

# **Improving the Performance of the Deferrable Server Based Garbage Collection Scheduling Strategy**

**Abstract**— Modern programming languages are provided with a mechanism for automatic memory management called garbage collection. A lot of researches had been introduced in this area. On the other hand, some problems existed when applying such languages to the real-time applications. Among these problems, there was the threat of predictability and schedulability of hard real-time tasks. As a solution, different concurrent garbage collection scheduling strategies had been proposed. Unfortunately, these concurrent garbage collectors suffered from redundant usage of system memory. A latter algorithm that minimizes the worst-case system memory requirement with the schedulability of tasks not jeopardized is the deferrable server based garbage collector scheduling strategy. The deferrable server based garbage collection had been proved to surpass all other garbage collection scheduling strategies in minimizing system memory requirements under the worst-case conditions.

In this paper, the performance of the deferrable server based garbage collector is explored under the actual real-time working environment. It also investigates the selection of thresholds to further reduce the memory requirement by using higher capacities of the deferrable server task. The simulation results demonstrate that the deferrable sever based garbage collector achieves better performance under actual real-time environment than other garbage collection algorithms. It also shows that using higher values for the server capacities can lead to better memory usage if a proper value of threshold is chosen.

**Keywords**—Real-time, garbage collector, concurrent, deferrable server, scheduling

## 1. INTRODUCTION

Memory management in real-time and embedded systems is handled using automatic memory management (i.e. Garbage collection or GC for short) which enables the programmers to overcome the potential danger of manual memory management, such as memory leaks, dangling pointers, fragmentation, and so on. The garbage collector distinguishes the memory objects that are no longer in use (*garbage*) from the live objects and reclaims the garbage for future use.

The advent of garbage collection to the real-time scene causes serious obstacles as traditional garbage collection threatens the schedulability and predictability of tasks that demand strict real-time requirements. So, some effort had been put in order to make it suited to real-time systems.

Scheduling incremental garbage collection algorithms was the solution to enhance the position of garbage collection in the real-time scene. The main aim towards scheduling garbage collection is to achieve low overhead and enough predictability for hard real-time tasks. Many works in the literature have classified garbage collection scheduling mechanisms into two categories: sequential and concurrent garbage collection [6]. Sequential garbage collection failed to achieve the aims of scheduling GC in real-time systems. Thus, the main trend is towards concurrent GC techniques.

Concurrent GC is a great step towards truly and efficient real-time garbage collection algorithms. Some variants of concurrent GC have been developed. Among them are the background approach [6], Metronome [7], time-triggered GC and its auto-tuning form [1,14]. Although all of these approaches remove the obstacles caused by traditional garbage collection in real-time systems, they rely on a relatively large amount of redundant system memory. So, some other concurrent GC algorithms had been put on reducing the system memory requirement and guaranteeing the schedulability of hard real-time mutators under automatic memory management.

Among these techniques are the sporadic server (SS) based GC [3] and the deferrable server (DS) based GC [2]. Both of them are based on the resource reserving mechanism. A garbage collector is treated as a

periodic or aperiodic task and is scheduled concurrently with other tasks in the system.

The deferrable server based GC achieves the most minimum worst-case response time of a garbage collector among all other concurrent garbage collectors mentioned before, and so is the worst-case system memory requirement while meeting hard deadlines [2].

The main contribution of this paper can be summarized into two points:

- 1) It tests the performance of the DS based GC technique under the actual run-time environment rather than the worst-case situation. In reality, the GC cycle time is variable and is relative to the amount of live memory. In most cases, the GC pause time is quite smaller than the largest bound. So, instead of assuming the worst-case situation all along, it is assumed that the GC cycle time is directly proportional to the amount of live memory. Hence, a higher CPU utilization is obtained.
- 2) It investigates using higher server capacities for the deferrable server. By selecting suitable values of thresholds, these capacities can achieve better performance on reducing system memory requirements while meeting the strict demands of hard real-time tasks.

## **2. BACKGROUND**

### **2.1 Garbage Collection Algorithms**

Garbage collector is a part of the runtime or operating system that performs the task of finding and recycling the memory occupied by all garbage objects. Many researches had categorized the basic algorithms of traditional garbage collection into two classes: reference counting and tracing algorithms. Reference counting requires an additional reference count (RC) field for each object [4,5]. RC field is updated when a pointer is changed by a mutator. When the RC value reaches zero, the object is reclaimed. This algorithm fails to detect circular linked structures and the overhead of managing counters can be high.

