Integrating Hybrid Garbage Collection with Dual Priority Scheduling

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

Abstract

In this report, we propose an approach to integrate a hybrid garbage collection algorithm into the current response time analysis framework for real-time systems. In our approach, the collection algorithm is a combination of reference counting and mark-and-sweep techniques so we can keep the advantages of reference counting while eliminating cyclic garbage by invoking mark-and-sweep occasionally. 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. By performing static analysis, we can get the GC thread's parameters such as period, deadline, priority and worst case execution time. With these parameters, 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). It has been widely and successfully used in many general purpose computing systems. 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 [3]. The goal of this work is to address some of the outstanding problem areas.

Garbage collection techniques can be classified into 3 main categories: *Reference Counting*, *Tracing* and *Contaminated Garbage Collection* [6]. Among these algorithms, reference counting algorithms are normally more efficient and predictable. However, they cannot find any cyclic garbage automatically because of the internal references within the cyclic data structures. Consequently, attempts have been made to modify them to perform some local tracing on potential cyclic structures so that they can recognize cyclic garbage [12, 2]. Unfortunately, doing this can ruin the efficiency and make the algorithms much less predictable. Here, we propose to use a hybrid approach which is a combination of reference counting and mark-and-sweep (global tracing) (see section 3).

In a garbage collected system, the main factors that influence the real-time performance include not only the GC algorithm but also the way to dispatch the system resources (especially CPU time) between the garbage collector and the mutator.

Scheduling models for garbage collection can be classified into 3 categories: *Stop-the-World*, *Incremental* and *Concurrent*. Potential real-time GC scheduling properties can only come from the latter two categories. Furthermore, the incremental scheduling methods come into two flavors: *work-based* and *time-based*. In work-based incremental GC, a small amount of GC work will be attached to each allocation so that we can guarantee that the mutator cannot be preempted by the garbage collector for extended periods, and the GC work can be finished before the heap is exhausted provided that the ratio of the amount of GC work to the amount of the allocated memory is high enough. As noticed by Bacon [1] and Johnstone [11], in the work-based incremental systems, the mutator utilization[1] can become very low when the mutator performs allocations intensively. That is, the mutator cannot progress during that time so that some other threads may miss their deadlines. In order to cope with this problem, Bacon introduces a time-based incremental GC algorithm according to which the time line is divided into fixed-size pieces (according to a parameter given by the programmer). Each of these pieces is further divided into 2 parts. One is given to the collector and the other one is given to the mutator. How much time should be given to the mutator is determined by the minimum mutator utilization. Based on such a mechanism, consistent mutator utilization can be achieved. Notice that if a thread is released during a GC dedicated period, it has to be delayed until the next mutator dedicated period.

Intuitively, irrespective of work-based or time-based, none of the incremental GC algorithms can take advantage of idle time and a thread has to be delayed if it is released when GC is progressing even if it has the highest priority. Another vital issue is that for those real-time threads with very tight deadlines, adding the execution time of the GC increments to their WCETs (worst case execution time) is unacceptable. Making the GC work totally concurrent is one way to address the above problems. In the concurrent GC algorithms, GC work is performed by a segregated thread (with normally the lowest priority) called the GC thread, which is also managed by the scheduler. Unfortunately, although applying this mechanism to real-time systems doesn't influence the execution of the real-time threads at all, the system cannot guarantee enough progress for the collector so a high priority real-time thread may be blocked by the low priority collector because of the lack of free memory. Since such unpredictable blocking is prohibited in real-time systems, the primitive concurrent GC model cannot be directly introduced to real-time systems.

In order to resolve this problem, we propose to modify the GC thread to a periodic real-time thread by giving it parameters such as period, deadline, priority, WCET and so on. As a periodic real-time thread, the GC thread can now be scheduled in the similar way as other real-time threads and more importantly, 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. If, according to the analysis, it is schedulable and the parameters we give the GC thread are correct, we'll be able to guarantee not only that the mutator is schedulable but also that the GC thread can get enough system resources to execute. 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 [8] to schedule the GC thread.

