Garbage Collection Scheduling of Aperiodic Tasks

Ning Zhang and Guang-Ze Xiong

Abstract—In the previous work of garbage collection (GC) models, scheduling analysis was given based on an assumption that there were no aperiodic mutator tasks. However, it is not true in practical real-time systems. The GC algorithm which can schedule aperiodic tasks is proposed, and the variance of live memory is analyzed. In this algorithm, active tasks are deferred to be processed by GC until the states of tasks become inactive, and the saved sporadic server time can be used to schedule aperiodic tasks. Scheduling the sample task sets demonstrates that this algorithm in this paper can schedule aperiodic tasks and decrease GC work. Thus, the GC algorithm proposed is more flexible and portable.

Index Terms—Aperiodic tasks, garbage collector, real-time, scheduling.

1. Introduction

Garbage collection overcomes the potential danger of manual memory management, such as memory leaks, dangling pointers, and fragmentation. However, traditional ‘stop-the-world’ garbage collectors which are used in JavaTM[1] and Microsoft’s C# halt the application until the GC cycle end, and then the reclaimed memory is available. This is clearly unsuitable for use in real-time systems.

To meet real-time requirement, incremental algorithms[2] split GC work into many small increments. The work of the work-based GC is performed in small increments triggered by allocation requests. Even though each increment is so small, a burst of allocation requests can cause the worst case, in which the response times of real-time tasks are too long to be compatible with hard real-time systems.

In recent years, to satisfy the hard real-time requirement, a trend is use time-based approaches more than the work-based approaches[3]. With time-based scheduling a garbage collector is treated as a periodic or aperiodic task and scheduled concurrently with other tasks in the system. Scheduling parameters of GC such as the worst case execution time and the worst case response time must be determined by off-line analysis, so that deadlines of mutators are always met and mutators will not run out of memory when scheduled. Henriksson[4], Bacon[5], Kim[6][7] and Robertz[8] et al. gave typical state-of-the-art concurrent garbage collections.

Previous work mostly gave scheduling analysis based on an assumption that there are no aperiodic mutator tasks. However, it is not true in practical real-time systems. In our GC model, aperiodic tasks are included and they also can be scheduled in our garbage collection. In an aperiodic scheduling technique, sporadic server is used. Different from Kim’s strategy[9], garbage collection can be dynamically executed to reduce actual GC work and execution time. The idle time saved by GC can be served to aperiodic tasks. Both of aperiodic tasks and GC can be executed in sporadic server. So, the proposed GC algorithm can make the system more flexible and portable.

2. Concurrent Scheduling GC

Baker’s incremental copying collector[2] represents traditional real-time garbage collection which is called work-based garbage collector[10]. Since work-based garbage collectors can not guarantee that the deadlines of real-time tasks are always met, they are not compatible with hard real-time systems.

In recent years, time-based real-time GC (or concurrent GC algorithm) has been proposed to meet hard real-time requirement. Garbage collector is treated as a single real-time task to be scheduled with other mutators. Therefore, real-time tasks will not be disturbed by garbage collection when executed and hard deadlines are guaranteed.

In semi-concurrent GC[4], GC got running when there was no high priority task needed executing. Thus, the system memory space requirement was much larger than any of the following concurrent GC algorithms. Bacon et al. used an incremental mark-sweep algorithm with partial on-demand compaction[5]. This approach provided a consistent utilization of mutator tasks and limited space overhead since from-space and to-space are mostly sharing physical storage. Kim et al. designed a concurrent GC based on a sporadic server (SS)[6][7]. A sporadic server was used to serve the needs of GC and the period was the shortest among all tasks. This algorithm showed better performance on reducing the system memory requirement.
than Henriksson’s background approach. However, in Kim’s GC model, aperiodic tasks could not use the capacity of the sporadic server even though the garbage collection was not in progress. Robertz et al. presented time-triggered garbage collection\cite{8} based on assigning the collector a deadline for when it must complete its current cycle.

