Low Memory Overhead Real-Time Garbage Collection for Java

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

ABSTRACT

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 better design of real-time garbage collection algorithms. Use of this indicator motivates an algorithm that combines both reference counting and mark-and-sweep techniques. The implementation of this algorithm for jRate is described and its performance reviewed. The use of dual priority scheduling of the garbage collection tasks allows spare capacity in the system to be reclaimed whilst guaranteeing deadlines.

General Terms

Languages, Algorithms, Performance

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and embedded systems; D.3.3 [**Programming Languages**]: Language Constructs and Features—*Dynamic storage management*; D.4.2 [**Operating Systems**]: Storage Management—*Garbage Collection*

Keywords

real-time hybrid garbage collection, garbage collection granularity, dual-priority scheduling, space overhead

1. INTRODUCTION

The management of resources is a key component of all real-time systems. Java provides 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 presence of an easy-to-use real-time memory model with low overhead is crucial for Java's use in embedded real-time system. The scoped memory model proposed by RTSJ (Real-Time Specification for Java) [4] has relatively low overhead but is criticised for its difficulties of use (e.g. by [1]). An alternative is garbage collectors, most of which now in use are tracing ones. 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 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, 13, 19, 12]. 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[5].

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 indicator motivates the design of a new real-time garbage collector. Instead of performing pure tracing or reference counting, we propose a hybrid approach along with concurrent dual priority scheduling, which can allow the reclamation of spare capacity in the whole system [9].

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 which is implemented on GCJ and jRate. Section 5 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. 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 or even eliminate 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 that needs more resources than one which adopts a more application memory-managed approach.

To make the second argument 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. Therefore, tracing garbage collection is inherently a family of high granularity garbage collection algorithms. Since the reclaiming phase of a GC cycle is usually short and bounded, the example shown above applies to the behaviour of tracing collectors very well.

Recent research on real-time garbage collection algorithms such as [1, 13, 17, 12] 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. Irrespective of the scheduling algorithm, any attempt to give significantly less computation resource to a tracing collector will raise the memory requirement dramatically [2, 13, 17, 12].

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 can only be released at the points 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]. Bear in mind 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,

[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 Latencies

we assume the heap is big enough to hold all the garbage objects and the live ones.

Now, we can define the performance indicator:

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

Notice 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 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, the arguments "*garbage_set_size*" and "*object_size*" can also be extremely large, even comparable with "L_{max}" in extreme situations. Consequently, the free memory producer latencies of reference counting algorithms could be very long in some systems as well.

In order to keep the redundant memory needed small, we suggest that the designer of a real-time garbage collector should try to minimise its free memory producer 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 reference counting algorithms is to change their garbage collection granularity. Siebert and Ritzau [18, 14] 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 to other reference counting algorithms. Ritzau's work [14] is such an algorithm. By reclaiming the same amount of memory immediately before each allocation, there is 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 hard real-time scheduling due to its problematic behaviour during burst allocations and the fact that it cannot reuse any form of spare capacity.

Combining reference counting with a mark-and-sweep collector (either global or local) is a common way to address the problem of cyclic garbage. This has been studied by [3, 7]. However, real-time issues are not considered in their work. By combining a fine grained reference counting collector with a global mark-and-sweep collector, predictability can be achieved. 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 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 (44.21% to 100.00% with the average of 81.52%) [11].

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

In this section, we present our hybrid garbage collection algorithm which is implemented by modifying the GCJ compiler (the GNU compiler for the Java language, version 3.3.3) and the jRate library (version 0.3.6)[8] — a RTSJ-compliant real-time extension to the GCJ compiler.