This paper is organized as follows: Section 2 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 and worst case response time. Section 6 proves that our algorithm doesn't compromise the schedulability of the threads with lower priorities than the GC thread in spite of the period variation of the GC thread. Finally, we present our conclusion and future work.

2 Related Work

Traditionally, GC algorithms and scheduling methods are discussed separately and most work focuses on the GC algorithms rather than the scheduling properties. This situation has changed in recent years. Researchers [16, 10, 13] have tried to work out GC scheduling methods that are suitable for real-time systems.

[1]As defined by Bacon [1], it is the fraction of the CPU time devoted to the mutator.

Henriksson [10] proposed a copying GC algorithm which is based on Brook's previous work [4]. In his approach, the GC work introduced by hard real-time processes is performed by a so called *High-Priority Garbage Collection Process* which has a priority lower than all the hard real-time processes but higher than all the soft real-time and non-real-time processes. On the other hand, the GC work introduced by soft real-time and non-real-time processes is scheduled in a work-based incremental manner. Since the *High-Priority Garbage Collection Process* is tightly coupled to hard real-time processes and doesn't have an explicit period and deadline, it is not an ordinary concurrent real-time process so it cannot be integrated directly into the current schedulability analysis framework. For those processes with lower priorities than the GC process, their response time must be calculated in a specific way rather than the standard one. By contrast with Henriksson's work, nearly all the GC work in our system is performed by a totally concurrent real-time thread with an explicit period and deadline. Therefore, our approach is more flexible and easier to be integrated into the current schedulability analysis framework.

Instead of using a segregated thread to collect garbage, Kim [13, 14] introduced aperiodic server techniques to garbage collection. In his earlier paper, a sporadic server with the highest priority is introduced to execute a copying garbage collector. On the other hand, in the latest version, another sporadic server with lower priority is used as well in order to make it easier to perform copying operations of large memory blocks. To provide temporal and spacial guarantees offline, the sporadic server's safe capacity must be calculated and used together with garbage collector's worst case execution time to calculate the garbage collector's worst case response time based on which worst case length of a GC cycle can finally be obtained. According to the estimations of mutator threads' allocation rate and the maximum amount of live memory, how much memory, in the worst case, we need for any GC cycle can also be decided.

The aforementioned two approaches are all 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, none of them can proceed in that scenario 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.

Robertz [16] introduced a way to modify a primitive concurrent GC thread to a real-time thread by giving it an explicit deadline and period. We follow a similar way to make our GC thread real-time. However, our approach is based on a hybrid garbage collection algorithm in which it is the more efficient reference counting algorithm that tends to be more likely to be executed rather than the tracing algorithm.

Although others have explored such a combination [9] and even implemented it in some systems (e.g. Inferno Operating System), so far as we know, none of them tried to introduce it into the real-time domain.

3 Basic Algorithm

The reason why we choose a combination of reference counting and mark-and-sweep rather than a pure tracing algorithm is given below:

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 a 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.

As discussed previously, reference counting algorithms cannot reclaim cyclic garbage per se. The basic principle of our method is to combine reference counting and tracing algorithms together so that all the acyclic garbage objects can be reclaimed in the GC cycles where they are generated and the cyclic garbage can be found by a single tracing collection in each GC cycle. More specifically, each GC cycle corresponds to a release of the GC thread; Such a release both reclaims acyclic garbage generated during its response time and performs a single tracing collection. 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 (i.e. the GC thread in our system have a longer period). 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 further discussed in section 5.

By combining the two approaches, we can also eliminate the root scanning phase and therefore the synchronization points [18] 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.

Next, we introduce our algorithm in more details.

First, we assume that heap is divided into fixed size blocks as introduced by Siebert and Ritzau [17, 15] so that fragmentation problems can be eliminated.

