coordinated. In general, barrier methods provide the coherency of heap between the mutators and the garbage collector. The barrier may be either a read-barrier or a write-barrier. Since the cost of read-barrier is more expensive than write-barrier [18], recent studies prefer write-barrier to read-barrier. The use of barrier method gives rise to the asynchronous evacuation. This evacuation overhead is often harmful in guaranteeing real-time properties. A recent study proposed a lazy evacuation method [7]. The scheme executes write barrier operation when a pointer update operation makes object in tospace point to a fromspace object. Instead of evacuating the fromspace object immediately, the write-barrier reserves an area for the object. Later, a background garbage collector evacuates the object before the normal evacuation process. This approach is quite effective in that it reduces the interference on the execution of hard real-time tasks.

2.3.2 Modified Copying Collector

This section introduces performance enhancement techniques for incremental copying collectors. Because the main focus of this paper is guaranteeing schedulability and small memory requirement by the use of scheduling support, the details of garbage collector implementation are omitted.

This paper suggests that a garbage collector scan the root-set incrementally as do in the previous works. While traditional approaches
scan the root-set entirely before normal evacuation, scan and evacuation processes are performed to each task one after another as illustrated in Figure 5 line 28~39 because the contents of stack may change by each task instance. This helps to reduce the floating garbage by scanning the tasks in proper order, such as longest-period-first. Since the shorter the period of a task the higher the possibility of mutation, longest-period-first stack scanning can reduce the additional evacuation and floating garbage. The scanning of global variables is delayed as late as possible. Since the global variables tend to be shared and modified by multiple mutators, lazy scanning can reduce the overhead of barrier processing.

In order to reduce the asynchronous evacuation overhead caused by write-barrier, this paper now attempts to modify the technique presented in [7]. The modified garbage collector performs the asynchronous evacuation after the normal evacuation process. By maintaining temporary Update Entry, the actual evacuation can be delayed. All pointer update operations cause asynchronous evacuation operation to record the modification into the Update Entry. The Update Entry is checked, and the remaining entries are evacuated at the end of \( \Phi_k \). Additional space for Update Entry should be bounded. For details, refer to Figure 5 line 7~21.

Unlike stop-and-go collectors, incremental copying collectors can be triggered before the mutator initializes the object and thus misinterprets the obsolete pointers [13]. This pointer misinterpretation may result in memory leak. Therefore, proper memory initialization is important for robustness of application. Instead of initializing the heap incrementally, the modified garbage collector initializes new tospace right after the flip. The initialization time can be reduced with an efficient hardware support. One of the safest ways of initialization is to fill the memory with zero. In typical mid-sized embedded systems, bus frequency is usually less than 50 MHz. Zero-fill operation runs up to about 190 Mbytes/sec as peak performance under well-designed synchronous DRAM (SDRAM) memory. However, this level of performance is not sufficient for atomic memory initialization. Ultra-fast zero-fill operation can be achieved with cost-effective hardware support using synchronous graphic RAM (SGRAM). SGRAM is fully upward compatible with SDRAM, and has more features for graphics functions. Ultra-fast zero-fill operation utilizes eight-column block write capability which is done by internal color register and mask register. SGRAM transfers color register data to eight DRAM columns in one clock cycle unless mask register specifies inhibition. Since 50MHz bus clock period is equal to the minimum block write command cycle, the peak performance is up to about 1.5 Gbytes/sec zero-fill operations; it initializes 32 Kbytes heap area in 20 \( \mu \text{sec} \). Since most
real-time tasks have periods of over hundreds of microseconds and requires tens of kilobytes memory, the atomic memory initialization process can be completed at the flip time.

2.4 Estimation of $C_{GC}$

$C_{GC}$ is composed of the following four components: root-set scanning, normal object evacuation, barrier processing and memory initialization (zero-fill). $C_{GC}$ is given by

$$C_{GC} = c_1 R + c_2 L_{max} + c_3 E + c_4 M$$  \hspace{1cm} (7)

In Equation (7), each component relies on the properties of a given task set and root-set management scheme. These information can be given by a programmer or a compiler. $R$ denotes the maximum size of stack and global variables. The maximum stack size can be known a priori because it is specified when a task is created on modern embedded OS [16]. In the case of Java, the virtual machine specifies the maximum stack size. The stack size is generally small enough to bound the scanning overhead to a certain value. Other elements of root-set are global variables. They should be registered as root-set by a compiler and their maximum size should be given by a programmer. For more accurate estimation, well-designed off-line analysis is required. Coefficients $c_1$, $c_2$, $c_3$ and $c_4$ are determined by underlying run-time environment, such as hardware, OS or off-line compiler supports. For most cases, the implementation should concentrate on reducing the value of coefficients as does in the incremental approaches. Especially, $c_4$ can be greatly reduced by the hardware support described in Section 2.3.2.

