Hard Real-time Hybrid Garbage Collection with Low Memory Requirements

Yang Chang and Andy Wellings
University of York, UK, {yang, andy}@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 paper explores the fundamental reason for this problem and proposes a new performance indicator for the evaluation of real-time garbage collection algorithms. Use of this performance indicator motivates an algorithm that combines both reference counting and mark-and-sweep techniques. In the presence of our collector, a garbage collected hard real-time system can achieve the correct balance of time-space tradeoff with less effort. In order to provide both temporal and spatial guarantees needed by a hard real-time application, an offline analysis is developed and integrated into the current response time analysis framework. Moreover, the use of dual priority scheduling of the garbage collection tasks 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. 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 among which, garbage collectors monitor the memory and automatically release chunks that are no longer being used. They free the programmer from the burden of memory management but are complex to implement and difficult to analyze.

Most garbage collectors now in use are tracing collectors. Since applications 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 the 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, 16, 9]. 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 paper, 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 a new performance indicator for describing the overall real-time capability of a garbage collector. The use of this performance indicator 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 tasks are executed concurrently with the user tasks and scheduled using the dual priority scheduling algorithm, which can allow the reclamation of spare capacity in the whole system [6].

The remainder of this paper 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 performance indicator. Section 3 briefly introduces the basic idea of our approach. Section 4 reveals more details of our algorithm and the scheduling method. Section 5 presents a static analysis of our garbage collected hard real-time system. Section 6 illustrates the performance of our collector. Finally, we present our conclusions and future work.

2 The Need for A New Performance Indicator

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 large, 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 system running out of memory or requiring 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.

To make this clear, a simple example is given below: 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.

Recent research on real-time garbage collection algorithms such as [1, 10, 14, 9] can only achieve predictability and low worst-case latency by preserving significant redundant space although they all struggle to keep it low. We argue that this is due to the fact that they are all high granularity tracing algorithms, which need to identify live objects before reclaiming any garbage. In the worst case situation where live memory reaches its upper bound, the collector has to scan at least all the live memory before reclamation. Irrespective of the scheduling algorithm, any attempt to give significantly less computation resource to a tracing collector will raise the memory requirement dramatically [2, 10, 14, 9].

To allow for better design of real-time garbage collection algorithms, we define a new performance indicator which characterises the granularity of a 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 new free memory, irrespective of its size, is made available to the allocator.

Moreover, the free memory producer is only eligible to execute 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 this point in time the *Free Memory Producer Release Point* [1]. Notice that a free memory producer does not 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 large enough to hold all the garbage objects and the live ones.

Now, we can define the performance indicator:

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

Garbage Collection Algorithm	Latency is a function of
Conventional Reference Counting	$garbage\_set\_size$
Deferred Reference Counting	$object\_size$
Non-copying Tracing	L_{max}
Copying Tracing	L_{max}

Table 1. Free Memory Producer Complexities

Definition 2 The *Free Memory Producer Latency* of a garbage collector is the worst case execution time of its free memory producer.

Bear in mind that the free memory producer latency is an indictor to the overall real-time performance of a garbage collector. It does not necessarily directly relate to the real latency experienced by any user task.

Lower free memory producer latency always means 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 redundant memory needed is smaller. Assuming throughputs are the same, the garbage collection algorithm which has lower free memory producer latency is more likely to be appropriate for real-time systems.

As can be seen in table 1, the free memory producer latencies of tracing algorithms are functions of the maximum amount of live memory (L_{max}) while those of reference counting algorithms are functions of either the total size of the garbage set or the size of the garbage object being processed. At first glance, the free memory producer latency of reference counting, particularly deferred reference counting, is very promising. However, "$garbage\_set\_size$" and "$object\_size$" can also be huge, 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.

3 The Hybrid Approach Overview

One way to improve reference counting algorithms is to change their garbage collection granularity. Siebert and Ritzau [15, 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 an algorithm "fine grained reference counting"), its free memory producer has a complexity of 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 systems compared with other reference counting algorithms. Ritzau's work [11] is such an algorithm. 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 hard real-time scheduling due to its problematic behaviour during bursty allocations and the fact that it cannot reuse any form of spare capacity.