In our approach, the reference count for each object is conceptually divided into two parts: one is for recording the number of roots that reference the object directly (“root count” for short); the other one is for recording the number of all the other direct references to that object (“reference count” for short). Also, we maintain 3 doubly linked lists: *root-list*, *tracing-list* and *white-list*. Any live object in the system must be in and simultaneously only in one of the aforementioned lists. The objects in root-list or tracing-list have the colour of black or grey. In order to determine the colour of objects, two additional pointers are introduced for root-list and tracing-list. As the name indicates, white-list contains only white objects. On the other hand, dead objects must be put into a linked list called *to-be-free-list* which is processed by the GC thread. Finally, the allocator searches free memory to allocate from the beginning of another linked list called *free-list* which is composed of fixed size blocks. (see figure1)

Next, we use pseudo code in figure2 as an example to illustrate object memory layout. Here, An object is composed of one or more 32bytes blocks.

Currently, we take advantage of two kinds of write barriers: *write-barrier-for-roots* and *write-barrier-for-objects*. They are invoked respectively when mutator threads assign a value to a root or to a reference field in an object (or array). Which one to invoke is determined off-line by the compiler so there’s no need to make such choices in run-time. The pseudo code of the write barriers is given in

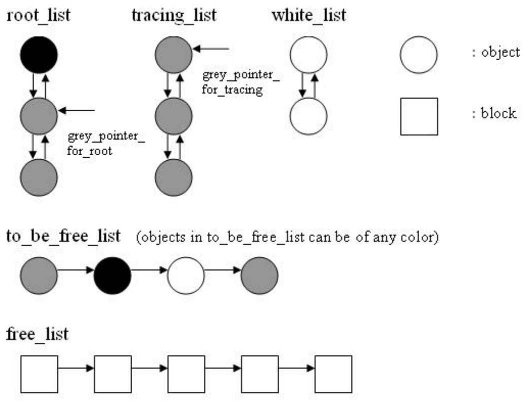

Figure 1: data structures

```
/*   a Java class is used as an example   */

public class TestObject{
   int a1, a2,......, a11;
}

/* In our system, an object of this class  *
 * contains two blocks. The memory layout  *
 * of the first block can be represented   *
 * as:                                     */

   struct first_block_of_TestObject{
      int      reference_count;
      void *   ref_prev;
      void *   ref_next;
      //for the doubly-linked lists
      int      a1, a2, a3, a4;
      second_block_of_TestObject * next_block;
   };

      /*   and the second block   */

   struct second_block_of_TestObject{
      int      a5, a6,.....,a11;
      third_block_of_TestObject * next_block;
      // this pointer should be NULL
   };
```

Figure 2: an object's memory layout

```
void write_barrier_for_objects(void *from, void *to)
{
   if( to !=NULL && from ==NULL )
   //when we assign NULL to a reference field which
   //still references something
   {   update(minus one) the object count of the
       object referenced by "to"
       if( both object and root counts of the object
           referenced by "to" are zero)
       {
          if(the object referenced by "to" is in
             "white_list")
             unlink it from the white list
          else
          {  unlink it from its current list
             update the corresponding grey_pointer
          }
          add it to the "to_be_free_list"
       }
   }
   else if( to ==NULL && from !=NULL)
   //when we assign a valid reference to a reference
   //field whose value is NULL
   {   update(add one) the object count of the object
       referenced by "from"
       if("to" corresponds to a field in a black
          object and "from" references a white object)
       {
          if(the object referenced by "from" is in
             "white_list")
             unlink it from the "white_list"
          add the white object to the end of
          "tracing_list"
          update the "grey_pointer_for_tracing"
       }
       else if(which list the object referenced by
               "from" should belong hasn't been
               decided)
          add it to the end of the "white_list"
   }
   else if(to !=NULL && from !=NULL)
   //when we assign a valid reference to a reference
   //field which still references something
   {
       perform what the function does when to ==NULL
       && from !=NULL
       perform what the function does when to !=NULL
       && from ==NULL
   }
}
```

Figure 3: pseudocode for write_barrier_for_objects

figure3 and figure4.

