Hard Real-time Hybrid Garbage Collection with Low Memory Requirement

Yang Chang
University of York, UK, yang@cs.york.ac.uk

Abstract

Current real-time garbage collection algorithms are usually criticised for their high memory requirements. Even when consuming nearly 50% of cpu time, some garbage collectors ask for at least twice the memory as really needed. This report explores the fundamental reason of this problem and proposes new metrics for real-time garbage collection algorithm designs. Use of these metrics motivate an algorithm that combines both reference counting and mark-and-sweep techniques. The use of dual priority scheduling of the garbage collection task allows spare capacity in the system to be reclaimed whilst guaranteeing deadlines.

1 Introduction

The management of resources is a key component of all real-time systems. One of the main advantages of using a high-level language is that it relieves the programmer of the burden of dealing with many low-level resource allocation issues. Activities such as assigning variables to registers or memory locations, allocating and freeing memory for dynamic data structures, etc., all distract the programmer from the task at hand, which is to produce application code to perform some functionality.

Languages like Java remove many of these distractions and provide high-level abstract models that the programmer can use. Unfortunately, for real-time and embedded systems programming there is a conflict. On the one hand, the use of high-level abstractions aid in the software engineering of the application. On the other hand, embedded and real-time systems often have only limited resources (time and space) and these must be carefully managed. Nowhere is this conflict more apparent than in the area of memory management. Embedded systems usually have a limited amount of memory available; this is because of cost, size, power, weight or other constraints imposed by the overall system requirements.

The run-time implementations of most programming languages provide two essential data structures to help manage the dynamic memory of the program: the stack and the heap. Whilst data on the stack is automatically reclaimed, data placed on the heap can be reclaimed in several ways:

- Require the programmer to return the memory explicitly - this increases the burden on the programmer and is error prone, but it is easy to implement (e.g. the use of malloc and free in Posix) and operates at a fine granularity.
- Use a language with explicit scope rules – the run-time support system can then monitor the memory and determine when it can **logically** no longer be accessed. Ada adopts this approach; when a reference type goes out of scope, all the memory associated with that reference type can be freed; This places a burden on the programmer, operates at a coarse granularity but is relatively simple to implement.
- Require the run-time support system to monitor the memory and release chunks that are no longer being used (garbage collection). This frees the programmer from the burden of memory management, operates at a fine granularity but is complex to implement. Java adopts this approach.

From a real-time perspective, the above approaches have an increasing impact on the ability to analyze the timing and memory properties of the program. In particular, garbage collection may be performed either when the heap is full (there is no free space left) or incrementally (either by an asynchronous activity or on each allocation request). In either case, running the garbage collector may have a significant impact on the response time of a time-critical thread. Furthermore, different collectors have different overheads on the extra memory they need (above the application's requirements).

Most garbage collectors now in use are tracing collectors. Since application can run indefinitely, garbage collectors need to run repeatedly. Each execution usually includes three phases:

Root Scanning – looks for all the references from outside the heap to somewhere in the heap.

Marking – marks (or moves) all the reachable objects starting from the references found in the first phase.

Reclaiming – reclaims all other objects and updates housekeeping information for survived objects where necessary (compaction would also be performed in this phase if fragmentation is to be resolved).

A complete execution of the above phases is named a GC cycle which can consume extensive computation resources. Therefore, in a non-real-time system, the user tasks could be blocked for a long time waiting for the collector to finish a GC cycle. The current efforts to make garbage collection real-time is to reduce the huge GC blocking time by performing garbage collection work incrementally or concurrently so that the perceived worst-case latency of user tasks is bounded and relatively low [1, 10, 14, 8]. However, the low perceived worst-case latency is obtained at the cost of high space overhead, which is also unfavourable in real-time embedded systems. The less computation resource is given to the garbage collector, the more space is needed and vice versa. Achieving the correct balance of this tradeoff is difficult and is one of the main criticisms of the garbage collection approach to real-time memory management[3].

In this report, we discuss the reasons why tracing real-time garbage collectors cannot perform sufficiently well in both temporal and spatial aspects at the same time. We present new performance metrics for describing the overall real-time capability of a garbage collector and use them to compare some classical garbage collection algorithms. The use of these metric motivates the design of a new real-time garbage collector. Instead of performing pure tracing or reference counting, we propose a hybrid approach which combines both the advantages of reference counting and mark-and-sweep. The garbage collection task is executed concurrently with user tasks and scheduled using dual priorities way, which can allow the reclamation of spare capacity in the whole system [5].

The remainder of this report is organized as follows. Section 2 explores the reason why current tracing real-time garbage collection techniques have the aforementioned drawbacks and proposes our new metrics. Section 3 briefly introduces the basic idea of our approach. Section 4 reveals more details of our algorithm and scheduling method. Section 5 presents a static analysis of our garbage collected hard real-time system. Section 6 shows some empirical results. Finally, we present our conclusion and future work.

2 The Need for New Performance Metrics

Reclaiming unused objects (garbage) with any garbage collection algorithm takes at least two steps: *identifying* and *reclaiming*. Using tracing collectors as an example, they need to go through both root scanning and marking phases to identify garbage and then recover it in a reclaiming phase. As we will discuss later, the granularity of such *identifying* and *reclaiming* cycles is of great importance in a real-time environment. If this granularity is too big, the algorithm will be unsatisfactory in either, or both, of the following ways:

1. User tasks will suffer from significant latencies due to garbage collection activities. These latencies may cause tasks to miss their deadlines or require the system to have a more powerful processor than needed.

2. Significant redundant space will be needed to reduce latencies. Unfortunately, this may result in an embedded systems running out of memory or require more memory than needed.

The end result will be a system which needs more resources than one which adopts a more application memory-managed approach.