3. System Model

A copying collector is chosen for the discussion because it can simplify the estimation of garbage collection time. Furthermore, incremental copying algorithm is easy to be implemented. It also has a main advantage that when copying garbage collector reclaims the unused memory, compaction is included. Thus it avoids the problems resulted from fragmentation. When GC is triggered and flip happens, the mutators allocate objects in to-space and GC moves live objects which are in from-space (that is ‘to-space’ where objects are allocated before flip) to to-space, as shown in Fig. 1.

Consider a real-time system with a set of periodic mutator tasks $M = \{M_1, M_2, \ldots, M_i\}$. The following symbols will be used in this paper.

- $G_k$: the $k$th garbage collection request.
- $G_{ki}$: the $i$th time slice of $G_k$.
- $L_k, L^*_k$: amount of live memory processed by $G_k$ and its maximum value.
- $N$: memory requirement of system.
- $T_i$: period time of mutators.
- $C_i$: execution time of mutators.
- $D_i$: deadline of mutators.
- $A_i$: maximum amount of memory allocated by $M_i$ during $T_i$.
- $\alpha_i$: portion of live objects out of $A_i$ when $M_i$ is inactive.
- $C_{GC}$: the worst case execution time of GC.
- $T_{GC}$: period time of GC.
- $R_{GC}$: response time of GC.
- $M_i$: is characterized by a tuple $M_i = \{C_i, T_i, D_i, A_i, \alpha_i\}$.
- $M_{ij}$: the $j$th routine of mutator $M_i$.

The discussion is based on the following assumptions that the context switching and task scheduling overhead are negligible. There is no precedence relations and no blocking factor among the tasks. $C_i, T_i, D_i, A_i, \alpha_i$ are known a priori and the deadline of each mutator task is equal to its period ($D_i = T_i$).

We use Kim’s sporadic server approach to schedule GC. When triggered, GC is scheduled as an aperiodic task, which is assigned to the highest priority. This algorithm makes GC be evenly distributed thus enable the off-line estimation of $R_{GC}$.

Fig. 2 shows the worst-case response time of GC when GC is scheduled by sporadic server approach. In many researches, $C_{GC}$ is estimated off-line and is regarded as a fixed value with trivial upper bound to calculate the worst-case response time of GC and required memory space. In Fig. 2, $C_{GC}$ is estimated equal to 5, thus the worst-case response time of GC is 29. However, $C_{GC}$ is not fixed but is dynamically changed based on execution of mutators. Consequently, $C_{GC}$ should be estimated more precise based on the actual execution.

4. Algorithm Description

4.1 Analysis of Live Objects

The new GC cycle begins instantly when the current GC cycle is ended. The formula (1) can be got from [9] and [11], where $\pi_i$ is the worst case instances of $M_i$ during $R_{GC}$

$$
N = 2 \left( \sum_{i=1}^{n} \pi_i A_i + L^*_k \right). \quad (1)
$$

Equation (1) shows that the memory reservation depends on $R_{GC}$, which is determined by $C_{GC}$ when parameters of other real-time tasks are known priori.

Kim et al. calculated $C_{GC}$ of copying garbage collection as\cite{9}

$$
C_{GC} = c_1 \left( \frac{R + L^*_x}{{\text{sizeof(word)}}} \right) + c_2 L^*_x + c_3 \epsilon + c_4 N. \quad (2)
$$

Equation (2) indicates that $L^*_x$ is the dominant factor to determine $C_{GC}$. Thus, decreasing value of $L^*_x$ will decrease GC workload and decrease $C_{GC}$ accordingly.