3 Scheduling Algorithm

This section models a garbage collector as an aperiodic task and describes its scheduling. Based on the response time of a garbage collector, the amount of memory required to maintain stable system is presented. For convenience, $D_i$ is assumed to be equal to $T_i$ without loss of generality.

3.1 Memory Requirement

Since a garbage collector is triggered when the amount of available memory becomes less than a certain threshold, its release time is not known in advance. Thus, an aperiodic task with execution time of $C_{GC}$ represents the garbage collector. Since the system halts if $t_e > t_{s+1}$, the system should have free memory no less than the space requested by the mutator tasks during $R_{GC}$. The total memory requirement is twice the sum of the live memory and the total memory requirement of the mutator tasks during $R_{GC}$. The total memory requirement of $M_i$ is derived by Equations (1) and (2). The total system memory requirement is given by

$$M = 2 \left( \sum_{i=1}^{n} \pi_i A_i + L_{max} \right)$$  \hspace{1cm} (8)

In Equation (8), $\pi_i$ depends on the length of $R_{GC}$. Thus, if $R_{GC}$ is reduced through efficient scheduling of a garbage collector, the memory requirement can also be reduced.

The baseline memory requirement, denoted by $M_{min}$, is evaluated as follows:

$$M_{min} \geq M_{th} = 2 \left\{ \max \left( \sum_{i=1}^{n} \frac{H_{A_i}}{m}, \max(A_i) \right) + L_{max} \right\},$$  \hspace{1cm} (9)

where $m = \left\lfloor \frac{H - \sum_{i=1}^{n} H_{C_i}}{C_{GC}} \right\rfloor$.

Equation (9) assumes that $G$ can fully exploit idle time. However, it is usually impossible to distribute idle periods uniformly. Hence, $M_{th}$ is said to be a trivial baseline, but it still gives an useful information.

The objective of scheduling algorithm in this paper is to meet the deadlines of all the mutator tasks and to reduce the system memory requirement.

3.2 Scheduling $G$ and $M_i$’s

All the mutators are assumed to have their priorities according to the rate monotonic scheduling [12], in which the shorter the task period, the higher the task priority.

There are a number of approaches to the problem of jointly scheduling hard real-time periodic tasks (mutators) and aperiodic tasks (a garbage collector). The least effective way is to service the aperiodic tasks as background ones. In this approach, the priority of an aperiodic task is always the lowest, and it executes only when there is idle time. Background scheduling is also the least effective in terms of memory requirement since it maximizes $R_{GC}$. In this case, $R_{GC}$ is computed using the critical instant [12] at which all $M_i$’s are released.

One of the most interesting approaches, called slack stealing [11], is proved to be the optimal in terms of average response time for aperiodic workload. It attempts to make time for servicing aperiodic tasks by stealing all the processing time that can be obtained from the periodic tasks without causing deadline miss. However, it is difficult to apply it to the scheduling of a garbage collector for the following reasons. First, the worst-case response time for $G$ can hardly be computed off-line since the response time of aperiodic task depends on run-time behaviors. Secondly, the implementation overhead for the slack stealing algorithm is quite high. It requires additional memory space for off-line data structure and on-line overhead to compute available slack time for newly arriving aperiodic workload [11].

This paper intends to adopt a sporadic server [14] to service the garbage collection work. The sporadic server is a high priority task for servicing aperiodic tasks. It preserves its server execution time waiting for the arrival of an aperiodic task. Once an aperiodic task arrives in the meantime, it will be executed as long as the capacity of server permits; however, if it can not finish, it will continue to re-execute when the consumed execution time for the sporadic server is replenished. The replenishment of consumed execution time will occur one server period after the sporadic server first services the aperiodic task. Sporadic server mechanism has many advantages over other aperiodic task scheduling algorithms in the light of garbage collection. First, it has moderately low run-time overhead compared with the slack stealing. Secondly, available time for aperiodic workload is evenly distributed due to the
nature of sporadic server thus enabling the off-line estimation of \( R_{GC} \). Thirdly, in most cases, it can guarantee that at least one time slice is available for the garbage collector at \( t^k \) by assigning the highest priority to the sporadic server. This feature is very useful for the modified garbage collection algorithm depicted in Section 2. As mentioned in the previous section, memory initialization is essential to prevent possible pointer misinterpretation. Since, the modified garbage collector starts to initialize the new tospace area at the flip time, the time slices available at \( t^k \) is often large enough to complete the memory initialization.