Recent research on real-time garbage collection algorithms such as [1, 10, 14, 8] is mainly focused on achieving predictability and low worst-case latency. However, we argue that focusing simply on latency hides the real problem. To make this clear, a simple example is given as follow: two garbage collectors have the same throughput but one can produce 32 bytes of free memory as a whole in every 10 microseconds (smaller granularity) whilst the other one can only produce 3200 bytes as a whole in the last 10 microseconds of every millisecond (larger granularity). They perform as bad, or as well, as each other in a non real-time environment since the only concern there is throughput. However, the first one outperforms the latter one in a real-time environment due to its much lower latency. Irrespective of how small a portion of memory a user task requests, the latter one needs 1 millisecond to perform its work before the user task can proceed. However, using incremental techniques, the second collector's work can be divided into small pieces, say 10 microseconds, which are then interleaved with user tasks. This reduces the perceived worst-case latency of user tasks to the same as that of the first collector. Unfortunately, nothing comes for free. Since 99 out of 100 increments cannot produce any free memory, allocation requests before the last increment must be satisfied by an extra memory buffer. This simple example explains the importance of garbage collection granularity in a real-time environment.

As previously discussed, all the tracing garbage collectors need to identify live objects before reclaiming garbage. In the worst case situation where live memory reaches its upper bound, the collector has to scan at least all the live memory. Since the reclaiming phase of a GC cycle is usually short and bounded, the example shown above can simulate the behaviour of tracing collectors very well.

This suggests that tracing techniques are inherently unsuitable for real-time systems due to their high granularity.

Irrespective of the scheduling algorithm, any attempt to give significantly less computation resource to a tracing collector will raise the memory requirement dramatically [14].

To allow for better performance evaluation of garbage collection algorithms, we define two new metrics which characterise the granularity of a garbage collector. First, the definition of a *Free Memory Producer* is given below:

Definition 1 The *Free Memory Producer* of a garbage collector is a logical task which works at the highest priority and executes the algorithm of that garbage collector without trying to divide its work but stops whenever any chunk of new free memory, irrespective of its size, is made available to the allocator.

Moreover, the free memory producer can only be released at the point where

1. the amount of live memory reaches its upper bound and there exists garbage with arbitrary size, or
2. the first time an object(s) becomes garbage after live memory reached its upper bound when there was no garbage.

We call such a point the *Free Memory Producer Release Point* [1]. Notice that a free memory producer doesn't necessarily need to be a fully functional garbage collector since it stops whenever any free memory is produced. Thus, it is required that the whole system should stop when the free memory producer stops. Because no garbage can be reclaimed before the free memory producer release point, the heap must be big enough to hold all the garbage objects and the live ones.

Now, we can define two metrics:

Definition 2 The *Free Memory Producer Complexity* of a garbage collector is the algorithmic complexity of its free memory producer.

Definition 3 The *Free Memory Producer Latency* of a garbage collector is the worst case execution time of its free memory producer.

Lower free memory producer complexity and latency always mean that the corresponding collector has lower granularity and is more responsive in producing free memory. Therefore, the user tasks suffer from shorter worst-case latency introduced by garbage collection or the overall memory needed is smaller. Assuming throughput is the same, the garbage collection algorithm which has lower free memory producer complexity and latency is more likely to be appropriate for real-time systems. Care must be taken when evaluating the real-time capabilities of those garbage collectors which also consume free memory. The reason is that their free memory producer complexities and latencies are obtained when there is already a memory buffer inherently preserved for the algorithms to proceed. Copying garbage collection [2] and train algorithms [9] are such examples.

[1]The Free Memory Producer Release Point doesn't necessarily exist in a real application but can be created intentionally in testing programs.

Garbage Collection Algorithm	FMP Complexity
Conventional Reference Counting	$O(garbage\_set\_size)$
Deferred Reference Counting	$O(object\_size)$
Non-copying Tracing	$O(L_{max})$
Copying Tracing	$O(L_{max})$

Table 1. Free Memory Producer Complexity

Consider now the free memory producer complexities of some popular garbage collection algorithms in table 1.

As you can see, all of them are of $O(n)$ complexity, which means they are all linear algorithms. However, the values of n are different.

$garbage\_set\_size$ is the total size of the garbage set (e.g. a long linked list) being processed

$object\_size$ is the size of the garbage object (e.g. object or array in an OO language) being processed

L_{max} is the maximum amount of live memory in the whole system

At first glance, the n value of reference counting, particularly deferred reference counting, is very promising. However, "$garbage\_set\_size$" and "$object\_size$" can be also very big, even comparable with "L_{max}" in extreme situations. Therefore, the free memory producer latencies of reference counting algorithms could be very long in some systems as well.

In order to keep memory requirement low, we suggest that a real-time garbage collector should try to minimise the free memory producer complexity and latency but not at a cost of very high inherent space overheads or very poor throughput. In the following section, we present our approach to achieve this goal.

3 The Hybrid Approach Overview

One way to improve the reference counting algorithms is to change their garbage collection granularity. Siebert and Ritzau [14, 11] both noticed that external fragmentation can be eliminated by dividing objects and arrays into fixed size blocks. This not only resolves the external fragmentation problem but also changes the granularity of garbage collection so that it can reclaim such blocks individually. For a deferred reference counting algorithm that maintains objects

and arrays as fixed size blocks (hereafter, we call such algorithm "fine grained reference counting"), its free memory producer has a constant complexity (O(1)) since the block size in a given system is fixed. On the other hand, because the block size is always small, a very low free memory producer latency can be achieved as well. Consequently, such a fine grained reference counting algorithm is more likely to be suitable for real-time compared to other reference counting algorithms. Ritzau's work [11] is an example of such algorithm. By reclaiming the same amount of memory immediately before each allocation, there's no need to preserve any free memory buffer. However, reference counting algorithms cannot reclaim cyclic garbage per se so Ritzau's pure reference counting algorithm is unable to reclaim all the garbage without the help from programmers. Furthermore, his algorithm is a work-based algorithm which is not well integrated with real-time systems.

By combining a fine grained reference counting collector with a mark-and-sweep collector, the problem of cyclic garbage can be resolved. However, such a hybrid algorithm has a very special free memory producer. When there exists reference-counting-recognizable garbage at the free memory producer's release point, the free memory producer will have the same complexity and latency as that of the fine grained reference counting algorithm, i.e. O(1) complexity and very low latency. On the other hand, when there's no such garbage, the free memory producer will have a similar behaviour as that of a pure tracing algorithm.

The most straightforward way to tackle the second situation's long latency is to introduce extra memory. But, this time, the memory buffer only holds garbage objects that are in cyclic data structures rather than all of the allocations due to the contribution of the reference counting collector. In many applications, acyclic data accounts for a considerable portion or even majority of total memory usage.