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

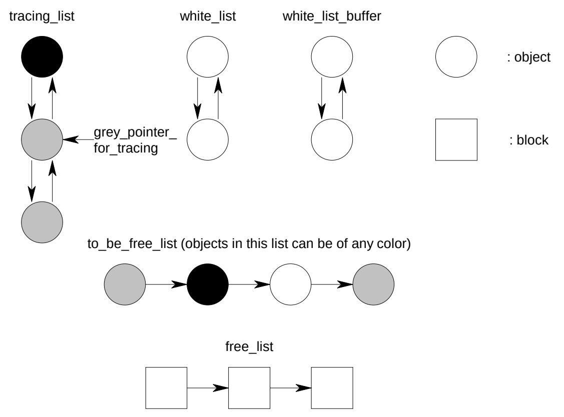

Figure 1: Data Structures

implementation. Which size to choose and how bad the memory access penalty is have been studied by Siebert [18].

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. Its 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). Therefore, we have a 3 words per object and 1 word per block space overhead. However, by allowing programmer to configure the block size, this overhead can be reduced [18].

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.

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" 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 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*. 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. 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.

Additionally, two more kinds of write barrier, *write-barrier-prologue* and *write-barrier-epilogue*, are applied to monitor reference parameter passing and function return respectively. They are, in fact, either the first or second outmost "if" branch of the "write-barrier-for-roots" so the expected cost should be lower.

By using these write barriers, we maintain not only the reference counting algorithm and root information but also the strong tri-colour invariant [10] 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 most acyclic 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.

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

[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

are no grey objects in the "tracing-list". The tracing task then moves 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.

On the other hand, 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 the object count and the 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 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.

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 (please see [6]). 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 with the same period and deadline as that of the tracing task.

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 the objects with overestimated reference counts can be identified as garbage by the tracing task after they die. Moreover, these objects are already considered as part of the cyclic structure when we perform offline analysis (please see [6]).

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 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 [9]. 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.

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 real-time user tasks could be blocked for arbitrary time due to the lack of free memory. Therefore, we consider the 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 [9].
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 6).

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 [9].

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 (GC tasks have adjacent priorities in either bands and the reclaiming task executes at the higher priority before the work limit is reached). 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.

How to determine the periods (deadlines), priorities, worst case computation time and response time of the GC tasks is beyond the scope of this paper (please see [6] for details). If a system is feasible according to our static analysis, the hard real-time tasks are always guaranteed not to be blocked by GC tasks due to the lack of free memory. Furthermore, the period of our GC tasks is mainly determined by the heap size and the rate of cyclic cumulation rather than the rate of allocation. As a result, the period could be much longer than its pure tracing counterpart. Therefore, tracing would be invoked less frequently.

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 does not necessarily mean that the user tasks' memory requirements can always be satisfied because the user tasks could be scheduled in preference to garbage collector although the collector is guaranteed to complete all its necessary work of this period before its deadline. 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 [12]. That is, before the promotion time, if the amount of free memory is lower than a certain value called F_{pre} (how to calculate this parameter is demonstrated in [6]), the priorities of the GC tasks should be promoted. Otherwise, they should be executed at their original priorities until the promotion time.

5. INITIAL PERFORMANCE ANALYSIS

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[15]. Although our implementation is based on jRate, its scoped memory was not used in any tests presented in this paper. That is, we only use heap.

In order to present our experimental results, the notation used in this section is summarized first in table 2.

Notation	Definitions
D	the deadline and also period of both reclaiming and tracing tasks
a_{max}	the worst case memory allocation executed during a GC period
R_{gc}	the worst case response time of the tracing task
T_j	the period of the user task j
C_j	the worst case execution time of the user task j
D_j	the deadline 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

Table 2: Notation definition

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 all kinds of write barriers (see figure 3) and allocations of different object sizes (see figure 4). In order to perform the analysis described in [6], RR, TR and MWR must be obtained from experiments as well (see figure 5). 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[3]. 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.