In our approach, the GC thread does mainly two things: processing the to-be-free-list and perform-

```
void write_barrier_for_roots(void *from, void *to)
{
   if( to !=NULL && from ==NULL )
   //when we assign NULL to a root which still
   //references something
   {  update(minus one) the root count of the object
      referenced by "to"
      if(both root and object counts of the object
         referenced by "to" are zero)
      {
         unlink it from "root_list"
         update the "grey_pointer_for_root"
         add it to the end of "to_be_free_list"
      }
      else if(root count of the object referenced by
              "to" is zero)
      {
         unlink it from "root_list"
         update the "grey_pointer_for_root"
         if(the object referenced by "to" is not black)
         {
            add it to the end of the "tracing_list"
            update "grey_pointer_for_tracing"
         }
         else
            add it to the beginning of "tracing_list"
      }
   }
   else if(to ==NULL && from !=NULL)
   //when we assign a valid reference to a root
   //whose value is NULL
   {  update(add one) the root count of the object
      referenced by "from"
      if(the root count of the object referenced by
         "from" is one)
      {
         unlink it from its current list
         update the corresponding grey_pointer if
         necessary
         if(it's black)
            add it to the beginning of "root_list"
         else
         {
            add it to the end of "root_list"
            update the "grey_pointer_for_root"
         }
      }
   }
   else if(to !=NULL && from !=NULL)
   //when we assign a valid reference to a root
   //which still references something
   {
       perform what the function does when to ==NULL
       && from !=NULL
       perform what the function does when to !=NULL
       && from ==NULL
   }
}
```

Figure 4: pseudocode for write_barrier_for_roots

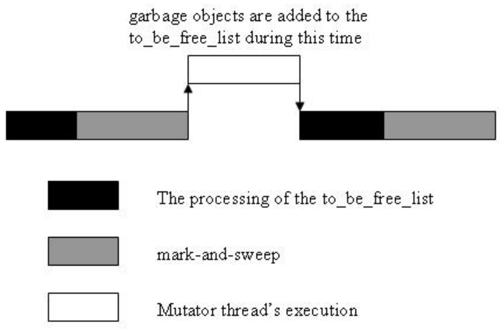
garbage objects are added to the to_be_free_list during this time
■ The processing of the to_be_free_list
▨ mark-and-sweep
□ Mutator thread's execution

Figure 5: how mark-and-sweep is preempted

ing mark-and-sweep. 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 otherwise. 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 figure5.

When the GC thread is processing the to-be-free-list, it always examines the first object. If that object has any direct child, the object count of its direct child must be decreased by 1. If both object count and root count is 0, that direct child object will be linked to the rear of the to-be-free-list. Otherwise, nothing is changed except the object count. Having processed all the direct children, the GC thread finally moves the first object in the to-be-free-list to the free-list.

When it comes to mark-and-sweep, the situation becomes a little bit more complex. The pseudo code is listed in figure6.

When the mark-and-sweep work is completed, the current release of the GC thread ends.

4 Scheduling

Assuming that the correct priority, period, deadline, WCET of the GC thread have been given, we can calculate the worst case response time of the GC thread using standard analysis techniques [5] and compare the worst case response time with the deadline to see whether the GC thread is schedulable or not. Since garbage collection is not a user

```
while( grey_pointer_for_root !=NULL
       || grey_pointer_for_tracing !=NULL)
//introduce this while-loop in case mutator threads
//add any grey object to root-list when the second
//inner while-loop is being executed
{
    while(grey_pointer_for_root !=NULL)
    //traverse root-list until all objects in that
    //list are black
    {
       for(all the direct children of the current
           object)
          if(the current child is in "white_list")
          {
             move it to the end of tracing list
             update "grey_pointer_for_tracing" if
             necessary
          }
       update "grey_pointer_for_root" to point to the
       object after the current one
    }
    while(grey_pointer_for_tracing !=NULL)
    //traverse tracing-list until all objects in that
    //list are black
    {
       for(all the direct children of the current
           object)
          if(the current child is in "white_list")
          {
             move it to the end of tracing list
             update "grey_pointer_for_tracing" if
             necessary
          }
       update "grey_pointer_for_tracing" to point to
       the object after the current one
    }
}
```

Figure 6: pseudocode for mark-and-sweep

function of the system, we hope to schedule it as infrequently as possible and also as late as possible so we require that the GC thread should always give 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).