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 is 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 the total memory usage (44.21% to 100.00% with the average of 81.52%) [8].

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.

4 The Hybrid Garbage Collection Algorithm

4.1 Data Structures

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 [15].

All the blocks of an object use their last word to store the link to the next block. 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

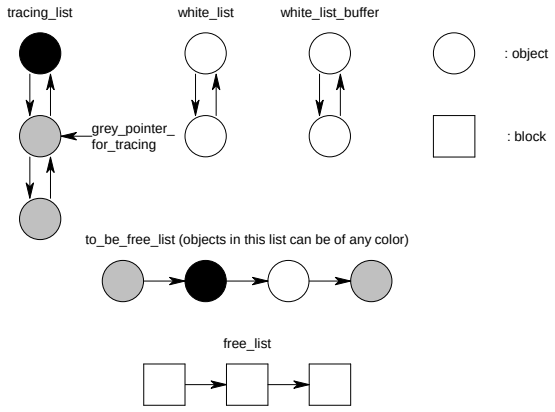

Figure 1. Data Structures

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.

In order to achieve the correct synchronization between the user tasks and a tracing garbage collector, the strong tri-colour invariant [7] must be maintained. It 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. As illustrated in figure 1, we maintain 3 doubly linked lists: *tracing-list*, *white-list* and *white-list-buffer*. Any object reachable from the root set must be in and simultaneously only in one of the first two lists. The objects in the "tracing-list" are either black or grey (already found to be alive by the tracing collector). In order to determine whether an object in the "tracing-list" has already been scanned (black) or not (grey), one additional pointer is introduced for the "tracing-list". As the name indicates, the "white-list" contains only white objects (which means they are potentially dead) and when a tracing GC cycle is completed all the objects still in the "white-list" are garbage, which will then be moved to the "white-list-buffer" waiting to be reclaimed. On the other hand, dead objects recognized by the reference counting algorithm must be put into a linked list called the *to-be-free-list*. 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

Write barriers are code added by a compiler or a runtime system to execute before certain reference assignments. Although it can also be implemented in the hardware level, our algorithm currently adopts a software approach. 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. The main difference between the two write barriers is that root barriers can recognize and manipulate the situations where an object is referenced by a root for the first time or disconnected from all the roots. Assuming a user task is assigning a reference with value "from" to a root variable with its current value "to", the pseudo code of the corresponding write barrier is given in figure2.

By using these write barriers, we maintain not only the reference counting algorithm and root information but also the strong tri-colour invariant. However, 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 most acyclic floating garbage.

4.3 Tracing and Reclamation

In our system, tracing and reclamation are performed by different GC tasks: the *reclaiming task* and the *tracing task*, which are executed concurrently at adjacent priorities (the reclaiming task has the higher priority).

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 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

[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 2. Pseudocode of Write Barrier for Roots

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 a 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.

4.4 Scheduling

Many real-time applications consist of a mixture of 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 [6]. 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 so that its WCET can be assured (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, 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 do not need the GC tasks to be very responsive, they ought to be executed as infrequently as possible and also as late as possible so that some user tasks for which high responsiveness is of great importance can get the resource needed (particularly, CPU time) earlier. On the other hand, delaying the GC tasks can make more space to be occupied by the dead objects. If the GC tasks miss their deadlines, hard real-time user tasks could be blocked for arbitrary time due to the lack of free memory. Therefore, we design the GC tasks as dual-priority scheduled 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 [6].
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 a 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 its release time plus the difference between its deadline and its 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 [6].

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. The above requirements are introduced to make sure that the reclaiming task always has higher priority than the tracing task (in either band) before it finishes its compulsory work. This is essential for the WCET estimation of our tracing task. 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.

Now, our approach can only ensure that the GC tasks get enough system resources in each period. 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 [9]. That is, before the promotion time, if the amount of free memory is lower than a certain value called F_{pre} (see section 5), the priorities of the GC tasks should be promoted. Otherwise, they should be executed at their original priorities until the promotion time.