The shortest period among all \( T_j \)'s is assigned to the sporadic server in order that it runs with the highest priority. Once the period of a sporadic server \( T_{SS} \) is determined, the server capacity can be obtained which can guarantee the schedulability of all \( M_i \). A necessary and sufficient condition for each task to meet its deadline under the rate monotonic scheduling [10] is used to compute the safe capacity of a sporadic server. Then, \( SS_{size} \) is given by

\[
SS_{size} = \min \{i = 1, ..., n\} \left\{ \left[ \frac{T_i}{T_{SS}} \right] \cdot \sum_{j=1}^{i} \left[ \frac{T_j}{T_i} \right] \cdot C_j \leq D_i \right\}
\]

(10)

### 3.3 Estimation of \( R_{GC} \) and \( \pi_i \)

With \( C_{GC}, SS_{size} \) and \( T_{SS} \) given, \( R_{GC} \) is computed as follows: As illustrated in Figure 6, if the release time of \( G_h \), \( t^h \), coincides with \( t^h \) and the remaining capacity of sporadic server is zero, the response time of \( G_h \) will be the worst-case response time \( R_{GC} \).

The response time of \( G_h \) is determined by the number of sporadic server periods required to complete garbage collection work and the remaining time for the next replenishment of server capacity. For example, \( R_{GC} \) in Figure 6 is computed as follows: When \( G \) is triggered, total capacity of the sporadic server has been consumed. Thus, \( G \) should delay its execution until the capacity of the server is replenished. Additional three sporadic server periods are required for \( G \) to complete its execution. Thus, \( R_{GC} = 29 \). Equation (11) summarizes the computation of \( R_{GC} \).

\[
R_{GC} = (T_{SS} - SS_{size} + \left( \left[ \frac{C_{GC}}{SS_{size}} \right] - 1 \right) \cdot T_{SS} + \left[ C_{GC} - \left( \left[ \frac{C_{GC}}{SS_{size}} \right] - 1 \right) \cdot SS_{size} \right])
\]

\[
+ C_{GC} = (T_{SS} - SS_{size}) + \left( \left[ \frac{C_{GC}}{SS_{size}} \right] - 1 \right) \cdot T_{SS} - SS_{size} + C_{GC}
\]

(11)

Given \( R_{GC} \), the worst-case number of active instances of \( M_i \) for a time interval \( R_{GC} \) can approximate \( \left[ \frac{R_{GC}}{T_i} \right] + 1 \). Since the above formula gives a rough upper bound, it requires further optimization of \( M_i \). It can be achieved using \( R_i \).

Consider the cases in Figure 7 which show the number of instances of \( M_i \) during \( R_{GC} \). Without loss of generality, \( M_{i,j} \) is assumed to be ready at time \( (j - 1)T_i \).

- Case (i): Let \( j = \left[ \frac{T_i^k}{T_i} \right] \) and denote the number of instances that have completed before \( T_i^k \). The instance depicted by case (i) is the \((j+1)^{th}\) instance of \( M_i \). If the condition \( jT_i + R_i \leq T_i^k \) holds, this instance has already completed its execution. Otherwise, this instance is regarded as active. The term, \( f(i) \) in Equation (12) describes this condition.

- Case (ii): It is clear that the \((j+1)^{th}\) instance of \( M_i \) is active for all \( M_{i,j} \) such that \( \left[ \frac{T_i^k}{T_i} \right] + 1 \leq j < \left[ \frac{T_i^k + R_{GC}}{T_i} \right] - 1 \), since the release and completion time of this instance is completely included in the garbage collection cycle.

- Case (iii): Let \( j = \left[ \frac{T_i^k + R_{GC}}{T_i} \right] \). Since, the release time of \( M_{i,j+1} \) comes earlier than \( T_i^k \), the \((j+1)^{th}\) instance of \( M_i \) is regarded as active.