![Fig. 1. Diagram for processing procedure of copying garbage collector.](image1)

![Fig. 2. The worst-case response time of GC ($T_x=10$, $C_x=2$, $C_{GC}=5$).](image2)
Table 1: Example task set: $T_i = 10$, $C_i = 3$

<table>
<thead>
<tr>
<th>Task</th>
<th>$C_i$ (ms)</th>
<th>$T_i$ (ms)</th>
<th>$D_i$ (ms)</th>
<th>$A_i$ (byte)</th>
<th>$\alpha_i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_1$</td>
<td>2</td>
<td>10</td>
<td>10</td>
<td>988</td>
<td>0.43</td>
</tr>
<tr>
<td>$M_2$</td>
<td>4</td>
<td>30</td>
<td>30</td>
<td>1028</td>
<td>0.36</td>
</tr>
<tr>
<td>$M_3$</td>
<td>10</td>
<td>60</td>
<td>60</td>
<td>1200</td>
<td>0.38</td>
</tr>
<tr>
<td>$M_4$</td>
<td>15</td>
<td>120</td>
<td>200</td>
<td>1696</td>
<td>0.27</td>
</tr>
</tbody>
</table>

As described by Kim et al.\cite{Kim}, for periodic real-time tasks, the live memory for each $M_i$ depends on the state of each task. Periodic tasks have two states that are active state and inactive state. When GC is triggered, the amount of live objects of inactive tasks is stabilized as $A_i\alpha_i$, while that of active tasks is $A_i$. Kim et al.\cite{Kim} proposed the formula

$$L_k = \sum_{M_j \text{active} (t'_j)} A_i + \sum_{M_j \text{inactive} (t'_j)} \alpha_i A_j.$$  \hspace{1cm} (3)

Kim found the active windows and the transitive preemption windows. The combination of all the preemption windows, which maximizes the amount of local live memory, can be found. Consequently, $L_k^*$ can be calculated under such situation.

The example task set given in \cite{Kim} is showed in Table 1. Take $M_1 \Rightarrow M_3 \Rightarrow M_4$ as the very combination example. In his case, $M_2$ is inactive and other tasks are active. The amount of live objects of $M_2$ is $A_2\alpha_2$, and the amount of live objects of other tasks is $A_1$, $A_3$ and $A_4$ respectively. $L_k$ is $A_1 + A_2\alpha_2 + A_3 + A_4$ and it is reduced by up to 13% compared with the trivial bound.

However, in the GC cycle, the amount of live objects in from-space varies with the running of tasks. Generally speaking, the amount of live objects decreases when mutator is running. When mutator becomes inactive, the amount of its live objects is stabilized as $A_i\alpha_i$. As known, GC is interleaved with mutators. According to variance of live objects described in previous study, GC is required to reclaim each task’s memory space one after another. GC should process inactive tasks first since the amount of live objects of inactive task is stabilized as $A_i\alpha_i$. After that, the amount of live objects of active tasks may be decreased when the task is running. Active tasks also may change to inactive tasks, then the amount of live objects retained to GC is decreased as $A_i\alpha_i$. GC can move as little floating garbage as possible by using this approach.

Fig. 3 shows the sample case. When GC is triggered, $M_{1,\text{in}}$ has finished its execution while $M_{1,j}$ has not. So $M_0$ is inactive and the amount of live objects is $A_i\alpha_i$, while $M_i$ is active and the upper bound of amount of live objects is $A_i$. Based on previous discussion, GC first reclaims memory space of $M_k$ in its first time slice $G_{i1}$. $M_{i,j}$ is delayed to be processed by GC in $G_{i2}$. At that time $M_{i,j}$ has completed its execution and becomes inactive. Thus the total amount of live objects which GC should process is $A_i\alpha_i + A_i\alpha_i$. Otherwise, if GC first reclaims memory space of $M_1$, the amount of worst case live objects is $A_i$ since $M_i$ is active when GC is just triggered. Then the total amount of live objects which GC should process is $A_i\alpha_i + A_i$, which is increased compared with that computed before.