By combining the two approaches, we can also eliminate the root scanning phase since our reference counting algorithm can provide enough information to the mark-and-sweep collector. Incremental root scanning techniques can be found in [2, 7, 14].

4 The Hybrid Garbage Collection Algorithm

This section presents details of our hybrid algorithm and its scheduling properties.

4.1 Data Structures

As discussed in the previous section, every object in our system is maintained as a linked list of fixed size blocks which are set to 32 bytes in the current implementation. Which size to choose and how bad the memory access penalty is have been studied by Siebert [13]. The pseudo code in figure1 illustrates an object's layout in memory.

```
/* 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 block1_of_TestObject {
     int     reference_counts;
     void *  ref_prev;
     void *  ref_next;
     int     a1, a2, a3, a4;
     block2_of_TestObject * next_block;
   };

      /*   and the second block   */

   struct block2_of_TestObject {
     int     a5, a6,.....,a11;
     block3_of_TestObject * next_block;
     // this pointer should be NULL
   };
```

Figure 1. An object's memory layout

The first block of an object is different from the others since it needs to store housekeeping information of that object. The first word keeps the reference counts. The most significant 27 bits of the second word and the whole third word are pointers used to maintain (doubly) linked lists of objects; Finally, the least significant 5 bits of the second word records status information for the garbage collector (e.g. the colour of the object and whether the object is referenced by any root).

In our approach, the reference count for each object is 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). Currently, they share a single word in an object. Moreover, we maintain 2 doubly linked lists: the *tracing-list* and the *white-list*. Any object reachable from the root set must be in and simultaneously only in one of the aforementioned lists. The objects in the "tracing-list" have the colour of black or grey. In order to determine the colour of objects, one additional pointer is introduced for the "tracing-list". As the name indicates, the "white-list" contains only white objects and when a tracing GC cycle is completed all the objects still in the "white-list" are garbage, which will then be moved to another list called the *white-list-buffer* waiting to be reclaimed. On the other hand, dead objects recognized by the reference counting al-

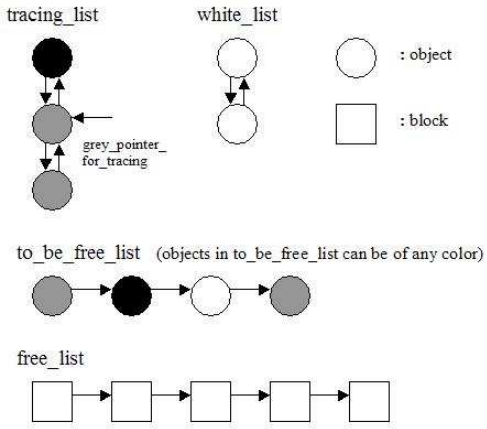

Figure 2. Data structures

gorithm must be put into a linked list called the *to-be-free-list*. Both the *white-list-buffer* and the *to-be-free-list* are processed by the reclaiming task (see section 4.3). Finally, the allocator searches free memory to allocate from the beginning of another linked list called the *free-list*, which is composed of fixed size blocks.

4.2 Write Barriers

The most important two write barriers in our algorithm are the *write-barrier-for-roots* and the *write-barrier-for-objects* from which all the other write barriers are developed. They are invoked automatically when user tasks 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. Assuming a user task is assigning a reference with value "from" to a root variable or an object field with its current value "to", the pseudo code of the corresponding write barriers is given in figure3 and figure4.

By using these write barriers, we maintain not only the reference counting algorithm and root information but also the strong tri-colour invariant [6] which argues that no black object should reference any white object and if so, the white object must be marked grey before the reference can be established. Our algorithm is slightly different in that the user tasks cannot even build references pointing from a grey or white object to another white object. Doing this can reduce the overhead of write barriers by performing less colour checks but introduce more floating garbage[2]. However, we argue that more floating garbage should not be a problem in our algorithm because:

1. The worst case amount of floating garbage in our current algorithm cannot be improved by restoring the original tri-colour invariant and a smaller average amount is of less importance in real-time systems.
2. The reference counting algorithm can help to identify

[2]Floating garbage is garbage that emerges but cannot be identified in the current GC cycle (it is guaranteed to be identified in the next GC cycle).

```
void write_barrier_for_roots(void *from, void *to)
{
   if(from != NULL) {
   //when we assign a valid reference to a root
   {  update(add one) the root count of the object
      referenced by "from";

      if(the root count of the object referenced by
         "from" is one)
      {  mark the corresponding bit in object header
         to announce that this object is directly
         referenced by root;

         if(the object referenced by "from" is in
            "white_list")
         {   unlink it from the "white_list" and
             add the white object to the end of
             "tracing_list";
         }
         else if(which list the object referenced by
                 "from" should belong hasn't been
                 decided)
         {   add it to the beginning of the
             "tracing_list";
         }
      }
   }

   if( to != NULL )
   //when we assign a value 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)
      {
          if(the object referenced by "to" is in
             "white_list" or "white_list_buffer")
          {  unlink it from its current list;
          }
          else
          {  unlink it from the tracing list;
          }
          add it to the "to_be_free_list";
      }
      else if(root count of the object referenced
              by "to" is zero)
      {
          clear the corresponding bit in object
          header to announce that this object
          is no longer directly referenced by
          any root;
      }
   }
}
```

Figure 3. Pseudocode of write barrier for roots

```
void write_barrier_for_objects(void *from, void *to)
{
   if( from != NULL )
   //when we assign a valid reference to a
   //reference field of an object
   {   update(add one) the object count of the
       object referenced by "from";

       if(the object referenced by "from" is in
          "white_list")
       {   unlink it from the "white_list" and
           add the white object to the end of
           "tracing_list";
       }
       else if(which list the object referenced
               by "from" should belong hasn't
               been decided)
       {   add it to the beginning of the
           "tracing_list";
       }
   }

   if(to != NULL)
   //when we assign a value to a reference field
   //that 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" or "white_list_buffer")
          {  unlink it from the its current list;
          }
          else
          {  unlink it from the tracing list;
          }
          add it to the "to_be_free_list";
       }
   }
}
```

Figure 4. Pseudo code of write barrier for objects

some floating garbage.