In order to achieve this, we propose a mechanism which takes advantage of the dual priority technique [8]. In the dual priority algorithm, a real-time thread can have two priorities and a promotion time before which the thread is executed at its lower priority. When the given promotion time has elapsed, the thread is promoted to its higher priority. Applying this mechanism to the GC thread has the consequence that the GC thread is potentially delayed for a certain time after its release. 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 7). At first glance, this may ruin the schedulability of the real-time threads with lower priorities than the GC thread since the number of the releases of the GC thread is increased. However, we will prove that this is not the case in section 6.

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. Compared to the primitive concurrent GC mechanism, our approach can now ensure that the GC thread can get enough system resources. However, this doesn't necessarily mean

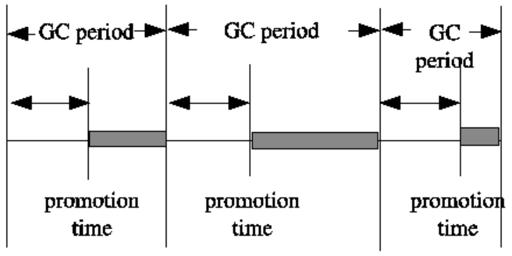

Figure 7: The period of the GC thread is changeable

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 the GC thread should always try to preserve enough free memory for the real-time threads with higher priorities than the GC thread (promoted) so that those higher priority threads can execute without any blocking from the GC thread [10]. To preserve memory for those threads, we need to change our algorithm a little bit: 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.

5 Static Analysis

In this section, we calculate the parameters for the GC thread. First, some symbols must be defined:

Assume that a real-time system is composed of a set of threads, P, whose live memory consumption has an upper bound, L_{max}. In addition, we provide this real-time system with a heap whose size is H. We also defined $hp(GC)$ which represents the set of all the threads with higher priorities than the GC thread (promoted). As previously discussed, the GC thread is a periodic real-time thread (albeit with varying period) so 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 other symbols can then be defined in table 1.

Now, we give some simple formulas without proof since they are self-explanatory.

$$H = L_{i+1} + F_{i+1} + CG_i + FCG_i \qquad (1)$$

which means the dead but not freed memory at the beginning of cycle $i + 1$ includes only all cyclic garbage found and all floating cyclic garbage generated in cycle i.

$$L_{i+1} = L_i + a_i - RG_i - CGG_i \qquad (2)$$

which means the cumulation of live memory in cycle i is all allocations happened in cycle i minus the amount of garbage generated in cycle i.

$$A_{i+1} = A_i + a_i - RG_i - CG_{i-1} \qquad (3)$$

which means the cumulation of allocated memory in cycle i is all allocations happened in cycle i minus the amount of garbage we can reclaim in cycle i. Furthermore, none of the cyclic garbage found in cycle i can be reclaimed.

$$CG_i \geq FCG_{i-1} \qquad (4)$$

which means all the floating cyclic garbage generated in cycle $i-1$ must be recognized as garbage by the end of cycle i.

Having got the above formulas, we can now try to calculate the deadline of the GC thread, D. Assuming, in the worst case, that L_i equals L_{max}, since L_{i+1} must be smaller than or equal to L_{max}, we can get $L_{i+1} - L_i \leq 0$. Applying equation 2 to here gives:

$$a_i - RG_i - CGG_i \leq 0 \qquad (5)$$

and therefore,

$$a_i - RG_i \leq CGG_i \qquad (6)$$

From equation 3 and 6, we can get:

$$A_{i+1} - A_i \leq CGG_i - CG_{i-1} \leq CUM_{max} \qquad (7)$$

Symbols	Definitions
L_i	the amount of live memory just before the ith release of the GC thread
F_i	the amount of free memory just before the ith release of the GC thread
A_i	the amount of allocated (not freed) memory just before the ith release of the GC thread
CGG_i/CGG_{max}	the amount of cyclic garbage generated in cycle i and its maximum value
$CG_i/CG_{min}/CG_{max}$	the amount of cyclic garbage found by the ith release of the GC thread and its minimum and maximum value
FCG_i	the amount of floating cyclic garbage generated in cycle i (it must be found by the $(i+1)$th release of the GC thread)
RG_i	the amount of acyclic garbage generated in cycle i(it must be reclaimed within cycle i)
a_i	new memory allocated in cycle i
CUM_{max}	the maximum value of $CGG_i - CG_{i-1}$
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
C_{GC}	the worst case execution time of the GC action during R_{pre}
D	the deadline of the GC thread
$WCET_{GC}$	the worst case execution time of the GC thread
R_{total}	the worst case response time of the GC thread
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