4.2 Scheduling Aperiodic Tasks

Garbage collection can decrease GC work, and save capacity of sporadic server through dynamically processing tasks’ memory space. In our algorithm, the saved capacity can be used for scheduling aperiodic tasks. So, both GC and aperiodic tasks can use sporadic server. Furthermore, whether sporadic server is occupied by GC or aperiodic tasks is dependent on the state of real-time task which GC will process. GC should process memory space of each task one by one, with inactive tasks first and active tasks later. When the state of task, which will be processed, is active, the sporadic server is scheduled to server aperiodic tasks. Since the processed tasks may become inactive when the next server period comes, garbage collection can reduce floating garbage and minimize its workload.

For example, we now investigate the situation $M_1 \Rightarrow M_3 \Rightarrow M_4$ of Table 2 as shown in \cite{Kim}. When GC is triggered, $A_1$, $A_3$, and $A_4$ is active while $A_2$ is inactive. Then GC first reclaims memory of $M_2$ in its first slice [70,73]. When it comes to GC’s second slice [80,83], $M_1$ becomes inactive. After GC reclaiming memory of $M_1$, window is [66,89] while the third slice of GC is [90,93]. However, the fourth slice of GC is [100,103], which is $M_3$. 

Fig. 3. Sample case of tasks interleaved with GC.

Fig. 4. Scheduling aperiodic tasks.
also becomes inactive because its current active overlapped with active window of $M_4$. Since $M_4$ is active when GC will process it, the sporadic server, that is, time slice [100,103] is set to server aperiodic tasks. When the fifth slice of GC [110,113] comes, $M_4$ becomes inactive since the active window of $M_4$ is [18,106]. So memory of $M_4$ that GC will process is $A_4a_4$, which is minimized compared with $A_4$ if GC process $M_4$ at the fourth slice as in [11]. If $C_{GC}$ is 40, the saved time slice is 10 which can be used for scheduling aperiodic tasks.

Fig. 4 shows the description above. Black rectangles represent GC time slices, while grey rectangles denote aperiodic tasks time slices. When the state of processed task is active, the sporadic server (the fourth slice of Fig. 4) is used to schedule aperiodic tasks. GC processes the memory space until the task becomes inactive. From Fig. 4, it can be found that the proposed algorithm can not only schedule aperiodic tasks but also decrease the response time of GC since processing inactive tasks first can reduce GC workload compared with processing active tasks first. In proposed garbage collection actual $C_{GC}$ is 14 and $R_{GC}$ is 52, while in garbage collection of [11] actual $C_{GC}$ is 24 and $R_{GC}$ is 73.

5. Conclusions

Not like ‘stop-the-world’ GC in Java[12] and other garbage collections, the proposed method treats GC cycle as a dynamic process and $C_{GC}$ is not treated as a fixed value. GC should process memory space of each task one by one, with inactive tasks first and active tasks later. Furthermore, different from the previous research work, the algorithm in this paper can schedule aperiodic tasks. Sporadic server is used by aperiodic tasks when the processed real-time task is active. Then GC is scheduled again when active task become inactive. Changing of state of task processed can decrease GC workload and save more server time for aperiodic tasks. Scheduling sample task sets shows that the proposed garbage collection is effective and flexible.

Though the proposed GC can schedule aperiodic tasks, scheduling analysis and memory analysis need to be executed more precisely for hard real-time systems. Future work will focus on it and the implementation of our garbage collector model.

Acknowledgment

The authors would like to express great appreciation to the adviser professor Guang-Ze Xiong for his hard work. Thanks to Ph.D. Jian Lei, Xia Yang, Wei Jiang, and Zheng-Wei Chang for their comments and suggestions.

References


Ning Zhang was born in Jiangsu Province, China, in 1975. He received M.S. degree in computer science from University of Electronic Science and Technology of China (UESTC). He is now pursuing his Ph.D. degree with UESTC. His research interests include real-time systems and garbage collection.

Guang-Ze Xiong was born in 1938. He is currently a professor and tutor of Ph.D. with School of Computer Science and Engineering, UESTC. His research interests include real-time operating system, real-time computing, and dependable computing.