The tracing algorithms are classified into mark-sweep and copying algorithms. The mark-sweep collector traverses the pointers to find live objects and marks them. Then, a collector scans the entire heap and reclaims garbage objects that have not been marked.

For copying algorithms, the heap is divided into two equally-sized spaces called fromspace and tospace. When a garbage collector is triggered, it traverses the pointers and copies the live objects into the new tospace. The tracing algorithms are of stop-the-world fashion. Their pause time is not suitable for hard real-time systems.

Incremental garbage collection algorithms [10, 11, 13] had been proposed to distribute and hide the garbage collection pause time through-out the execution of mutators. Although these approaches reduce the intermediate pause delay, they are not alone suitable for hard real-time systems. They have to cooperate with a scheduling mechanism.

This paper uses the modified incremental copying garbage collection strategy found in [3]. It is an incremental copying garbage collection algorithm which is a variant of Brook's algorithm [11] adopted with some modifications mentioned in [3]. Figure (1) illustrates the heap view of incremental copying algorithms.

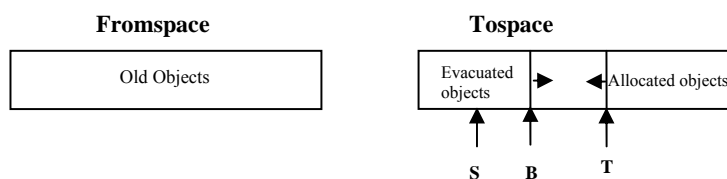


Figure (1): The heap of incremental copying garbage collector  
 B: evacuation pointer, S: scan pointer, T: allocation pointer

## 2.2 Concurrent Garbage Collection Scheduling Strategies

The scheduling of garbage collection is mainly divided into two kinds: sequential garbage collection and concurrent garbage collection. In sequential garbage collector, the GC work is triggered by allocation requests and pointer manipulation operations. It suffers from the accumulative GC delays due to successive memory management operations. Consequently, the schedulability of tasks may be missed.

In concurrent garbage collectors, the GC is treated as a separate thread which runs in parallel with the application threads. It is a combination of incremental GC and a specific real-time scheduling strategy.

Concurrent garbage collectors are designed to achieve predictability and schedulability in hard real-time systems, among them are the

background approach, Metronome, time-triggered GC and its auto-tuning form.

In the background approach, the GC work never interrupts the high priority tasks. It gets running when there is no high priority task needs executing. The low priority processes use the standard scheduling techniques of soft real-time GC algorithms. The system memory requirement is much larger than that of any concurrent GC algorithm.

In time-triggered GC, the collector is assigned a deadline for when it must complete its current cycle. The GC is treated as a periodic activity. An auto-tuning GC scheduling strategy was proposed in order to tune the GC speed according to the requirements of the individual application and to do without the priori schedulability analysis which depends on the worst-case assumptions [14]. It gives portability but suffers from high run-time overhead.

In Metronome, the collector and mutator are interleaved using fixed time quanta. The GC is triggered by a fixed interval of time.

Although all of the above algorithms guarantee predictability and schedulability of hard real-time tasks, they rely on a relatively large amount of redundant system memory. Some other concurrent garbage collection scheduling strategies had been proposed to minimize the system memory requirements with the schedulability of tasks not threatened, among them are the sporadic server based GC and the deferrable server based GC.

In sporadic server based GC, the GC is treated as an aperiodic activity. The sporadic server task (SS) [9] is used to serve the needs of GC. According to rate monotonic scheduling strategy (RMS)[12], the task with the smallest period is given the highest priority. So, the SS task is given a period which is equal to or greater than that for the highest priority periodic task. A single GC cycle may last several periods of the server before it completes, which increases the worst-case response time of GC. This algorithm shows better performance on reducing the system memory requirement than the background approach.

The deferrable server based GC [2] resembles the sporadic server based GC in treating GC as an aperiodic activity. It assumes that the GC is the only aperiodic task in the system (this is only for simplicity). The algorithm utilizes the deferrable server strategy (DS) [8] of scheduling aperiodic tasks in real-time systems. The DS task can service an aperiodic task anytime during its period provided that it still has some unused execution time. It is called bandwidth preserving algorithm. At the end of the DS period, if any portion of the execution

time is not used, it is discarded. The server has the shortest period among all other tasks; consequently, it will be assigned the highest priority according to the rate monotonic scheduling strategy [12]. The capacity of the server is selected through a particular parameter configuration scheme. This scheme is addressed using two different approaches, that is, the utilization based analysis and the exact analysis of the selection of parameters. The exact analysis is better than utilization based analysis because it takes into consideration the individual task set. In this way, DS based GC achieves the worst –case response time for the GC thread comparing with all previous concurrent garbage collectors. So, it achieves the best results in minimizing the system memory requirements with the schedulability of tasks with hard time constraint not jeopardized.