which means that if only the GC thread can provide as much free memory as CUM_{max} at the beginning of any GC cycle, the cumulation of the allocated memory can always be satisfied. However, in order to synchronize reclamation and allocation, we need to preserve F_{pre} free memory as well. Therefore, in order to guarantee that the application never runs out of memory, we should be able to provide at least $F_{pre} + CUM_{max}$ free memory at the beginning of any GC cycle. As defined previously, $F_{pre} + CUM_{max}$ can be represented as:

$$F_{min} = F_{pre} + CUM_{max} \tag{8}$$

By changing the form of equation 1, we can get:

$$F_{i+1} = H - L_{i+1} - CG_i - FCG_i$$

In the worst case, this can be modified as:

$$F_{min} = H - L_{max} - (CG_i + FCG_i)_{max} \tag{9}$$

Applying equation 8 to this equation:

$$F_{pre} + CUM_{max} = H - L_{max} - (CG_i + FCG_i)_{max} \tag{10}$$

In the worst case, FCG_i equals CGG_{max} and CG_i equals FCG_{i-1}. That is, none of the cyclic garbage generated in cycle i can be found within the same cycle and all the garbage found in cycle i is floating garbage from cycle $(i-1)$. Assuming that GC cycle $i-1$ is also in the worst case, we can get $CG_i = FCG_{i-1} = CGG_{max}$, so

$$(CG_i + FCG_i)_{max} = 2 \cdot CGG_{max} \tag{11}$$

Applying this to equation 10 gives:

$$F_{pre} + CUM_{max} = H - L_{max} - 2 \cdot CGG_{max} \tag{12}$$

Now, let's go back to equation 7. In the worst case, $CG_{i-1} = 0$ so $(CGG_i - CG_{i-1})_{max} = CUM_{max} = (CGG_i)_{max}$. Therefore,

$$CUM_{max} = CGG_{max} \tag{13}$$

Applying this to equation 12, we can get:

$$3 \cdot CUM_{max} = H - L_{max} - F_{pre}$$

and therefore,

$$CUM_{max} = \frac{H - L_{max} - F_{pre}}{3} \tag{14}$$

As defined previously, $CGG_i - CG_{i-1} \leq CUM_{max}$ so we can get:

$$CGG_i - CG_{i-1} \leq \frac{H - L_{max} - F_{pre}}{3} \tag{15}$$

and therefore,

$$CGG_i \leq \frac{H - L_{max} - F_{pre}}{3} + CG_{i-1} \tag{16}$$

Assuming the worst case scenario that all the mutator threads arrive at the same time, CGG_i can be represented as:

$$CGG_i = \sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \tag{17}$$

Applying this to inequality 16, we can get:

$$\sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \leq \frac{H - L_{max} - F_{pre}}{3} + CG_{i-1} \tag{18}$$

For simplicity and without loss of safety, we can modify this inequality into:

$$\sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \leq \frac{H - L_{max} - F_{pre}}{3} + CG_{min} \tag{19}$$

Unfortunately, we cannot resolve this inequality since we haven't got the F_{pre}. In order to obtain this value, we need to first calculate R_{pre}:

$$R_{pre} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{pre}}{T_j} \right\rceil \cdot C_j \right) + C_{GC} \tag{20}$$

$$C_{GC} = RR \cdot \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{pre}}{T_j} \right\rceil \cdot a_j \right) \tag{21}$$

therefore,

$$R_{pre} = \sum_{j \in hp(GC)} \left[\left\lceil \frac{R_{pre}}{T_j} \right\rceil (C_j + RR \cdot a_j) \right] \tag{22}$$