4.3 Tracing and Reclamation

In our system, tracing and reclamation are performed by different GC tasks: the *tracing task* and the *reclaiming task*, which are executed concurrently.

Whenever the "to-be-free-list" is not empty, the reclaiming task will examine every block (objects are composed of blocks) in it. If any block has any direct child object, the object count of its direct child must be decreased by 1. If both object count and root count are 0, that direct child object will be linked to the rear of the "to-be-free-list". Having been processed, all its direct children, the current block can be reclaimed and the next block will be processed in the same way. After this procedure stops, the "white-list-buffer" will be checked and all the objects currently residing in it will be reclaimed, one by one without any attempt to process their children. Then, the "to-be-free-list" will be processed again. When both lists are empty, the reclaiming task will have nothing to do and therefore suspend itself.

One thing needs to be noticed is that the reference count of an object could be overestimated if it is a part of a cyclic data structure but not found to be dead at the same time as other objects in that structure - since no cyclic garbage will be examined to update the information of its children. This is not a problem because:

1. The objects with overestimated reference counts can be identified as garbage by the tracing task after they die.

2. These objects are already considered as part of the cyclic structure when we perform offline analysis (see section 5).

The periodically released tracing task starts by moving all the objects in the "tracing-list" to the empty "white-list". This is a very efficient operation only involving several reference assignments. Afterwards, the "white-list" will be traversed so that all the objects directly referenced by roots can be moved back to the "tracing-list" and all the other objects can be marked white. Marking begins from the head of the "tracing-list", continuously moving objects from the "white-list" to the "tracing-list", and completes when there are no grey objects in the "tracing-list". We then move all the objects in the "white-list" to the end of the "white-list-buffer" and wait for the next period. The deadline of the tracing task is the same as its period.

In some cases, the tracing task can also end with acyclic garbage in the "white-list". Therefore, the "white-list-buffer" could contain acyclic garbage as well. If such garbage is not scanned before reclamation, its children's reference counts will be overestimated. Consequently, the behaviour of the whole system will be unpredictable. In order to avoid this, the processing of the "to-be-free-list" is always given precedence over the processing of the "white-list-buffer" so that any object in the "white-list-buffer" needs to be scanned is scanned and reclaimed before others are reclaimed.

4.4 Scheduling

Many real-time applications consist of a mixture or periodic hard real-time tasks and aperiodic soft or non real-time tasks running on the same processor. In order to satisfy those tasks' dramatically different requirements, many flexible scheduling approaches have been proposed among

which, dual priority scheduling is an efficient means of identifying and reclaiming spare capacity in favour of soft or non real-time tasks whilst guaranteeing hard deadlines [5]. In this section, we demonstrate how the dual priority scheduling technique can bring the same flexibility to a garbage collected hard real-time system such as ours.

As introduced previously, the tracing task in our system is a periodic task with a deadline. On the other hand, the reclaiming task is always ready to run except when it has no garbage to reclaim. In order to make sure that the processor is not occupied by the reclaiming task all the time, we set a limitation on the amount of work the reclaiming task can perform in one period of the tracing task (see section 5). The amount of work can either be stated as computation time or number of blocks reclaimed. Although the two metrics are equivalent in the worst case, choosing different metrics can still give the system different behaviours in other cases, particularly when the computation time of the reclaiming task is underestimated. Currently, we measure the work amount in terms of the number of blocks reclaimed. Before the number of blocks reclaimed by the reclaiming task in the current period reaches its limit, the reclaiming task is always given a higher priority than the tracing task. Otherwise, the reclaiming task has done enough work and gives way to any other task in the system. At the next release of the tracing task, this work amount limit will be reset. Thus, we can consider the reclaiming task as a special periodic task as well.

Because we only need the GC tasks to be schedulable rather than responsive, they ought to be executed as infrequently as possible and also as late as possible. On the other hand, if the GC tasks miss their deadlines, hard user real-time tasks could be blocked for arbitrary time due to the lack of free memory. Therefore, we consider GC tasks as periodic hard real-time tasks.

In order to discuss the scheduling approach, the properties of user tasks must be defined first:

1. Priorities are split into 3 bands: Upper, Middle and Lower [5].

2. Hard real-time tasks (including both user and GC tasks) are released periodically and execute in either the lower or upper band.

3. Soft real-time tasks are released aperiodically and their priorities are always within the middle band.

4. Soft real-time tasks neither produce any cyclic garbage nor allocate memory from the heap. Eliminating this limitation is part of our future work (see section 7).

In the dual priority algorithm, a hard real-time task can have two priorities one in the upper and one in the lower band. Upon its release, it executes at its lower band priority so giving preference to the soft or non real-time tasks in the middle band. Moreover, each hard real-time task has a promotion time which is the release time plus the difference between its deadline and the worst case response time. When the given promotion time has elapsed, the hard real-time task is promoted to its higher band priority therefore guaranteeing its deadline. If, however, the hard real-time task is ever activated before the promotion time has elapsed, the promotion time should be extended by the length of that interval, so that spare capacity can be reclaimed [5].

Applying this technique to our GC tasks only needs a few trivial modifications to the original algorithm. First, we consider the two GC tasks as a whole and define the promotion time as the release time plus the difference between the period and the worst case response time of the tracing task. Secondly, instead of giving arbitrary priorities to hard real-time tasks in the lower band, we need to maintain the same priority order of hard real-time tasks in both upper and lower band. Moreover, as the reclaiming task finishes its compulsory work, it goes to the lowest priority in the lower band and returns to its original lower band priority upon the next release of the tracing task. We will introduce how to calculate priorities for the GC tasks in section 5.

Another approach we use to reclaim spare capacity is to set a lower limit on the reclaiming task's work amount if, in the previous tracing period, the reclaiming task did any extra work.

Compared to many other concurrent GC mechanism, our approach can now ensure that the GC tasks get enough system resources in each period. However, this doesn't necessarily mean that the user tasks' memory requirements can always be satisfied because the user tasks can sometimes consume all the free memory before the GC tasks can reclaim enough garbage objects, although they may reclaim enough or even more in the near future. To guarantee that the real-time tasks with higher priorities than the reclaiming task (in the same band) can never be blocked because of the lack of free memory, we require that the GC tasks should always try to preserve enough free memory for them [8]. That is, before the promotion time, as long as the amount of free memory is lower than a certain value called F_{pre} (see section 5), the priority of the GC tasks should be promoted. Otherwise, it should be executed at their original priority until the promotion time.