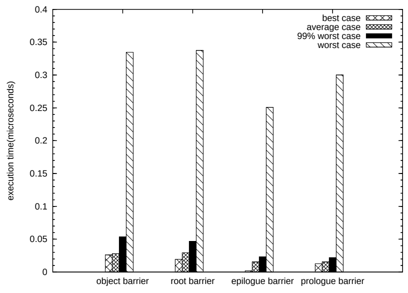

Figure 3: Write Barrier Computation Time

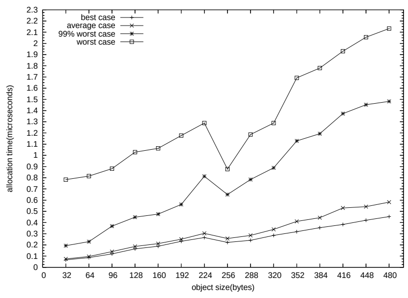

Figure 4: Computation Time of Allocations of Different Object Sizes

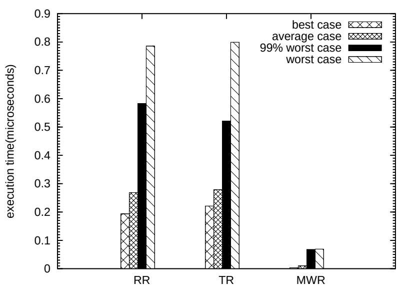

Figure 5: RR, TR and MWR Results

[3]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

As can be observed in figure 3, 4 and 5, there is a large gap between the worst case and average case values. Sometimes, even the 99% worst value can be much lower than the worst value as well. This is due to cache misses and wrong branch predictions which are very common in modern processors. Another thing worth mentioning is the drop of allocation time for object sizes between 256 and 320 bytes. This is because our implementation is optimized to allocate every 8 blocks, which is 256 bytes in total, in a more efficient way.

As defined in table 2, 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 the above information, we can perform analysis (see [6]) 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[4]. 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 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, the maximum amount of live memory L_{max}[5] is calculated first according to [13]. 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, L_{max} cannot exceed 23424 bytes for task set 1 or 36864 bytes for task set 2. By performing the static analysis discussed in [6] with the given heap sizes H, we assign GC tasks with promoted priorities between task 3 and 4 for both task sets

[4]DMPO means deadline monotonic priority ordering
[5]L_{max} includes all the per-object and per-block overheads.

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

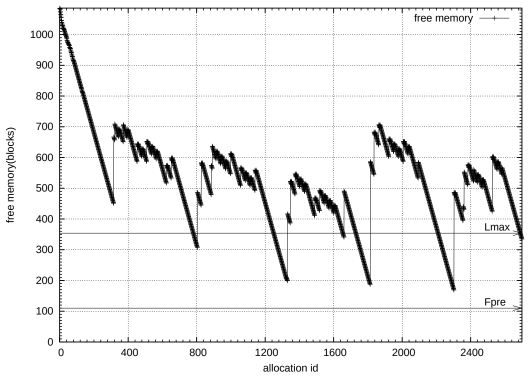

Figure 6: Task Set 1 with Promotion Time 10ms

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 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 due to the lack of free memory. This is theoretically proved based on the information about user tasks' behaviour, heap size, RR, TR and MWR rather than empirical observation [6]. The memory usages of both task sets are presented in figure 6, 7, 8 and 9[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 users tasks should be given preference over GC tasks by squeezing the heap harder.

In order to explore the performance of a non-real-time task under dual priority scheduling scheme, we modify the non-real-time task in task set 1 so that it starts at the same time as all the other tasks and performs certain number of it-

[6]Allocation id x means the xth allocation.