Equation (12) implies \( \pi_i \) as follows:

\[
\pi_i = \left[ \frac{R_{GC}}{T_i} \right] + f(i), \text{ where } f(i) = \begin{cases} 1 & \text{if } R_i \geq T_i - \left[ \frac{R_{GC}}{T_i} \right] T_i + 1) \\ 0 & \text{otherwise} \end{cases}
\]

(12)

Estimation of \( R_{GC} \) and \( \pi_i \) enables to compute the worst-case memory requirement of the proposed scheduling policy. If it is possible to estimate \( R_{GC} \) and \( \pi_i \) properly, memory requirement for background and other scheduling policies can be computed similarly.

### 4 Performance Evaluation

This section addresses the memory requirement of the new approach against the traditional background method and other aperiodic scheduling disciplines. A major issue of interest is to estimate \( R_{GC} \) and compare the worst-case memory requirement with those of alternative scheduling algorithms.
To prove the higher efficiency of the algorithm, four sets of periodic tasks are chosen. All of them are synthetic task sets. Task sets TS1 and TS2 shown in Table 3 consist of 4 tasks, with total periodic utilization of 68.3% and 84.0%, respectively. Task sets TS3 and TS4 in Table 4 consist of 6 tasks with somewhat higher average memory allocation requirement than TS1 and TS2. Note that the memory allocation requirement of high priority tasks in Table 4 is higher than TS1 and TS2.

Table 3: Task set TS1 ($U = 68.3\%$) and TS2 ($U = 84.0\%$)

<table>
<thead>
<tr>
<th>Task</th>
<th>$C_1$</th>
<th>$T_1$</th>
<th>$D_1$</th>
<th>$A_1$</th>
<th>$\alpha_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>2</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>488</td>
</tr>
<tr>
<td>$T_2$</td>
<td>4</td>
<td>6</td>
<td>30</td>
<td>30</td>
<td>528</td>
</tr>
<tr>
<td>$T_3$</td>
<td>10</td>
<td>13</td>
<td>50</td>
<td>50</td>
<td>800</td>
</tr>
<tr>
<td>$T_4$</td>
<td>15</td>
<td>18</td>
<td>100</td>
<td>100</td>
<td>1296</td>
</tr>
</tbody>
</table>

Table 4: Task set TS3 ($U = 69.2\%$) and TS4 ($U = 88.2\%$)

<table>
<thead>
<tr>
<th>Task</th>
<th>$C_1$</th>
<th>$T_1$</th>
<th>$D_1$</th>
<th>$A_1$</th>
<th>$\alpha_1$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_1$</td>
<td>4</td>
<td>5</td>
<td>10</td>
<td>10</td>
<td>600</td>
</tr>
<tr>
<td>$T_2$</td>
<td>6</td>
<td>8</td>
<td>50</td>
<td>50</td>
<td>520</td>
</tr>
<tr>
<td>$T_3$</td>
<td>8</td>
<td>10</td>
<td>100</td>
<td>100</td>
<td>1980</td>
</tr>
<tr>
<td>$T_4$</td>
<td>10</td>
<td>13</td>
<td>200</td>
<td>200</td>
<td>3200</td>
</tr>
<tr>
<td>$T_5$</td>
<td>5</td>
<td>8</td>
<td>300</td>
<td>300</td>
<td>1040</td>
</tr>
<tr>
<td>$T_6$</td>
<td>13</td>
<td>18</td>
<td>600</td>
<td>600</td>
<td>4400</td>
</tr>
</tbody>
</table>

Figure 8: Feasible memory requirement of TS1 and TS2: (a) Background (b) Slack Stealing (c) Polling Server (d) New Algorithm

Estimation of $C_{GC}$ is performed by Equation (7) of which coefficients are derived from a static measurement of the garbage collector running on 50 MHz MPC860 processor with SGRAM. As mentioned in Section 2, the memory initialization can be done at a rate of 1.5 Gbytes/sec with hardware support. The estimation of $C_{GC}$ for TS1 in Table 3 is 3 ms and this value is deemed appropriate compared with the results in [7].

Equation (8) implies the worst-case memory requirement for a given task set. For the given workload, the new algorithm can achieve about 14–38% reduction in the worst-case memory requirement against the background approach. Table 5 summarizes the results.

Table 5: Worst-case memory requirement of four sets of tasks

Another interest is the feasible memory requirement of each scheduling algorithms ranging parameters such as utilization and memory allocation requirement of tasks. A feasible memory requirement means that there is no deadlock due to memory starvation and the schedulability of all the $\mathcal{M}_i$s and $\mathcal{G}$ are guaranteed under the given amount of heap memory. As one would expect, the feasible memory requirement is smaller than the worst-case mem-
suspends itself until the next period, and it does not preserve the
both the worst-case memory requirement and the
bandwidth allocated for aperiodic workload. In all simulations,
per a time slice, typically 1 ms. Although the uniform allocation
requirement considerably.

