# Applying Priorities to Memory Allocation*

Sven Robertz
Department of Computer Science, Lund University
sven@cs.lth.se

## ABSTRACT

In embedded systems, memory is a scarce resource and great attention must be given to memory management. A novel approach of applying priorities to memory allocation is presented and it is shown how this can be used to enhance the robustness of real-time applications. Focus is on systems with automatic memory management, but the ideas are also applicable to manually managed memory. In systems with automatic memory management, the proposed mechanisms can also be used to increase performance by limiting the amount of garbage collection work. Furthermore, a way of introducing priorities for memory allocation in a Java system without making any changes to the syntax of the Java language is proposed. This has been implemented in an experimental Java virtual machine.

**This is a shorter version of a paper to appear at the International Symposium on Memory Management (ISMM), Berlin, Germany, June 20-21, 2002.*

## 1. INTRODUCTION

With the recent development in small, cheap and fast processors for embedded systems and the emerging trend of writing embedded applications in high level object oriented languages, the performance limiting bottleneck may no longer be CPU time but rather memory and memory management. This is accentuated by the high relative cost of memory in embedded systems and systems on chip.

Memory management is a system-global problem and currently puts a great responsibility on programmers. For instance, a memory leak or excessive memory allocation in one module of a system will eventually cause the entire system to run out of memory and fail. Therefore it is interesting to study whether it is possible to apply priorities to memory as well as CPU time allocation; just as we don't want an important process to be delayed because a less important one is executing we don't want an unimportant memory allocation to cause a critical process to fail or be delayed, because the system runs out of memory or has to do a large amount of garbage collection work to satisfy its allocation needs.

We propose a novel approach which addresses two problems: firstly, how to increase program robustness by avoiding out-of-memory problems and secondly, to increase application performance in systems with automatic memory management by reducing the garbage collection (GC) workload. Section 3 briefly describes both aspects, whereas the rest of the paper will focus on the robustness issue.

While this paper focuses on object oriented systems with garbage collection, especially Java, the robustness issues should be equally applicable to any memory allocator.

A note on terminology; in order to avoid confusion we will use the terms *high priority* (HP) and *low priority* (LP) to denote the CPU time priority of a process and the terms *critical* and *non-critical* for our new notion of priorities for memory allocations.

## 2. BACKGROUND

It has been shown that it is possible to schedule GC work in such a way that high priority processes are not disturbed by using a technique called semi-concurrent garbage collection scheduling [3]. The fundamental idea of this technique is that since we don't want the high priority processes to be delayed by garbage collection, we suspend the garbage collector when they are executing. The GC work neglected during the execution of the high priority processes is then performed in the pauses between the activations of high priority processes. The remaining CPU time will be divided between executing low priority processes and performing GC work motivated by the actions of the LP processes, using traditional incremental techniques [5].

Basically, a system using this strategy can be described as having three levels of priority:

1. High priority processes
2. Garbage collection required to satisfy the high priority process
3. Low priority processes and traditional incremental garbage collection

Figure 1 shows how the CPU time will be used in a system

with one periodic high priority process and one low priority process.

Coupled with good worst case execution time and memory requirements estimates and a good garbage collection work metric, semi-concurrent garbage collection scheduling allows us to make hard real-time guarantees for the high-priority threads by using traditional schedulability analysis.

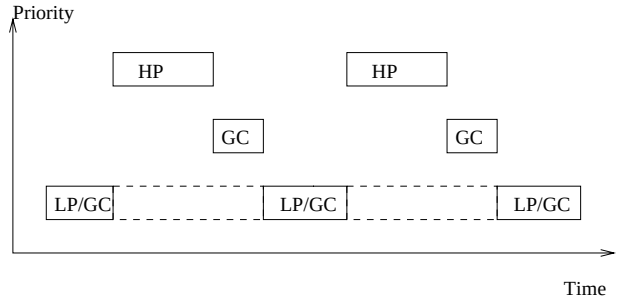


**Figure 1: Dividing the CPU time between processes. The system consists of one periodic high priority process (*HP*) and one low priority process (*LP*). Whenever a high priority process is suspended, and no other HP process is eligible for execution, the garbage collector (*GC*) is run. GC work is also interleaved with the low priority process using traditional incremental garbage collection.**

# 3. APPLYING PRIORITIES TO MEMORY ALLOCATIONS

We like to view memory allocation as any other resource allocation. Our goal is to provide run-time system support for doing the most important memory allocation if the system has limited memory in analogy with how the process scheduler makes sure that the most important process is run and less important ones are delayed if CPU time is scarce.