### 3. ACTUAL REAL-TIME GARBAGE COLLECTOR

A brief description of the real-time environment in which the deferrable server based GC works is presented. The task set (TS1) used here is the same as in [2]. It is shown in table 1.

The tasks are scheduled according to rate monotonic scheduling strategy [12]. This scheduling strategy assigns fixed priorities to tasks according to their periods. The task period is equal to its deadline. The task with the shortest period will be given the highest priority in order to hurry the response time of the task before its deadline. For DS,  $T_{ds} = 4\text{ms}$  and  $C_{ds} = 1\text{ms}$ . Therefore, the DS task will have the highest priority among all other tasks.

Table 1: Task set (TS1)

Task	$C_i$	$T_i$	$D_i$	$A_i$	$\alpha_i$
$\tau_1$	2	10	10	488	0.53
$\tau_2$	4	30	30	528	0.46
$\tau_3$	10	50	50	800	0.38
$\tau_4$	15	100	100	1296	0.57

According to this strategy, if a low priority process is executing and a high priority one is invoked, then the low priority process is preempted and has to wait till the high priority process finishes its execution. At this time, it can continue its execution unless there is no any other high priority task ready to execute. In this way, the four tasks of the task set with the DS task are scheduled.

To simplify the analysis, some assumptions are made as in [2].

- A1** There is no aperiodic mutator task.
- A2** There's no blocking factor among the tasks.
- A3** The context switching and scheduling overhead is negligible.
- A4**  $C_i$ ,  $T_i$ ,  $D_i$ ,  $A_i$  and  $\alpha_i$  are known a priori, and the deadline of each periodic task is equal to its period ( $D_i = T_i$ ).
- A5**  $L_{\max}$ ,  $\mathcal{F}_{\max}$ ,  $v_c$  are known a priori.

The nomenclature denoted in this paper is illustrated in table 2.

The DS task is used to provide the GC thread (the only aperiodic task according to **A1**) with immediate serving upon arrival. The invocation of the GC thread is irregular and is triggered by a threshold. The threshold means the amount of used memory (system memory requirements) for when the GC thread is ready to execute and will be served immediately by the deferrable server task.

Table 2

Symbol	Description
$\tau_i$	Periodic mutator task $i$
$C_c, v_c$	Worst-case execution time, Collection speed for garbage collector
$A_i$	Maximum amount of memory allocated by $\tau_i$ during $T_i$
$\alpha_i$	Portion of live memory out of $A_i$
$C_i, T_i, D_i$	Worst-case execution time, period, deadline for $\tau_i$
$R_i, R_c$	Worst-case response time of $\tau_i$ and GC
$L_{\max}$	Maximum amount of live memory
$\mathcal{F}_{\max}$	Maximum amount of the uncollected garbage
$T_{ds}, C_{ds}$	Period and capacity for a deferrable server

The worst-case execution time of the GC thread was estimated in [3] for TS1. It is equal to 3 ms. This means that the GC thread will take 3 periods of the deferrable server periodic task to do its worst-case execution time. The new simulation of the DS based GC also takes 3 ms for  $C_c$ .

To simplify the simulation, other assumptions based on [2] are made.

- A6** The execution time of any mutator task  $\tau_i$  is in the worst case all along, that is, is always equal to  $C_i$ . So is the amount of memory allocated by  $\tau_i$  during  $T_i$ .
- A7** Not until the end of one GC cycle can the memory reclaimed in that cycle be available to mutators.

**A8** For each periodic mutator task  $\tau_i$ , the memory consumption behavior repeats periodically, and the consumed memory during one period becomes 'dead' when that period completes.

**A9** The amount of floating garbage arising during a GC cycle is equal to the amount of garbage generated by mutators during the cycle.

**A10** For each  $\tau_i$ , the amount of memory consumed by it in each time grain is equal to  $A_i/C_i$ .

In [2], **A6** was extended to include the worst-case execution time  $C_c$  for the GC task all along. This means that the GC thread upon invocation has to last for 3 ms all along the 300 ms (the hyperperiod of all periodic mutator tasks). Indeed, basing the simulation on the worst-case assumptions leads to unacceptably low CPU utilization [14] because it takes into consideration all possible worst –case situations for the system. This leads to the longest response times for all tasks and consequently, there is a little time left to utilize the processor if exists.