5 Static Analysis

In this section, we calculate the scheduling parameters for the GC tasks. Table 2 summarises the notation used in this section.

In order to derive the parameters, we need to calculate how much work the GC tasks need to perform. This is dis-

Symbols	Definitions
P	the set of user tasks in the whole system
L_i	the amount of live memory just before the ith release of the tracing task
F_i/F_{min}	the amount of free memory just before the ith release of the tracing task and its minimum value
A_i	the amount of allocated (non-free) memory just before the ith release of the tracing task
CGG_i/CGG_{max}	the amount of cyclic garbage generated in cycle i and its maximum value
FCG_i	the amount of floating cyclic garbage emerged in cycle i
RG_i	the amount of acyclic garbage generated in cycle i
a_i/a_{max}	new memory allocated in cycle i and its maximum value
R_i	the total amount of garbage reclaimed in cycle i
G_i	the total amount of garbage that can be recognized by the end of cycle i
L_{max}	the upper bound of live memory consumption of the whole system
H	the size of heap
$hp(GC)$	the set of all the users tasks with higher priorities than the GC tasks (promoted)
F_{pre}	the amount of free memory the system should preserve for the user tasks with higher priorities than the reclaiming task (promoted)
R_{pre}	the worst case response time of the reclaiming task (promoted) to reclaim as much memory as the user tasks allocate during that time
T_j	the period of the user task j
C_j	the worst case execution time of the user task j
a_j	the worst case memory allocation executed in one release of the user task j
cgg_j	the worst case amount of cyclic garbage emerged in one release of the user task j
L_j	the worst case amount of live memory of the user task j
RR	the time needed to reclaim one unit of memory in the worst case
TR	the time needed to trace one unit of memory in the worst case
MWR	the time needed to mark one object white in the worst case
NTM	the worst case number of objects in $H - F_{pre}$. This is also the worst case number of objects need to be marked white in a cycle
D	the deadline and also period of both reclaiming and tracing tasks

Table 2. Notation definition

cussed first.

Minimum free memory needed

First, we give some simple formulae without proof since they are self-explanatory.

$$H = L_{i+1} + F_{i+1} + FCG_i + G_i - R_i \tag{1}$$

$$A_{i+1} = L_{i+1} + FCG_i + G_i - R_i \tag{2}$$

which means the dead but not yet reclaimed objects at the beginning of cycle $i+1$ include the floating cyclic garbage of cycle i and the garbage not reclaimed due to work limitation on the reclaiming task.

$$G_i = RG_i + FCG_{i-1} + CGG_i - FCG_i + G_{i-1} - R_{i-1} \quad for\ \ i > 1 \tag{3}$$

$$G_0 = RG_0 + CGG_0 - FCG_0 \tag{4}$$

which means the garbage objects that can be recognized by the end of cycle i include all the acyclic garbage of cycle i, all the floating cyclic garbage of cycle $i-1$(zero if $i = 0$), a portion of cyclic garbage emerged in cycle i and all the garbage abandoned by the reclaiming task in cycle $i-1$ (zero if $i = 0$).

$$L_{i+1} = L_i + a_i - RG_i - CGG_i \tag{5}$$

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

$$A_{i+1} = A_i + a_i - R_i \tag{6}$$

Having got the above formulae, we can now try to calculate the minimum amount of free memory needed at the beginning of each cycle. 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 get $L_{i+1} - L_i \leq 0$. Applying equation 5 to this gives:

$$a_i - RG_i - CGG_i \leq 0 \tag{7}$$

and therefore,

$$a_i - RG_i \leq CGG_i \tag{8}$$

From equation 6, we get:

$$A_{i+1} - A_i = a_i - R_i \tag{9}$$

If the value of R_i is not smaller than that of a_i, there will be no cumulation of allocated memory at the end of cycle i. As discussed previously, we set an upper bound on the value of R_i. The initial value of such an upper bound for each cycle is a_{max} since by reclaiming a_{max} garbage, $A_{i+1} - A_i$ is already guaranteed not to be greater than zero. Cumulation of allocated memory can only happen when $R_i < a_i$. In the worst scenario, the GC tasks in cycle i only reclaims acyclic garbage of that cycle and the amount of such garbage is less than the allocations in that cycle. That is, $R_i = RG_i$ and $RG_i < a_i$. Thus, in the worst case,

$$A_{i+1} - A_i = a_i - RG_i \tag{10}$$

Applying inequality 8 to the above equation, we can get:

$$A_{i+1} - A_i \leq CGG_i \leq CGG_{max} \tag{11}$$

which means that if the GC tasks can provide as much free memory as CGG_{max} at the beginning of any 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 must be able to provide at least $F_{pre} + CGG_{max}$ free memory at the beginning of any cycle.

Minimum free memory provided

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

$$F_{i+1} = H - (L_{i+1} + FCG_i + G_i - R_i) \tag{12}$$

In order to calculate F_{min}, the upper bound of $L_{i+1} + G_i - R_i + FCG_i$ (or A_{i+1} in another word) should be determined first:

Theorem 1. *If the deadlines of GC tasks are guaranteed, the amount of allocated (non-free) memory at the very beginning of any release of the tracing task is bounded by $L_{max} + CGG_{max}$. (the proof of this theorem is included in appendix)*

Consequently, $L_{i+1} + G_i - R_i + FCG_i \leq L_{max} + CGG_{max}$ and equation 12 can be modified as:

$$F_{min} = H - L_{max} - CGG_{max} \tag{13}$$

Deadline and Priority

To ensure that the user tasks execute without any blocking due to garbage collection, the minimum amount of free memory at the beginning of each cycle must be satisfied. Therefore,

$$F_{pre} + CGG_{max} = H - L_{max} - CGG_{max} \tag{14}$$

and thus,

$$CGG_{max} = \frac{H - L_{max} - F_{pre}}{2} \tag{15}$$

As defined previously, $CGG_i \leq CGG_{max}$ so we can get:

$$CGG_i \leq \frac{H - L_{max} - F_{pre}}{2} \tag{16}$$

Assuming in the worst case that all the hard real-time user tasks arrive at the same time (soft real-time user tasks don't contribute to CGG_i), 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, the deadline of the GC tasks can finally be calculated.

$$\sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \leq \frac{H - L_{max} - F_{pre}}{2} \tag{18}$$

Since D is mainly determined by the heap size and the rate of cyclic garbage cumulation rather than the rate of allocation, the value of D could be much higher than its pure tracing counterpart. Therefore, tracing would be invoked less frequently.