Based on the value of R_{pre}, we can finally decide F_{pre}:

$$F_{pre} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{pre}}{T_j} \right\rceil \cdot a_j \right) \tag{23}$$

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 22, 23 and 19 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. A simple example is illustrated in figure8: the GC thread first has the lowest priority in the system. Then we calculate its deadline $deadline1$ which corresponds to the second highest priority according to DMPO. Again, we calculate $deadline2$ (according to the new priority) which corresponds to the second lowest priority. Finally, the $deadline3$ is calculated and the whole procedure ends since $deadline3$ corresponds to the second lowest priority as well.

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.

$$WCET_{GC} = RR \cdot CG_{max} + RR \cdot \sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot rg_j \right) + TR \cdot L_{max} \tag{24}$$

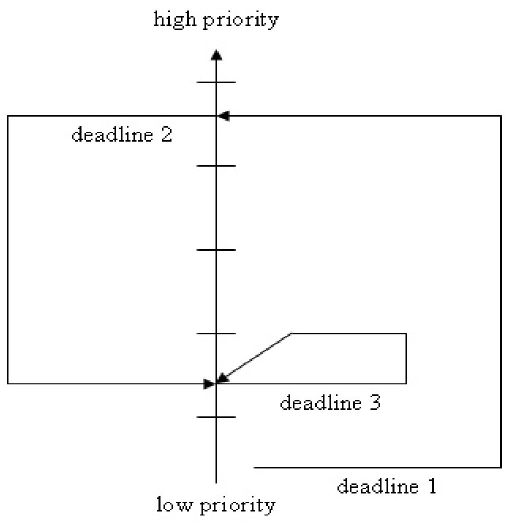

Figure 8: The procedure of determining GC thread's deadline and priority

$$R_{total} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{total}}{T_j} \right\rceil \cdot C_j \right) + WCET_{GC} \quad (25)$$

As what happens in the standard 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 assuming GC thread is another periodic real-time user 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.

6 Dealing with Period Variation

As we discussed before, the period of each release of the GC thread could be different. Which GC period should we use to calculate the response time of the lower priority threads? Here, we propose to use the worst case period, T_{GC}, which equals D and therefore is the same for all the releases. Next, we will prove the correctness of this analysis approach.

For convenience, we divide a GC cycle into two periods: one is called GC busy period during which only the GC thread and higher priority threads can execute; the other one is called GC idle period during which the GC thread cannot execute. In the worst case, the GC busy period of a GC cycle has the length of R_{total} and the GC ideal period has the length of $T_{GC} - R_{total}$. In addition, during the GC idle period, the lower priority threads will suffer from interference from those threads with higher priorities than the GC thread so the actual time can be used by the lower priority threads in each release of the GC thread is (we assume that all the real-time threads with higher priorities than the GC thread are released simultaneously at the beginning of each GC idle period):

$$T_{GC} - R_{total} - \sum_{j \in hp(GC)} \left\lceil \frac{T_{GC} - R_{total}}{T_j} \right\rceil \cdot C_j \quad (26)$$

In the normal cases, the real period $T'_{GC} \leq T_{GC}$ and the real response time $R'_{total} \leq R_{total}$. However, as we defined before,

$$T_{GC} - R_{total} = PromotionTime$$

and since we argue that the GC thread cannot be completed before its promotion time, we can also get

$$T'_{GC} - R'_{total} = PromotionTime = T_{GC} - R_{total}$$

Notice that the symbol $PromotionTime$ represents the length of a time period rather than a time point.

By using the same approach, the time can be used by the lower priority threads in one release of the GC thread in the normal cases can be described as:

$$T'_{GC} - R'_{total} - \sum_{j \in hp(GC)} \left\lceil \frac{T'_{GC} - R'_{total}}{T_j} \right\rceil \cdot C_j \quad (27)$$

Therefore, irrespective of the value of T'_{GC}, formulas 26 and 27 have the same value.