The new simulation technique tries to reduce to some extent the bad influence of assuming worst-case situation in the old simulation of the deferrable server based GC scheduling strategy [2]. This is done by eliminating the continuous worst-case execution time of the GC thread.

Based on **A5**,  $v_c$  (Collection speed for GC task),  $L_{max}$  (Maximum amount of live memory) are known a priori. These variables were not given explicitly in [2]. For Collection speed, it is assumed that the GC thread evacuates memory in a uniform fashion. This means that the amount of evacuated memory till a certain instant is equal to the maximum amount of live memory ( $L_{max}$ ) multiplied by the time of GC thread till this instant and divided by the worst-case execution time of a GC task ( $C_c$ ). Thus, the amount of live memory increases regularly and with a fixed ratio. It is assumed that the GC Performs flipping and initialization of tospace in the 1<sup>st</sup> time grain of every GC cycle invocation. Hence, the other two time grains are used to make the uniform evacuation.

According to [3], it is assumed that the upper bound of live memory is reached by assuming all tasks have one active instance. Then, the upper bound of live memory  $L_{max}$  can be obtained by the following equation:

$$L_{max} = \sum_{i=1}^n \alpha_i A_i \quad (1), \quad \text{where}$$

$n$  is the number of tasks.

During the simulation of previous work [2], it is observed that the amount of live memory varies from a cycle to another and it decreases greatly from the worst-case situation ( $L_{max}$ ) in most cases. In reality, the GC cycle time should be variable not constant. So, assuming worst-case execution time of the GC thread in each cycle is not correct. Only, this can be done to calculate the maximum deferrable server capacity that achieves the schedulability condition for all tasks from the exact analysis. It should not be used when simulating the running tasks.

In copying algorithms, it is known that the GC cycle time is proportional to the amount of live memory [6]. Also, the worst-case live memory is the dominant factor to determine the worst-case garbage collection time. An assumption **A11** is put to adapt with the previous two facts.

**A11** The GC cycle time is directly proportional to the amount of live memory and can be calculated by  $L C_{\zeta \text{ rest}} / L_{max}$ .  $L$  denotes the amount of live memory in the current GC cycle. Only the worst-case situation occurs at the upper bound of live memory  $L_{max}$ . In this assumption,  $C_{\zeta \text{ rest}}$  is substituted by 2 ms.

The new assumption aids to simulate the performance of GC in the real-time system and expresses two facts, the GC is variable and is relative to the amount of live memory. This gains a higher CPU utilization which is very important for real-time systems

Instances of the old simulation versus the new one are illustrated in figures (2-a), (2-b) respectively at no threshold.

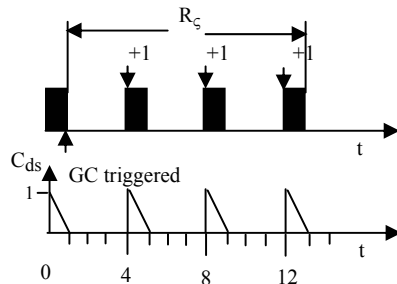


Figure (2-a) : The execution time of GC is equal to 3 ms all along at the parameters  $T_{ds} = 4$ ,  $C_{ds}=1$ ,  $C_{\zeta}=3$ .

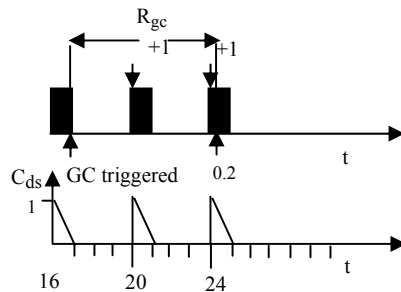


Figure (2-b): The execution time of GC is equal to 1.2 ms according to the amount of live memory evacuated in this cycle at the parameters  $T_{ds} = 4$ ,  $C_{ds}=1$ ,  $C_{\zeta}=3$ .

Figure (2-a) depicts the simulation of the running tasks with the worst-case execution time of the GC thread (**A6**) given in [2]. This sample includes the only GC invocation situation that leads to the worst-case response time of the GC thread (the longest response time). The GC cycle here is in its worst-case situation regardless of the amount of live memory evacuated in this cycle. For 300 ms, it works for 3 ms all along. This leads to increase the response time of the GC thread and consequently the system memory requirements.