Next, we present how to determine F_{pre}:

$$R_{pre} = \sum_{j \in hp(GC)} \left[\left\lceil \frac{R_{pre}}{T_j} \right\rceil (C_j + RR \cdot a_j) \right] \tag{19}$$

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

Notice that to calculate R_{pre} and F_{pre}, we need to know the GC tasks' priorities 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 tasks so there's obviously a recursion. In order to resolve this recursion, we need to treat equation 19, 20 and 18 as a group.

At the very beginning, assume that the priorities of GC tasks are the lowest two priorities among all the hard real-time tasks (within the same band). Then, we can get the corresponding R_{pre}, F_{pre}, D and consequently the priorities corresponding to the D. If the GC priorities are the same as we assumed, that is the result. Otherwise, we should use the new GC priorities to recalculate R_{pre}, F_{pre}, D and the

new priorities until the old version and the new version of the GC tasks' priorities equal each other[3].

WCET and Response Time

So far, we have been able to estimate the WCETs and the worst case response time of the GC tasks: $WCET_{tracing}$, $WCET_{reclaiming}$ and R_{gc} respectively.

$$WCET_{tracing} = TR \cdot (L_{max} + CGG_{max}) + MWR \cdot NTM \quad (21)$$

$$WCET_{reclaiming} = RR \cdot a_{max} \quad (22)$$

$$R_{gc} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{gc}}{T_j} \right\rceil \cdot C_j \right) + WCET_{tracing} + WCET_{reclaiming} \quad (23)$$

We can now compare the GC tasks' worst case response time R_{gc} with their deadline D. If $R_{gc} \leq D$, the GC tasks are schedulable. Moreover, we can also use the parameters of the GC tasks (e.g. the period, D, $WCET_{tracing}$ and $WCET_{reclaiming}$) to estimate the response time of the hard real-time tasks with lower priorities than those of the GC tasks (within the same band). If their response time values are smaller than their deadlines, they are also schedulable. Otherwise, the designer has to redesign the system to reduce either the memory usage or the WCET of some of the user tasks.

6 Empirical Results

We have implemented our algorithm by modifying GCJ compiler (the GNU compiler for the Java language, version 3.3.3) and jRate library (version 0.3.6)[4], which is a RTSJ-compliant real-time extension to the GCJ compiler. All the results presented in this section were obtained on a 1.5 GHz Intel CPU with 1MB L2 cache and 512MB RAM, running SUSE Linux 9.3 together with "linux_lib" architecture MaRTE OS version 1.57.

For the purpose of this report, impacts of our algorithm on the user task performance are studied first. This includes testings on worst case computation time of both object and root write barriers (see figure 5). In order to perform the analysis described in the previous section, RR, TR and MWR must be obtained from experiments as well (see figure 6). All the testings are performed 1000 times in a row through the worst case path and the results are presented in terms of worst value, best value, average value and 99% worst value[4]. Notice that the computation time of operations that are used to protect critical sections are not included in the aforementioned tests because first, this is a platform dependent overhead which could be as low as several instructions or as high as several function calls for each pair of them; Secondly, how frequently such operations are executed depends on the requirements of applications.

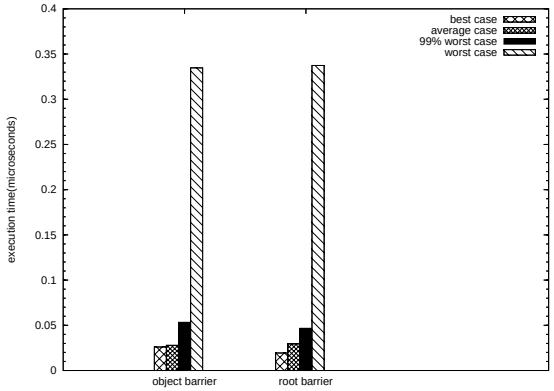

Figure 5. Write Barrier Computation Time

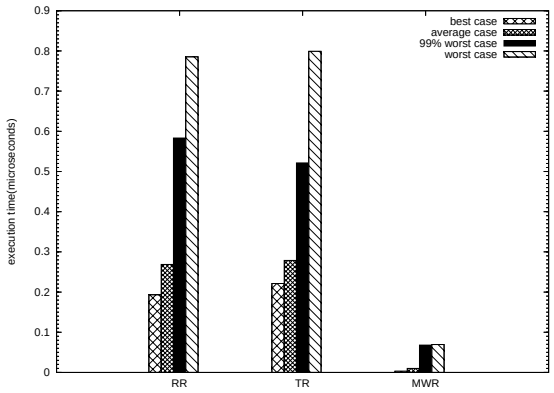

Figure 6. RR, TR and MWR Results

As defined previously, RR is the worst case computation time needed to reclaim a memory block. If any acyclic garbage exists at the free memory producer release point, this is exactly the free memory producer latency. Otherwise, the free memory producer latency of our approach is comparable with that of a pure tracing collector.

With these information, we can perform analysis on hard real-time task sets given in table 3 and 4. All the values in tables hereafter are measured in milliseconds for time or bytes for space. Priorities are assigned according to DMPO.

[3]Unfortunately, this process sometimes does not converge, so we may have to use other priority assignment techniques for the GC tasks, for example, perhaps using some global optimization approaches such as genetic algorithms .

[4]99% worst value means the highest value below the top 1% values.

