Integrating Hybrid Garbage Collection with Dual Priority Scheduling

Yang Chang and Andy Wellings
Real-time Systems Research Group
Department of Computer Science
University of York, York YO10 5DD, UK
{yang, andy}@cs.york.ac.uk

Abstract

In this paper, we propose an approach to integrate a hybrid garbage collection algorithm (a combination of reference counting and mark-and-sweep techniques) into the current response time analysis framework for real-time systems. Instead of collecting garbage incrementally, we put most GC work into a periodic real-time thread, namely the GC thread, which is scheduled according to the dual priority scheduling method. More importantly, we can perform schedulability analysis (response time analysis) for all the real-time threads including the GC thread.

1 Introduction

As an automatic memory management technique, garbage collection is crucial for modern programming languages (e.g. Java). Although there have been improvements in real-time garbage collection techniques in recent years, they are still considered inappropriate for many types of systems, particularly hard real-time areas. The goal of this work is to address some of the outstanding problem areas.

Instead of performing pure tracing or reference counting, we propose a hybrid approach which combines both reference counting and mark-and-sweep. The reasons why we choose such a hybrid algorithm are:

First, in most tracing garbage collection algorithms, the collectors can only reproduce free memory either at the end or at the beginning of a GC cycle. That is, the amount of memory that is available to the mutator threads is monotonically decreasing (i.e. no replenishment) within one GC cycle. Therefore, the total amount of allocation in that cycle must be bounded. Based on such bound, the deadline of a GC cycle can be calculated. However, as we will discuss soon, this deadline can be much shorter than that of a hybrid algorithm. Consequently, the GC thread could be invoked much more frequently if a pure tracing algorithm is adopted.

Another problem of tracing collection is that all the live objects have to be processed at least once in each GC cycle. In our opinion, this is a waste of system resources if not many garbage objects are generated, particularly when the GC thread has a very short deadline and period.

On the other hand, reference counting algorithms are capable of reclaiming individual garbage object soon after it emerges. Therefore, the amount of memory available to the mutator threads is no longer monotonically decreasing. The algorithm has virtually no limitation on the amount of allocation. Now, the issue turns to be how to synchronize reclamations and allocations, that is, how to guarantee that enough memory can always be reclaimed before the mutator threads require allocation.

Unfortunately, reference counting algorithms cannot reclaim cyclic garbage per se. However, by combining reference counting with mark-and-sweep collector, cyclic garbage can be found by a single tracing collection in each GC cycle while all the acyclic garbage objects can be reclaimed in the GC cycles where they are generated. Although there is more work to do in a GC cycle in our approach compared to the tracing algorithms, our GC cycles often have much longer lengths. This is because the period and the deadline (they equal each other in our system) of our GC thread are mainly determined by the speed of cyclic garbage cumulation rather than the speed of allocation. This will be justified in section 5.

By combining the two approaches, we can also eliminate the root scanning phase and therefore the synchronization points [5] since reference counting can provide enough information to the mark-and-sweep collector. This is very important for real-time systems because root scanning can prevent the mutator threads from executing and even for incremental root scanning, the context switch time could be unacceptable due to the limitation on the number of synchronization points. For further information about how to combine the two distinct approaches, please see the full version of this paper [1].

When it comes to scheduling, we propose a segregated

GC thread which is actually a periodic real-time thread. By doing so, we can integrate garbage collection into the current response time analysis framework to see whether the whole system (including GC) is schedulable prior to the execution. Because garbage collection is not a user function of the system, we hope to schedule it as infrequently as possible and also as late as possible. For this purpose, we decide to use dual priority scheduling method [2] to schedule the GC thread.

This paper is organized as follows: Section 2 briefly introduces the related work and describes our contributions. Section 3 presents our hybrid garbage collection algorithm. Section 4 describes how to schedule the GC thread according to the dual priority scheduling method. Section 5 analyzes our system for GC thread's parameters such as priority, deadline, F_{pre}, WCET (worst case execution time) and worst case response time. Finally, we present our conclusion and future work.

2 Related Work

Instead of using a segregated thread to collect garbage, Kim [3] introduced aperiodic server techniques to garbage collection. In his approach, a sporadic server with the highest priority is introduced to execute a copying garbage collector. To provide temporal and spacial guarantees offline, the worst case length of a GC cycle is calculated. How much memory, in the worst case, we need for any GC cycle can then be decided.

Kim's approach is based on copying collection technique which has high algorithm complexity in both temporal and spacial aspects due to the nature of copying. Moreover, the whole system will fail when the garbage collection work misses its deadline since there won't be any free memory for garbage collector and the mutator threads. Fundamentally, this is because both garbage collector and mutator threads are memory consumer. By contrast, only the mutator threads can consume free memory in our approach so even if our garbage collector misses its deadline, the whole system can still function correctly except that some mutator threads may suffer from further blocking (perhaps miss deadline) because of the lack of free memory. Hence, the system gracefully degrades.