In figure (2-b),  $R_{gc}$  denotes the response time of the GC in the current cycle. The figure shows that the execution time of GC is equal to 1.2 ms, instead of 3 ms. Hence, the execution time of GC decreases by 1.8 from the worst case situation due to the new assumption **A11** (the execution time of GC thread is directly proportional to the amount of live memory evacuated in this GC cycle). This leads to decrease the response time of the GC thread and consequently the system memory requirements. The values shown in the figure are obtained from the simulation. This can be applied at different thresholds of the deferrable based GC scheduling strategy.

The simulation results part illustrates that the new simulation results are better than the old one with keeping that the DS task requires less memory is still maintained.

#### 4. EFFECT OF USING HIGH SERVER CAPACITIES

In previous work [2], Based on the exact analysis of the parameter configuration scheme for the deferrable server based GC scheduling strategy, for a given  $T_{ds}$ , the maximum  $C_{ds}$  that satisfies the schedulability condition for all tasks from equation (8) was selected. This results in a value for the budget ( $C_{ds}$ ) = 1 ms or server utilization factor ( $U_{ds}$ ) = 0.25 for TS1 at the selected  $T_{ds}$  (4). All other capacities that are either less than the selected value that achieve the schedulability condition for all tasks or higher than the selected value that do not achieve the schedulability condition for some tasks were discarded.

This section investigates the effect of using higher server capacities than the one derived from the exact analysis. Although these capacities do not achieve the schedulability condition for some tasks from the exact analysis at no threshold, the entry of threshold to the DS based GC scheduling strategy makes it feasible to use such capacities without

any fear. Better selection of threshold with these capacities leads to a schedulable task set ( all tasks meet their deadlines).

The thought towards the selection of higher capacities is to reduce response time of the GC task and consequently the system memory requirements .

The selected server capacity ( $C_{ds}$ ) is 2 ms,  $T_{ds} = 4$  ms and  $C_c = 3$  ms. The suitable selected thresholds are 2000,3000 while threshold values 0,1000 lead some tasks to miss their deadlines. Hence, the selection of threshold is exploited to further reduce the system memory requirements. The influence of thresholds can hardly be figured out by static off-line analysis. It depends on the run-time behaviour of mutator tasks and GC very much, and it can be tested by the simulation of the running tasks.

In this section, the execution time of the GC thread is always in its worst-case situation (3 ms) and all assumptions in [2] are used for analysis and simulation.

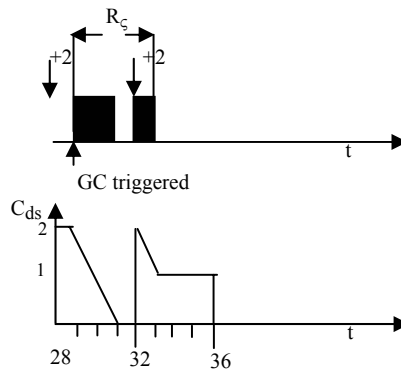


Figure (3):  $T_{ds}=4$ ,  $C_{ds}=2$ , threshold=2000

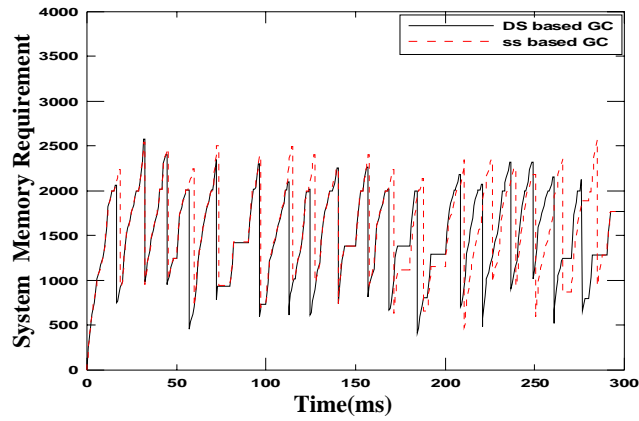
Figure (3) shows that using  $C_{ds} = 2$  leads to reduce the response-time of the GC thread and consequently the system memory requirements. This takes two server periods only to execute the GC thread. While using  $C_{ds} = 1$  leads to a longer GC response time because it takes three server periods to execute the GC thread.