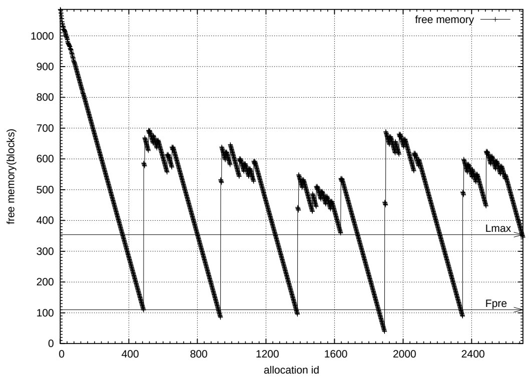

Figure 7: Task Set 1 with Promotion Time 90ms

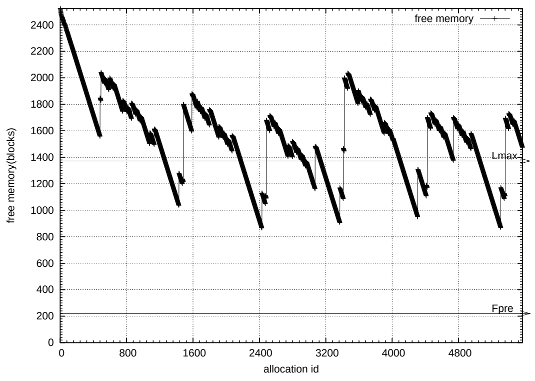

Figure 8: Task Set 2 with Promotion Time 10ms

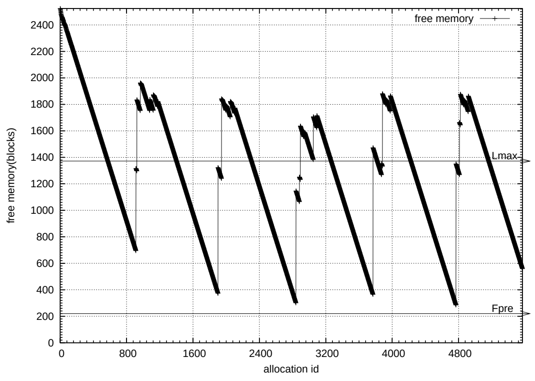

Figure 9: Task Set 2 with Promotion Time 80ms

erations of floating computation. Executions with different numbers of iterations and GC promotion time are performed and corresponding non-real-time task response time are illustrated in figure 10. As one can see, in many cases, dual priority scheduling provides 7-10 milliseconds improvement on non-real-time task response time compared with fixed priority scheduling. Theoretically, the response time of a non-real-time task in a system configured with shorter GC promotion time should never be shorter than that of the same task in a system configured with longer GC promo-

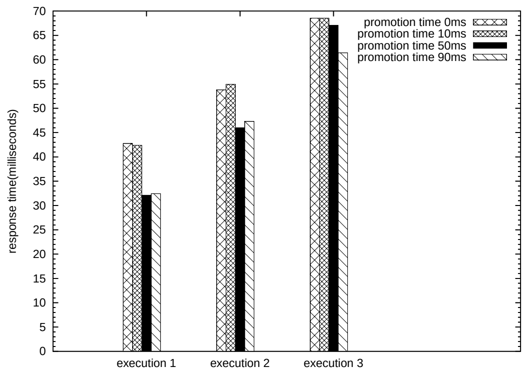

Figure 10: Response Time of Non-real-time Tasks with Different GC Promotion Time

tion time. However, figure 10 shows some exceptions. The reason is that the execution time of and interference to the non-real-time task are different from test to test.

We choose [16] as an example of pure tracing collectors with which we compare our algorithm. 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]. Notice that the user task utilization of a pure tracing system can be lower than ours due to the fact that our write barriers are more expensive than their pure tracing counterparts. However, this cannot change the garbage collection work amount because user tasks are periodically scheduled. 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 [16], 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.

6. 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. Such an approach has been described along with its scheduling parameters 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, our static analysis (see [6]) 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 (see [6]).
- 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.

The limitation of our approach can be summarized as follows:

- 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.
- Further investigation is needed for tight estimation of the parameter cgg_j.
- The current approach we use to protect critical sections is not efficient enough. A pair of such operations can be almost 2 times slower than reclaiming a block. We are now looking for better approach to the protection of critical sections.

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

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

7. 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] D. F. Bacon and V. Rajan. Concurrent cycle collection in reference counted systems. In *Proceedings of the 15th European Conf. on Object-Oriented Programming*, pages 207–235, 2001.

[4] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, 2000.

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

[6] 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.

[7] T. W. Christopher. Reference count garbage collection. *Software-Pratice and Experience*, 14(6):503–507, 1984.

[8] 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.

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

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] F. Siebert. Guaranteeing non-disruptiveness and real-time deadlines in an incremental garbage collector. In *Proceedings of the first international symposium on Memory management*, pages 130–137, 1998.

[18] 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.

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