3 Basic Algorithm

In our approach, the GC thread does mainly two things: performing mark-and-sweep and processing the to-be-free-list which contains all acyclic garbage and cyclic garbage that has been recognized. Whenever the to-be-free-list is not empty, the GC thread will traverse it and reclaim the garbage objects. On the other hand, mark-and-sweep can only be performed when the GC thread has nothing to do. Notice that the execution of mark-and-sweep can also be "preempted" by the processing of the to-be-free-list. Such a scenario is illustrated in figure1.

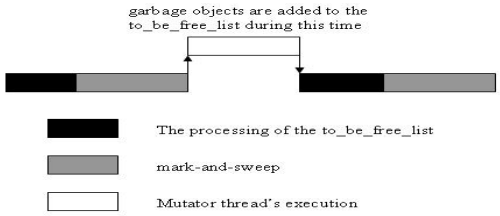

Figure 1. how mark-and-sweep is preempted

When the mark-and-sweep work is completed, the current release of the GC thread ends. (For further information, please see [1])

4 Scheduling

As discussed previously, we require that the GC thread should be scheduled according to dual priority scheduling algorithm. Applying this technique to the GC thread has the consequence that the GC thread always gives way to the mutator threads as long as it can still meet its deadline (i.e. it can reclaim enough memory for the mutator threads). We define the promotion time as the release time plus the difference between the period and the worst case response time of the GC thread. Before the promotion time, the GC thread is executed at the lowest priority of the whole system. Otherwise, it is executed at a priority given by the system designer (We will introduce how to calculate priority for the GC thread in section 5). If however the GC thread is ever activated before its promotion time has elapsed, then the promotion time should be increased by the length of that activation period. That is, the length of time when the GC thread is waiting at the lowest priority is fixed for any release. Currently, if a release of the GC thread follows its worst case path, that release must be completed at exactly the deadline and the GC thread is released again immediately.

So far, there are still two problems to be resolved. First, since the GC thread can have a real response time which is smaller than the worst case one, it can be completed before the deadline and also the period. Therefore, there could be a time interval during which no garbage collection work can be performed. We simply cannot preserve enough memory for such time intervals because we cannot predict the lengths of such time intervals. Our solution is to require that the GC thread should be released again immediately on its completion. That is, in our approach, the period of the

GC thread is changeable (see figure 2). In the full version of this paper [1], we have proved that this cannot ruin the schedulability of the real-time threads with lower priorities than the GC thread.

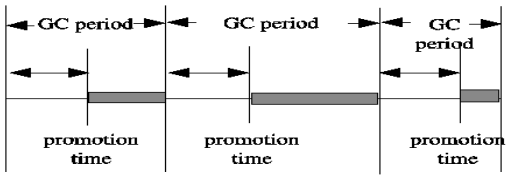

Figure 2. The period of the GC thread is changeable

The second problem is how to guarantee that the real-time threads with higher priorities than the GC thread (lowest/promoted) can never be blocked by the GC thread because of the lack of free memory. Notice that meeting the deadline of the GC thread doesn't necessarily mean that the mutator's memory requirements can always be satisfied because the mutator threads can sometimes consume all the free memory before the GC thread can reclaim enough garbage objects although it may reclaim enough or even more in the near future. In our approach, we require that before the promotion time, as long as the amount of free memory is lower than a certain value called F_{pre} (We will introduce how to calculate this value in section 5), the priority of the GC thread should be promoted. Otherwise, it should be executed at the lowest priority until the promotion time. By doing this, we always preserve enough free memory for the real-time threads with higher priorities than the GC thread (promoted).

5 Static Analysis

In this section, we calculate the parameters for the GC thread. Since our GC thread is a periodic real-time thread (albeit with varying period), we can divide the whole time line into several blocks each of which corresponds to a GC cycle, that is, a release of the GC thread. Some symbols can then be defined in table 1.

Due to the limitation on space, we discuss our analysis process without proof in this paper. However, you can find all those proofs in [1].

The formula which can be used to calculate GC thread's deadline D is given below:

$$\sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \leq \frac{H - L_{max} - F_{pre}}{3} + CG_{min} \quad (1)$$

With this formula, we can justify what we claimed before, i.e. our GC work's deadline depends on the speed of cyclic garbage cumulation rather than the speed of allocation.

Also, we present how to calculate F_{pre}:

$$R_{pre} = \sum_{j \in hp(GC)} \left[\left\lceil \frac{R_{pre}}{T_j} \right\rceil (C_j + RR \cdot a_j) \right] \quad (2)$$

$$F_{pre} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{pre}}{T_j} \right\rceil \cdot a_j \right) \quad (3)$$

Notice that to calculate R_{pre} and F_{pre}, we need to know the GC thread's priority in advance since we need to know who belongs to $hp(GC)$. However, if we adopt DMPO (deadline monotonic priority ordering) mechanism, we should have already known the deadline of the GC thread so there's obviously a recursion. In order to resolve this recursion, we need to treat equation 2, 3 and 1 as a group.