Assume that there is a $k \in \{1, 2, 3 \cdots\}$ and $k \cdot \left(T_{GC} - R_{total} - \sum_{j \in hp(GC)} \left\lceil \frac{T_{GC} - R_{total}}{T_j} \right\rceil \cdot C_j\right)$ has the smallest value which is greater than the sum of the WCET of a specific thread $\mathcal{A}$ with lower priority than the GC thread and the interferences from other threads which have priorities lower than the GC thread but higher than the thread $\mathcal{A}$. Therefore, one release of the thread $\mathcal{A}$ must be able to be completed within k releases of the GC thread. Since formulas 26 and 27 always have the same value and the worst case interference to the thread $\mathcal{A}$ is the same, one release of the thread $\mathcal{A}$ in any normal case can also be finished within k releases of the GC thread. Since $T'_{GC} \leq T_{GC}$, k releases of the GC thread in the normal cases must be completed before k releases in the worst case. Therefore, if thread $\mathcal{A}$ is schedulable in the worst case, it must also be schedulable in any normal case. In addition, this also proves that although the dual priority scheduling scheme doesn't improve the worst case response time of the lower priority threads, it can help us achieve shorter average response time.

7 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 [19, 15]. 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 [7], which is a RTSJ-compliant real-time extension to the GCJ compiler. We choose this platform because it is relatively easy to obtain and modify its source code and it is much more convenient to build up our implementation on a platform already with comprehensive real-time features. The new compiler and library collectively generate binary executables for programs written in Java and compliant with RTSJ (real-time specification for Java) except that the RTSJ memory model is substituted by our garbage collection approach.

References

[1] David F. Bacon, Perry Cheng, and V.T. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of POPL 2003*, pages 285–298. IBM T.J. Watson Research Center, January 2003.

[2] David F. Bacon and V.T. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the 2001 European Conf. on Object-Oriented Programming*. IBM T.J. Watson Research Center, June 2001.

[3] G. Bollella, B. Brosgol, P. Dibble, S. Furr, J. Gosling, D. Hardin, M. Turnbull, and R. Belliardi. *The Real-Time Specification for Java (1.0)*. Addison Wesley, 2001.

[4] Rodney A. Brooks. Trading data space for reduced time and code space in real-time garbage collection on stock hardware. In *Proceedings of the 1984 ACM Symposium on LISP and functional programming*, pages 256 – 262. Computer Science Department, Stanford University, 1984.

[5] Alan Burns and Andy Wellings. *Real-Time Systems and Programming Languages*. Addison Wesley, third edition, 2001.

[6] Dante J. Cannarozzi, Michael P.Plezbert, and Ron K. Cytron. Contaminated garbage collection. In *Proceedings of PLDI 2000*, pages 264 – 273. Computer Science Department, Washington University, May 2000.

[7] Angelo Corsaro and Douglas C. Schmidt. The design and performance of the jrate real-time java implementation. In *Proceedings of the 4th International Symposium on Distributed Objects and Applications*. University of California, 2002.

[8] Robert Davis and Andy 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.

[9] L. P. Deutch and D. Bobrow. An efficient incremental automatic garbage collector. *Communications of the ACM. Association for Computing Machinery*, 19(9):522–526, September 1976.

[10] Roger Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, July 1998.

[11] Mark Stuart Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, The University of Texas at Austin, December 1997.

[12] Richard Jones and Rafael Lins. *Garbage Collection*. John Wiley & Sons, 1996.

[13] Taehyoun Kim, Naehyuck Chang, and Heonshik 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.

[14] Taehyoun Kim and Heonshik Shin. Scheduling-aware real-time garbage collection using dual aperiodic servers. In *Proceedings of the IEEE 9th RTCSA*, pages 3–20. Seoul National University, February 2003.

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

[16] Sven Gestegård Robertz and Roger Henriksson. Time-triggered garbage collection—robust and adaptive real-time gc scheduling for embedded systems. In *Proceedings of LCTES 2003*, pages 93–102. Lund University, 2003.

[17] Fridtjof Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *Compilers, Architectures and Synthesis for Embedded systems(CASES2000)*, November 2000.

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

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