5 Static Analysis

In this section, we calculate the scheduling parameters for the GC tasks[3]. Table 2 summarises the notation used in this paper.

Minimum free memory needed

Due to the existence of reference counting, the cumulation of allocated memory throughout any GC cycle cannot exceed CGG_{max} given that L_{max} has been reached, 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

First, a theorem is given without proof. More information can be found in [4].

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* $L_{max} + CGG_{max}$.

Consequently, our GC tasks can maintain at least $H - L_{max} - CGG_{max}$ free memory at the beginning of any GC cycle. To ensure that the user tasks execute without any blocking due to garbage collection, the minimum amount of free memory needed must be satisfied. Therefore,

$$F_{pre} + CGG_{max} = H - L_{max} - CGG_{max} \qquad (1)$$

[3] Due to the limitation on space, we discuss our analysis process without proof in this paper. However, you can find all those proofs in [4].

Symbols	Definitions
P	the set of user tasks in the whole system
L_{max}	the upper bound of live memory consumption of the whole system
CGG_i/CGG_{max}	the amount of cyclic garbage generated in cycle i and its maximum value
a_i/a_{max}	new memory allocated in cycle i and its maximum value
H	the size of heap
$hp(GC)$	the set of all the user 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

Deadline and Priority

By transforming equality 1, we can get:

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

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

$$CGG_i \leq \frac{H - L_{max} - F_{pre}}{2} \tag{3}$$

Assuming in the worst case that all the hard real-time user tasks arrive at the same time (soft real-time user tasks do not contribute to CGG_i), CGG_i can be represented as:

$$CGG_i = \sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \tag{4}$$

Applying this to inequality 3, the deadline of the GC tasks can finally be calculated.

$$\sum_{j \in P} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot cgg_j \right) \leq \frac{H - L_{max} - F_{pre}}{2} \tag{5}$$

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}:

$$R_{pre} = \sum_{j \in hp(GC)} \left\lceil \left\lceil \frac{R_{pre}}{T_j} \right\rceil (C_j + RR \cdot a_j) \right\rceil \tag{6}$$

$$F_{pre} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{pre}}{T_j} \right\rceil \cdot a_j \right) \tag{7}$$

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 is obviously a recursion. In order to resolve this recursion, we need to treat equation 6, 7 and 5 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[4].

WCET and Response Time

Given the above, we are able to estimate the WCETs and the worst case response time of the GC tasks: $WCET_{tracing}$, $WCET_{reclaiming}$ and R_{gc} respectively.

$$WCET_{tracing} = TR \cdot (L_{max} + CGG_{max}) + MWR \cdot NTM \tag{8}$$

$$WCET_{reclaiming} = RR \cdot a_{max} \tag{9}$$

$$R_{gc} = \sum_{j \in hp(GC)} \left(\left\lceil \frac{R_{gc}}{T_j} \right\rceil \cdot C_j \right) + WCET_{tracing} + WCET_{reclaiming} \tag{10}$$

[4]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 .

6 Performance Evaluation

We have implemented our algorithm by modifying the GCJ compiler (the GNU compiler for the Java language, version 3.3.3) and the jRate library (version 0.3.6)[5], 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[12].Although our implementation is based on jRate, its scoped memory was not used in any tests presented in this paper.

For the purpose of this paper, 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 3). In order to perform the analysis described in the previous section, RR, TR and MWR must be obtained from experiments as well (see figure 4). 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[5]. 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.

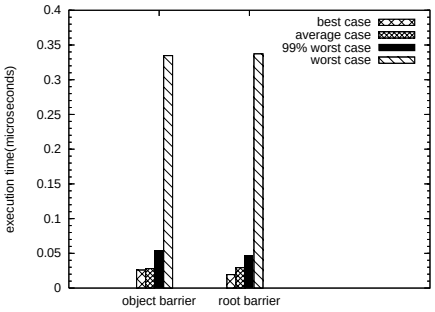

Figure 3. Write Barrier Computation Time

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.

With the above information, we can perform analysis on two synthetic 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. Furthermore, a non-real-time task which

[5]99% worst value means the highest value below the top 1% values.