## 5. SIMULATION RESULTS

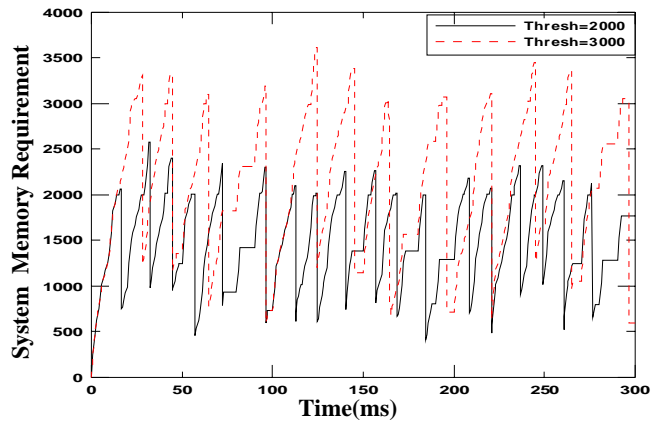
### 5.1 Simulating actual real-time garbage collector

This subsection consists of two parts:

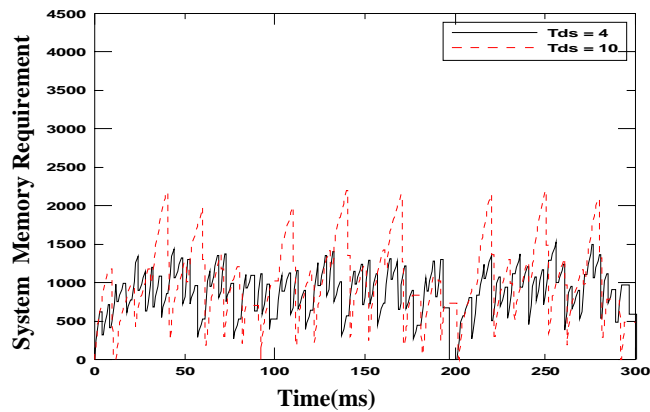
5.1.1 The contrast of the DS based GC scheduling strategy with the new simulation approach and some other GC strategies with the same simulation approach.



Figure(4): System memory requirement of DS based GC and SS based GC with the new simulation approach



Figure(5): System memory requirement of DS based GC at different thresholds with the new simulation approach



Figure(6): System memory requirement of DS based GC at different server periods with the new simulation approach

Figure (4) illustrates the contrast of the DS based GC algorithm and the SS based GC at threshold 2000 with the new simulation approach. From this figure, it can be seen that the algorithm with the new simulation approach outperforms the SS based GC under the same conditions on reducing the system memory requirement.

Figure (5) compares the DS based GC with the new simulation approach at thresholds 2000, 3000. Different selection of thresholds brings different results even if the concurrent GC algorithm is the same.

Figure (6) compares the DS based GC with the new simulation approach at different server periods.

5.1.2. The contrast of the new simulation results and the simulation results of previous work [2].

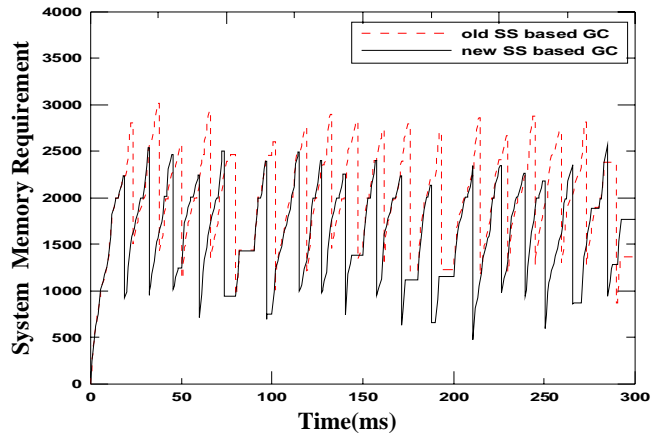
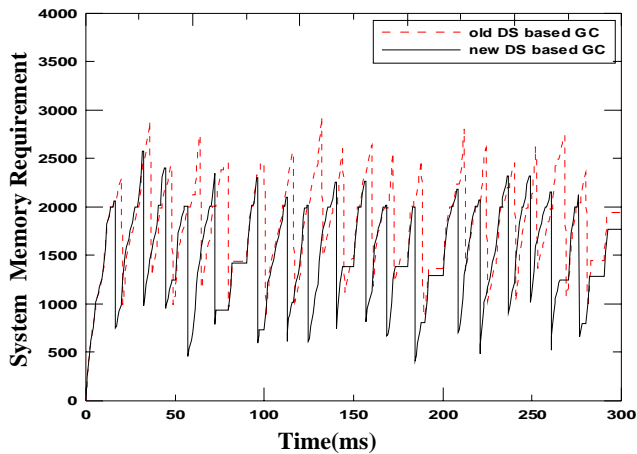