This paper also conducts a series of simulations to compare the
scheduling policies. The period and capacity of a polling server are
behavior can not fully realize the actual memory allocation behav-
or the real-time workload, it is sufficient to differentiate alternative
scheduling policies. The period and capacity of a polling server are
assumed to be equal to those of the corresponding sporadic server
in the proposed algorithm.

Figures 8 and 9 present memory allocation behavior of periodic
tasks and the feasible memory requirement of each scheduling al-
gorithm for two pairs of task set, (TS1, TS2) and (TS3, TS4). In
general, a significant performance degradation is observed as the
utilization of task set increases. The result comes from the fact that
higher utilization gives rise to lower server capacity.

A series of simulations demonstrate that the new algorithm outper-
forms background scheduling and can produce the feasible mem-
ory requirement comparable to the complex on-line algorithm, in
most cases. Background scheduling exhibits the worst perfor-
mance, whereas the performance of the proposed algorithm and the
polling server are the best. Although the polling server algorithm
shows the performance comparable to the proposed algorithm, its
response time for $G$ is slightly worse. Furthermore, the proposed
can allocate at least one time slice to a garbage collector
immediately after the flip in most cases thus reducing the memory
initialization of fromspace mentioned in the previous sections.

The results of simulations on TS3 and TS4, which are worth pay-
careful attention, are shown in Figure 9. Readers would expect
that slack stealing would tend to show the best performance in the
memory requirement. However, noticeable differences in perfor-
ance between the proposed algorithm and the slack stealing are
observed under high utilization and memory requirement of short-
period tasks. Regardless of the total utilization, the proposed algo-
shows performance enhancement up to 44% in the memory
requirement against the slack stealing. Since the slack stealing ag-
gressively uses idle time, there will be little idle time left around the
end of the hyperperiod or even in the middle of the hyperperiod. If
a garbage collection cycle spans over the successive hyperperiods,
the slack stealing algorithm often causes $G$ to miss its deadline.

Thus, it can be remarked that the memory requirement of given set
of tasks depends primarily on the characteristics of memory allo-
cation behavior and the amount of memory needed by each task.
In Figure 9, all of the response times of $G_k$ ($k \geq 1$) for TS4 in
the proposed algorithm are identical, 13 ms. By contrast, the slack
stealing algorithm yields the response times ranging from 6 ms to
37 ms. The simulation results show that the proposed algorithm is
robust in the memory allocation behavior compared with the

![Figure 9: Feasible memory requirement of TS3 and TS4](image)

(a) $T_{TS3} = 22300, T_{TS4} = 31500$

(b) $T_{TS3} = 9800, T_{TS4} = 14500$

(c) $T_{TS3} = 5900, T_{TS4} = 11500$

(d) $T_{TS3} = 5000, T_{TS4} = 11500$
slack stealing and reduces the deviation of response time of G.

5 Conclusions and Future Work

This paper addresses the problem of modeling a garbage collector as real-time task in embedded real-time systems and proposes a new algorithm to schedule both a garbage collector and real-time mutator tasks. A modified garbage collector for real-time systems is presented as well. Using the sporadic server approach, the scheduling algorithm can reduce both the worst-case and feasible memory requirement while guaranteeing the schedulability of all the real-time mutator tasks. Performance evaluation demonstrates that the proposed algorithm can considerably reduce the worst-case memory requirement against the background scheduling. The algorithm can also produce the feasible memory requirement comparable to the complex on-line algorithm such as slack stealing. Furthermore, it outperforms slack stealing under high utilization and memory requirement of short-period tasks. Reducing the system memory requirement, the price/performance competitiveness of underlying embedded system can be enhanced.

The worst-case memory requirement derived in this paper is somewhat overestimated because it covers arbitrary memory allocation behavior. Proper manipulation of the memory allocation behavior will improve the accuracy of the worst-case memory requirement. Copying collector shows relatively deterministic behavior than other tracing collectors and thus is suitable for real-time applications. However, its memory utilization and copying overhead can be drawbacks for embedded real-time systems at the same time. Future work will focus on other tracing collectors for embedded real-time systems.

References