Tasks	C_j	$T_j(D_j)$	a_j	cgg_j	L_j
1	1	5	320	0	320
2	2	10	960	192	1600
3	5	50	1920	320	3200
4	12	120	5760	640	9600

Table 3. Hard Realtime Task Set 1

Tasks	C_j	$T_j(D_j)$	a_j	cgg_j	L_j
1	1	5	640	192	640
2	2	10	1920	576	3200
3	5	50	3840	1152	6400
4	12	120	11520	3456	19200

Table 4. Hard Realtime Task Set 2

Parameters	task1	task2
H	34752(1.48L_{max})	80768(2.19L_{max})
F_{pre}	3520	7040
D	120	120
a_{max}	30720	61440
$WCET_{tracing}$	5.14	11.22
$WCET_{reclaiming}$	3.07	6.13
R_{gc}	24.21	38.35
utilization	6.84%	14.46%

Table 5. GC parameters

Furthermore, a non-real-time task which simply performs an infinite loop executes at a priority lower than all the tasks in table 3 or 4. For simplicity and without loss of generality, only the GC tasks are scheduled according to dual-priority algorithm. We'll apply dual-priority scheduling to other hard real-time tasks in the near future.

The execution time of a pair of operations that protect critical sections on our platform is 2.4 microseconds according to our test. As a result, we adjust the RR to 3.19 microseconds, TR to 3.20 microseconds and finally MWR to 2.47 microseconds. To perform static analysis, L_{max} is calculated first according to [10]. The maximum amount of live memory of all the hard real-time tasks in task set 1 and 2 are estimated as 14080 and 27520 bytes respectively and we set the maximum amount of static global live memory to be 9344 bytes for both task sets. Therefore, the total amount of live memory cannot exceed 23424 bytes for task set 1 or 36864 bytes for task set 2. By performing static analysis with the given heap sizes, we assign GC tasks with promoted priorities between task 3 and 4 for both task sets and all the other parameters needed by the GC tasks are presented in table 5.

Given these parameters, we execute both task sets with our garbage collector to justify the correctness of our algorithm and static analysis. Two different GC promotion time are selected for each task set to compare their impacts on the

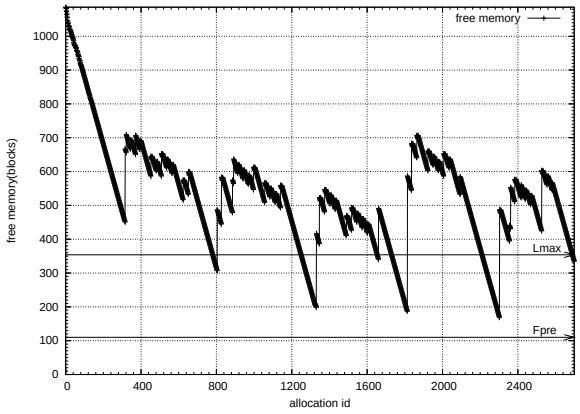

Figure 7. Task set 1 with Promotion time 10ms

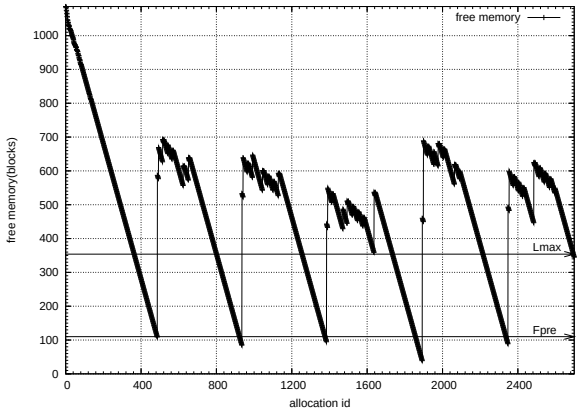

Figure 8. Task set 1 with Promotion time 90ms

memory usage of our system. Both configurations for both task sets can generate safe execution which means no deadline is missed and no task is blocked by the garbage collector. The memory usages of both task sets are presented in figure 7, 8, 9 and 10[5].

These figures illustrates the fundamental difference between our approach and a pure tracing one, which is that the amount of free memory in our system no longer decreases monotonically in each GC period. This is because our approach possesses a relatively lower free memory producer latency. Not only tracing but also reclamation can be performed incrementally. Secondly, the later the promotion time is, the smaller the space margin we'll have. This supports our argument in section 4.4, which suggests that users tasks should be given preference over GC tasks by squeezing the heap harder.

[5]Allocation id x means the xth allocation.

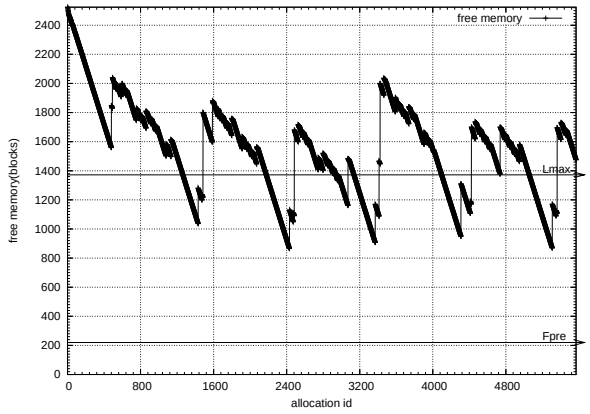

Figure 9. Task set 2 with Promotion time 10ms

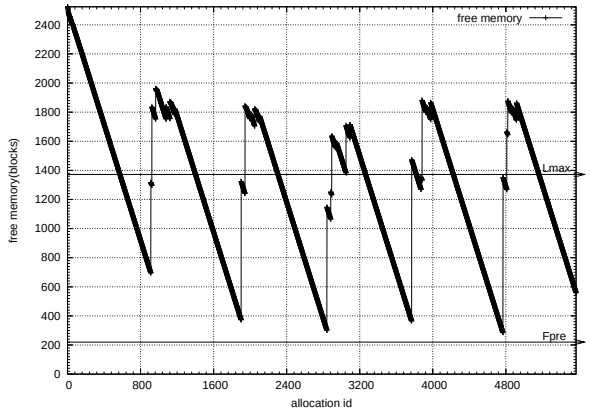

Figure 10. Task set 2 with Promotion time 80ms

To compare our algorithm with a pure tracing one, we choose [12] as a rival. According to their analysis, one cannot calculate a deadline for a pure tracing GC task with task set 1 but for task set 2, we can expect the longest deadline to be 20 milliseconds which implies a 56.1% utilization[6]. On the other hand, if we assume the priority and utilization of the pure tracing GC task are the same as our GC tasks, the pure tracing period will be 75.13 milliseconds for task set 1 or 77.6 milliseconds for task set 2. According to [12], this corresponds to a heap of 68224 bytes ($2.91L_{max}$) for task set 1 or 126464 bytes ($3.43L_{max}$) for task set 2. By contrast, the heap sizes of our system vary from $1.48L_{max}$ to $2.19L_{max}$.