Figure (7): System memory requirement of SS based GC with the old simulation approach and new simulation approach at threshold 2000



Figure( 8): System memory requirement of DS based GC with the old simulation approach and new simulation approach at threshold 2000

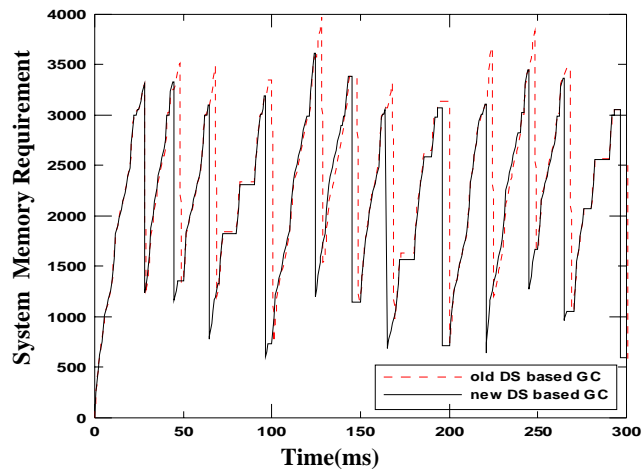


Figure (9): System memory requirement of DS based GC with the old simulation approach and new simulation approach at threshold 3000

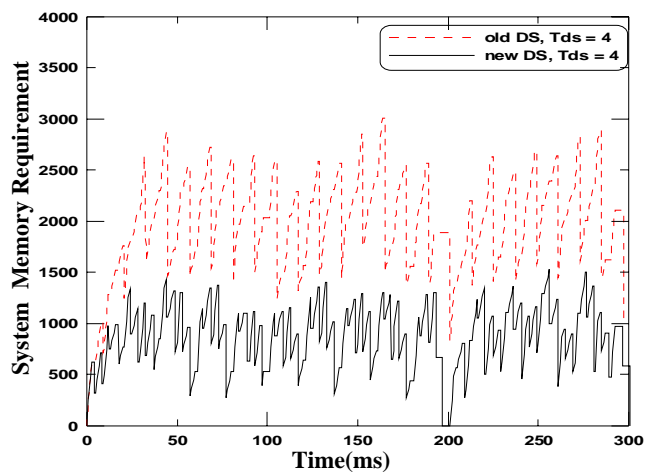


Figure (10): System memory requirement of DS based GC with the old simulation approach and new simulation approach at no threshold &  $T_{ds} = 4$

From figures(7-10), it is shown that the deferrable server based GC with the new simulation approach requires less memory than with the old simulation. It expresses the variability of the garbage collection cycle time and this is done in the actual run time environment. The great benefit from this new approach is to gain a higher CPU utilization.

The algorithm with the new simulation approach also acts as time-based GC scheduling strategy at no threshold as before. This means that the collector and mutator are interleaved using fixed time quanta. Thus, the time-based GC can achieve better performance on reducing the

system memory requirement than the original time-based GC scheduling strategy found in [7].

## 5.2 Simulating high capacities of the DS based GC

This subsection shows the effect of using higher server capacities than the one derived from the exact analysis of the deferrable server based GC scheduling strategy. The selected capacity of the server is 2 ms and it can be applied with certain values of threshold (e.g. 2000, 3000).

The results show that, the higher the capacity of DS, the less system memory requirement is obtained. This means that the new value of  $C_{ds}$  gives better performance in terms of memory requirement. However, we have to put in mind that the new server budget does not achieve the schedulability condition for some tasks (in the case of valueless threshold). However, better selection of thresholds makes the task set schedulable. Moreover, more minimizing to system memory requirements can be attained.

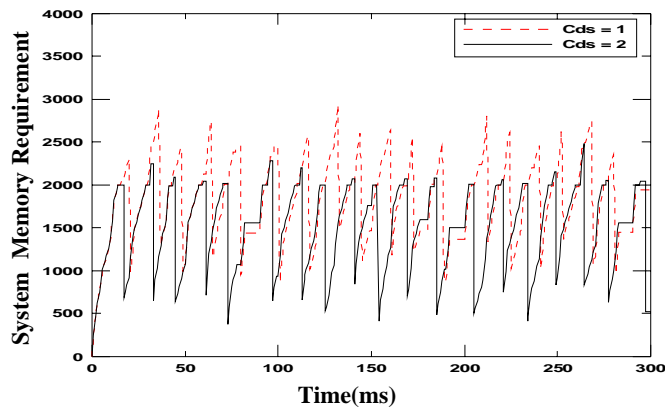


Figure (11): System memory requirement of TS1 with DS based GC at different capacities with threshold 2000