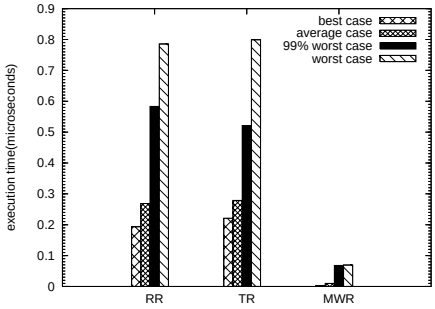

Figure 4. RR, TR and MWR Results

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

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 the dual-priority algorithm. We will 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 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

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
GC utilization	6.84%	14.46%

Table 5. GC Parameters

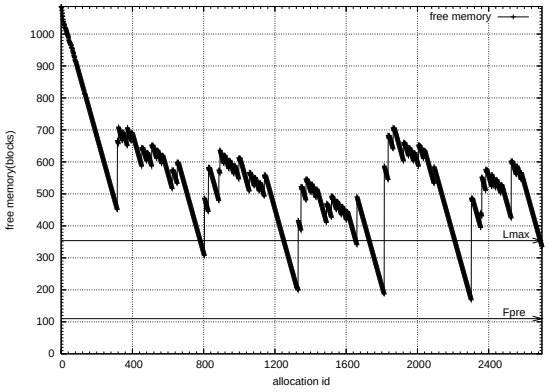

Figure 5. Task Set 1 with Promotion Time 10ms

are selected for each task set to compare their impacts on the 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. This is theoretically proved rather than an empirical observation. The memory usages of both task sets are presented in figure 5, 6, 7 and 8[6].

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 will have. This supports our argument in section 4.4, which suggests that user tasks should be given preference over the GC tasks by squeezing the heap harder.

We choose [13] as an example of pure tracing collectors with which we compare our algorithm. In their approach, a mark-and-sweep garbage collector is performed by a segregated periodic real-time task scheduled in the same way as any other real-time task. They also presented a static analy-

[6]Allocation id x means the xth allocation.

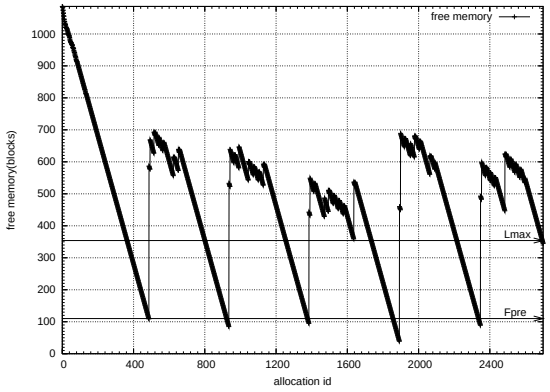

Figure 6. Task Set 1 with Promotion Time 90ms

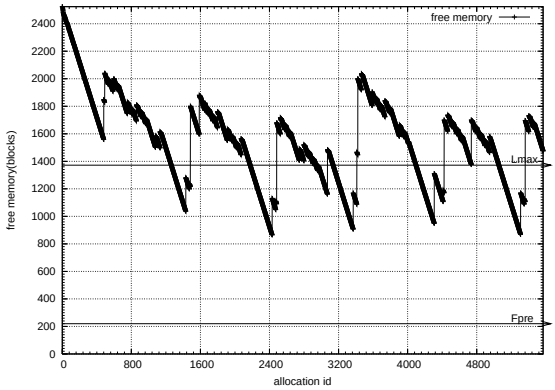

Figure 7. Task Set 2 with Promotion Time 10ms

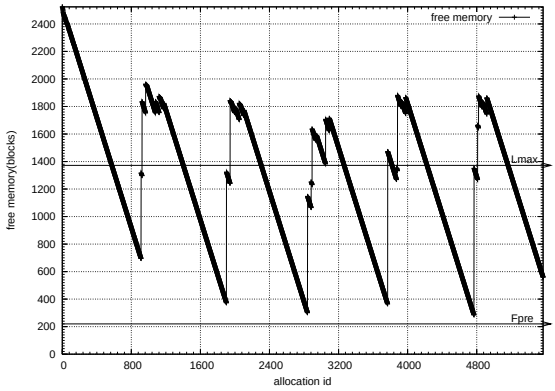

Figure 8. Task Set 2 with Promotion Time 80ms

sis which can provide the estimations of the period, deadline and priority of the GC task given a specific application and heap size.