[6]We assume that the pure tracing GC task has the same WCET as that of our tracing task.

7 Conclusion and Future Work

This report has illustrated the inherent limitation of currently used real-time tracing garbage collectors and proposes two new metrics that can better describe the overall real-time capability of a garbage collector. These metrics motivate the development of a hybrid approach to garbage collection. Such an approach has been described along with its scheduling parameters, static analysis and some empirical results. The benefit of our approach includes:

- Due to the contribution of reference counting algorithm and the fine grained model, our approach can achieve relatively low memory consumption.
- 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 greater amount of unnecessary system resource consumption is avoided.
- Our approach is flexible enough so that the GC tasks can adapt to different applications and heap sizes automatically: the smaller the heap size is or the more cyclic garbage, the shorter the deadline could be. For a system which is mainly composed of acyclic data structures, the deadline of the GC tasks could be very long. However, for a system which is mainly composed of cyclic data structures, our approach gracefully degrades. Fortunately, the above analysis provides the designers with a way to quantitatively determine whether our approach is suitable for their application or not.
- We can provide real-time guarantees for all the hard real-time tasks as in non-garbage-collected real-time systems.
- All the hard real-time tasks follow dual priority scheduling approach so spare capacity can be reclaimed and the responsiveness of soft real-time tasks is improved.

Our current work is now focused on the soft real-time tasks and how their impact on the overall memory consumption can be identified and kept under control.

References

[1] D. F. Bacon, P. Cheng, and V. Rajan. The metronome: A simpler approach to garbage collection in real-time systems. In *Proceedings of OTM 2003 Workshops*, pages 466–478.

[2] H. G. Baker. List processing in real time on a serial computer. *Communications of the ACM*, 21(4):280–294.
[3] A. Borg, A. Wellings, C. Gill, and R. K. Cytron. Real-time memory management: Life and times. In *Proceedings of Euromicro 2006*.
[4] A. Corsaro and D. 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*.
[5] R. Davis and A. Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*.
[6] E. W. Dijkstra. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975.
[7] B. Goldberg. Incremental garbage collection without tags. In *Proceedings of the 4th European Symposium on Programming*.
[8] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University.
[9] R. L. Hudson and J. E. B. Moss. Incremental collection of mature objects. In *Proceedings of the International Workshop on Memory Management*.
[10] 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.
[11] T. Ritzau. Hard real-time reference counting without external fragmentation. In *Java Optimization Strategies for Embedded Systems Workshop at ETAPS 2001*.
[12] S. G. Robertz and R. Henriksson. Time-triggered garbage collection—robust and adaptive real-time gc scheduling for embedded systems. In *Proceedings of LCTES 2003*, pages 93–102.
[13] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for java. In *Compilers, Architectures and Synthesis for Embedded systems(CASES2000)*.
[14] F. Siebert. *Hard Realtime Garbage Collection in Modern Ojbect Oriented Programming Languages*. PhD thesis, University of Karlsruhe, May 2002.

Appendix

Theorem 1. *If the deadlines of GC tasks are guaranteed, the amount of allocated (non-free) memory at the very beginning of any release of tracing task is bounded by* $L_{max} + CGG_{max}$.

Proof. Since at the first release, this is automatically guaranteed, proving the above theorem is equivalent to proving: $L_{i+1} + G_i - R_i + FCG_i \leq L_{max} + CGG_{max}$ with $i \geq 0$.

When $G_i - R_i = 0$,

$$L_{i+1} + G_i - R_i + FCG_i = L_{i+1} + FCG_i \quad (24)$$

In the worst case scenario, all the cyclic garbage objects emerged in cycle i become floating so the upper bound of FCG_i is the same as that of CGG_i, i.e. CGG_{max}. Therefore,

$$L_{i+1} + G_i - R_i + FCG_i \leq L_{max} + CGG_{max} \quad (25)$$

On the other hand, when $G_i - R_i > 0$, R_i reaches its upper bound a_{max}. Consequently, we only need to prove that:

$$L_{i+1} + G_i - a_{max} + FCG_i \leq L_{max} + CGG_{max} \quad (26)$$

since $L_{i+1} = L_i + a_i - RG_i - CGG_i$ and $G_i = RG_i + FCG_{i-1} + CGG_i - FCG_i + G_{i-1} - R_{i-1}$, we can get

$$L_{i+1} + G_i - a_{max} + FCG_i = L_i + G_{i-1} - R_{i-1} + FCG_{i-1} + a_i - a_{max} \quad (27)$$

Because a_i is defined to be smaller than a_{max}, $L_{i+1} + G_i - a_{max} + FCG_i$ is bounded by:

$$L_{i+1} + G_i - a_{max} + FCG_i \leq L_i + G_{i-1} - R_{i-1} + FCG_{i-1} \quad (28)$$

As we have proved, $L_i + G_{i-1} - R_{i-1} + FCG_{i-1}$ is bounded by $L_{max} + CGG_{max}$ when $G_{i-1} - R_{i-1} = 0$. Thus, inequality 26 is proved in the situation where $G_{i-1} - R_{i-1} = 0$.

If however, $G_{i-1} - R_{i-1} > 0$, $L_i + G_{i-1} - R_{i-1} + FCG_{i-1}$ is bounded by $L_{i-1} + G_{i-2} - R_{i-2} + FCG_{i-2}$. Therefore, we only need to prove that $L_1 + G_0 - R_0 + FCG_0 \leq L_{max} + CGG_{max}$. Since this has already been proved if $G_0 - R_0 = 0$, we only consider the situation when $G_0 - R_0 > 0$.

Based on equations 4 and 5, the following equation can be developed:

$$L_1 + G_0 - R_0 + FCG_0 = L_0 + a_0 - a_{max} \leq L_0 \quad (29)$$

and therefore,

$$L_1 + G_0 - R_0 + FCG_0 \leq L_{max} \leq L_{max} + CGG_{max} \quad (30)$$

which completes the proof. □