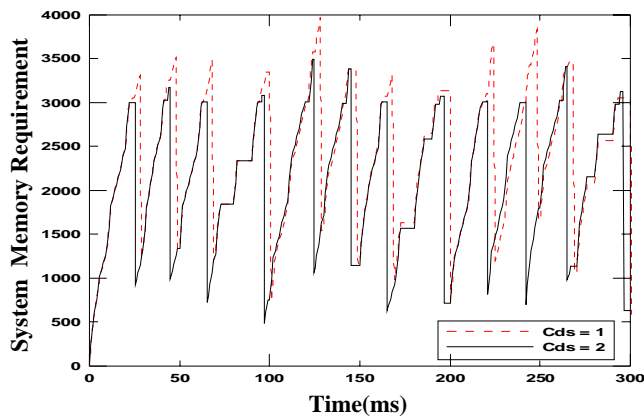


Figure (12): System memory requirement of TS1 with DS based GC at different capacities with threshold 3000

## 5. CONCLUSIONS

In the actual run time system, the garbage collection cycle time is variable and is relative to the amount of live memory. This paper tests the performance of the deferrable server based garbage collection scheduling algorithm under such situation instead of using the worst-case situation all along. It has been proved that the deferrable server surpasses all other garbage collection scheduling strategies in minimizing system memory requirement under the actual real-time environment. This research also exploits the selection of thresholds to further reduce the system memory requirements. It is deduced that better selection of thresholds with higher budgets of the DS based GC than the one derived from the exact analysis leads to a schedulable task set and further better performance on reducing the system memory requirement. This is important for embedded real-time systems without regarding large memory.

## 6. REFERENCES

- [1] S.G.Robertz and R. Henriksson, "Time-Triggered Garbage Collection," *In proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pp.93-102, June 2003.
- [2] Y. Xian and G. Xiong, "Minimizing Memory Requirement Of Real-Time Systems With Concurrent Garbage Collector," *ACM SIGPLAN Notices*, Vol.40 (3), Mar 2005.
- [3] T. Kim, N. Chang, N. Kim, and H. Shin, "Scheduling garbage collector for embedded real-time systems," *ACM SIGPLAN Notices* 34, 7, pp. 55-64, July 1999.
- [4] L.P. Deutsch and D.G Bobrow, "An efficient incremental automatic garbage collector," *Communications of the ACM* 19, 9, pp. 522-526, Sept. 1976.
- [5] P.R. Wilson, "Uniprocessor garbage collection techniques," *Tech.rep., University of Texas*, Jan 1994.
- [6] R. Henriksson "Scheduling Garbage Collection in Embedded Systems," *PhD thesis, Lund university*, July 1998.
- [7] D.F. Bacon, , P. Cheng., and V.T. Rajan, "A real-time garbage collector with low overhead and consistent utilization ," *In The 30th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* , pp. 285-298, January 2003.
- [8] J. K. Strosnider, J. P. Lehoczky, and L. Sha, "The deferrable server algorithm for enhanced aperiodic responsiveness in hard real-time environments," *IEEE Transactions on Computers* 44, 1, pp.73-91, January 1995.
- [9] B. Sprunt, L. Sha, and J. P. Lehoczky, "Aperiodic task scheduling for hard real-time systems," *Real-Time Systems* (1989), pp. 27-60.
- [10] H. G. Baker, "List processing in real time on a serial computer," *Communications of the ACM* 21, 4 *compiler, and tool for embedded systems*, pp.280-294, Apr.1978.

- [11] R. A. Brooks , “Trading data space for reduced time and code space in real-time collection on stock hardware,” In *Conference Record of the 1984 ACM Symposium on LISP and Functional programming*, pp. 256– 262, 1984.
- [12] C.L. Liu and Layland, “Scheduling algorithms for multiprogramming in a hard real-time environment,” *Journal of the ACM* 20,1, pp. 46–61, January 1973.
- [13] T. Yuasa, “Real-time garbage collection on general-purpose machines,” *Journal of Software and Systems* 11, 3, pp. 181-198, 1990.
- [14] S.G. Robertz, “Flexible automatic memory management for real-time and embedded systems,” *Licenciate thesis, Department of Computer Science ,Lund Institute of Technology, Lund University*, pp. 1–93, 2003.