At the very beginning, assume that the GC thread has the lowest priority among all the real-time threads. Then, we can get the corresponding R_{pre}, F_{pre}, D and consequently the priority corresponding to the D. If the GC priority is the same as we assumed, that is the result. Otherwise, we should use the new GC priority to recalculate R_{pre}, F_{pre}, D and the new priority until the old version and the new version of the GC thread's priority equal each other. Unfortunately, this process sometimes cannot converge so we have to manually choose a priority for the GC thread which may violate the DMPO.

So far, we have already been able to calculate the WCET and the worst case response time of the GC thread: $WCET_{GC}$ and R_{total} respectively.

$$WCET_{GC} = RR \cdot CG_{max} + RR \cdot \sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot rg_j \right) + TR \cdot L_{max} \quad (4)$$

$$R_{total} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{total}}{T_j} \right\rceil \cdot C_j \right) + WCET_{GC} \quad (5)$$

As what happens in the normal response time analysis, we can now compare the GC thread's worst case response time R_{total} with its deadline D. If $R_{total} \leq D$, the GC thread is schedulable. Moreover, we can also use the parameters of the GC thread (e.g. the period, D and $WCET_{GC}$)to calculate the response time of the real-time threads with lower priorities than the GC thread. If their response time values are smaller than their deadlines respectively, they are also schedulable. Otherwise, the designer has to redesign the system to reduce either the memory usage or the WCET of some mutator thread.

Symbols	Definitions
P	the set of mutator threads in the whole system
L_{max}	the upper bound of live memory consumption of the whole system
H	heap size
$hp(GC)$	the set of all the threads with higher priorities than the GC thread (promoted)
CG_i/CG_{min}/CG_{max}	the amount of cyclic garbage found by the ith release of the GC thread and its minimum and maximum value
F_{pre}	the amount of free memory the system should preserve for the threads with higher priorities than the GC thread
R_{pre}	the worst case response time of the GC thread to reclaim as much memory as the mutator threads allocate during that time
T_j	the period of thread j
C_j	the worst case execution time of the thread j
a_j	the worst case memory allocation executed in one release of the thread j
cgg_j	the worst case amount of cyclic garbage generated in one release of thread j
rg_j	the worst case amount of acyclic garbage generated in a release of the thread j
RR	the time needed to reclaim one unit of memory (by reference counting)
TR	the time needed to trace one unit of memory (by mark-and-sweep collector)

Table 1. symbol definition

6 Conclusion and Future Work

In this paper, we described our collection algorithm, scheduling property and some analyses for the GC thread's parameters. We believe that the benefit of our approach includes:

- It is commonly accepted that reference counting garbage collection has many advantages [6, 4]. Our approach inherits all of them.
- We make reference counting and mark-and-sweep cooperate with each other. On the one hand, the occasionally invoked mark-and-sweep can help reference counting find cyclic garbage. On the other hand, reference counting can eliminate the root scanning phase for the mark-and-sweep collection and make it much less frequent so that a great amount of unnecessary system resource consumption is avoided.
- Our approach is flexible enough so that the GC thread can adapt to different applications automatically: the more the cyclic garbage is generated, the shorter the deadline could be. For a system which is mainly composed of acyclic data structures, the deadline of the GC thread could be very long so that our approach can be very efficient. However, for a system which is mainly composed of cyclic data structures, our approach gracefully degrades. Fortunately, the above analysis provides the designers a way to quantitatively determine whether our approach is suitable for their application or not.
- We can provide real-time guarantees for all the real-time threads as in non-garbage-collected real-time systems.

However, there are still some problems to solve. For example, object finalization is not concerned in this paper; correct and tight estimations of cgg_j and rg_j are not available so far. In the future, we will try to resolve these problems and modify our approach where necessary.

Currently, we are trying to implement our algorithm for further evaluation. The implementation will be built upon the basis of GCJ compiler (the GNU compiler for the Java language) and JRate library, which is a RTSJ-compliant real-time extension to the GCJ compiler.

References

[1] Y. Chang. Integrating hybrid garbage collection with dual priority scheduling. Technical Report YCS 388, University of York, 2005.

[2] R. Davis and A. Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposia*, pages 100–109. Real-Time Research Group, University of York, 1995.

[3] T. Kim, N. Chang, and H. Shin. Joint scheduling of garbage collector and hard real-time tasks for embedded applications. *Journal of Systems and Software*, 58(3):247–260, September 2001.

[4] T. Ritzau. Hard real-time reference counting without external fragmentation. In *Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001*. Linköping University, April 2001.

[5] F. Siebert. *Hard Realtime Garbage Collection in Modern Ojbect Oriented Programming Languages*. PhD thesis, University of Karlsruhe, May 2002.

[6] P. R. Wilson. Uniprocessor garbage collection techniques. In *Proceedings of International Workshop on Memory Management*. University of Texas, September 1992.