According to their analysis, our task set 1 along with a pure tracing GC task is infeasible since $(\forall D)(\sum_{j \in task1} \left(\left\lceil \frac{D}{T_j} \right\rceil \cdot a_j \right) > (H - L_{max})/2)$ but for task set 2, we can expect the longest deadline to be 20 milliseconds which implies a 56.1% GC utilization[7]. On the other hand, if we assume the priority and utilization of the pure tracing GC task are the same as those of 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 the static analysis in [13], 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, our heap sizes are theoretically proved as $1.48L_{max}$ or $2.19L_{max}$. Since the heap sizes and L_{max} for both approaches are presented with the same per-object and per-block space overheads, the ratios between the heap sizes and the L_{max} can be compared between the two approaches.

7 Conclusions and Future Work

This paper has illustrated the inherent limitation of currently used real-time tracing garbage collectors and proposes a new performance indicator that can better describe the overall real-time capability of a garbage collector. This indicator motivates the development of a hybrid approach to garbage collection, which needs smaller redundant memory and less frequent whole heap tracing and eliminates root scanning. Such an approach has been described along with its scheduling parameters, static analysis and some empirical results. We can provide real-time guarantees for all the hard real-time tasks as in non-garbage-collected real-time systems. Furthermore, All the hard real-time tasks follow the dual priority scheduling approach so spare capacity can be reclaimed and the responsiveness of soft real-time tasks is improved.

However, our approach has some limitations as well. First, further investigation is needed for the tight estimation of the parameter cgg_j, which is crucial for the static analysis. Secondly, as with many other reference counting algorithms, our approach has relatively high write barrier overheads. We have not yet tried to optimize out unnecessary barriers. Another limitation we are trying to eliminate is the rigid requirement that soft or non-real-time tasks neither produce any cyclic garbage nor allocate any memory. To date, a multi-heap method is already being developed in our group.

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

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, 2003.

[2] D. F. Bacon, P. Cheng, and V. Rajan. A real-time garbage collector with low overhead and consistent utilization. In *Proceedings of POPL 2003*, pages 285–298, 2003.

[3] A. Borg, A. Wellings, C. Gill, and R. K. Cytron. Real-time memory management: Life and times. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 237–250, 2006.

[4] Y. Chang. Hard real-time hybrid garbage collection with low memory requirement. Technical Report YCS-2006-403, University of York, 2006. http://www.cs.york.ac.uk/ftpdir/reports/YCS-2006-403.pdf.

[5] 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*, pages 900–921, 2002.

[6] R. Davis and A. Wellings. Dual priority scheduling. In *Proceedings of the 16th IEEE Real-Time Systems Symposium*, pages 100–109, 1995.

[7] E. W. Dijkstra. On-the-fly garbage collection: An exercise in cooperation. *CACM*, 21(11):966–975, 1978.

[8] M. Hampton. Using contaminated garbage collection and reference counting garbage collection to provide automatic storage reclamation for real-time systems. Master's thesis, Washington University, May 2003. http://www.seas.wustl.edu/Research/FileDownload.asp?263.

[9] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Lund University, 1998. http://www.cs.lth.se/home/Roger_Henriksson/thesis.pdf.

[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, 2001.

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

[12] M. A. Rivas and M. G. Harbour. MaRTE OS: An Ada kernel for real-time embedded applications. In *Ada-Europe*, pages 305–316, 2001.

[13] 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, 2003.

[14] F. Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In *Proceedings of The 1st International Symposium on Memory Management*, pages 130–137, 1998.

[15] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. In *Compilers, Architectures and Synthesis for Embedded systems(CASES2000)*, pages 9–17, 2000.

[16] F. Siebert. The impact of realtime garbage collection on realtime Java programming. In *Proceedings of ISORC 2004*, pages 33–40, 2004.