## 3.1 Avoiding out-of-memory situations

A high priority process in an embedded system may perform other tasks[1] in addition to its core functionality. For example, a digital controller process may produce log data in addition to calculating and outputting its control signal. In such a process, memory allocations by the less important tasks (e.g., producing log data) must never interfere with the core functionality (calculating the control signal).

This can, of course, be achieved by manually ensuring that the amount of log data never exceeds a certain value, e.g., by using a bounded buffer for delivering it to the logger process. Doing this manually has the drawback that the size of the buffer has to be calculated and this calculation is highly platform and application dependent. (I.e., each time a change that affects the application's memory allocation behaviour is made, the maximum amount of non-critical memory has to be recalculated.) If more than one process does unrelated non-critical memory allocations, the complexity of managing this increases rapidly. Thus, manual solutions require a lot of work and risk beeing unnecessarily conservative, error prone, or both.

Our approach to this problem is to transfer the responsibility for making the decisions about when to allow non-critical memory allocations from the programmer to the run time system. Then, the only a priori calculation that has to be done is to calculate the amount of critical allocations done by each (high priority) process during its period and this depends only on the application and not on any properties of the target platform.

This approach can also be used to provide a "limp home" mode, i.e., a mode of operation with lesser performance but radically lower memory consumption that will allow the application to continue executing in an out of memory situation, facilitating a more graceful degradation. This may be useful for adding some amount of predictability to applications with non-predictable memory allocation behaviour.

Finally, non-critical memory allocation gives programmers the possibility to add more features to a system without risking that these additions cause the system to run out of memory and jepoardise the core functionality of the system even if it is moved to a smaller platform. E.g., a low priority process with only non-critical memory allocations cannot cause a system to fail since, if the CPU load is dangerously high it will not get any CPU time and if the amount of memory is too low, it will not be allowed to allocate any memory.

This also has the advantage that it makes it easier to make hard real-time guarantees since worst case and schedulability analysis only has to be done on the critical parts of the system. Such analysis still has to be done using existing techniques [6, 10, 7].

## 3.2 Improving performance by reducing the GC workload

Another reason to limit non-critical memory allocations is to reduce the amount of garbage collection work needed and thereby increaseing the amount of CPU time available to the application. This can, in turn, improve the application's performance by e.g., allowing more accurate calculations or a higher sampling rate.

Furthermore, in a real-time GC system, such as the one devised by Henriksson [3], additional memory allocations done by a high priority process may cause starvation of low priority processes; either directly, through increased execution time, or indirectly, due to the increase in GC work caused by these allocations (since the garbage collector for the high priority processes run at a higher priority than the system's low priority processes). In complex systems, however, the LP process may be more important for good system performance than a secondary task of the high priority process. By using priorities for memory allocations, the application may be written so that, if the system runs low on memory, the primary tasks of both the HP and the LP processes are executed, but the less important task of the HP process is not.

[1]The word task is used in the sense "a piece of work to be done" and not in the real-time programming sense. For the latter, the words process and thread are used.

# 4. NON-CRITICAL MEMORY ALLOCATIONS

The semi-concurrent garbage collection scheduling model introduces a special garbage collection scheduling for the high priority processes in order to guarantee that they are never delayed. In this work we take this a step further by also considering the behaviour of the memory allocator and the risk of running out of memory, due to, for instance, unpredictable application behaviour or even wrong worst case estimates. This is done by introducing the notion of non-critical memory allocation requests, i.e., requests for memory that the run-time system may choose to deny without causing the program to fail.

Ultimately, what we want to do is to keep the amount of live non-critically allocated memory below a certain limit in order to make guarantees that critical allocations never will fail. Unfortunately, live memory amount is not a very suitable measurement, since keeping track of this is not always practically possible [2]. In automatically managed memory systems, where we have the problem with floating garbage[3], there is no real way of knowing how much live memory there is in the system. The only factor we can be sure of is the amount of memory available for allocation, so we need to base our decisions on this.

## 4.1 Non-critical allocation limit

The decision whether to grant or deny a non-critical memory allocation request has to be as simple as possible if it is to be used in high performance applications. We do this by introducing an allocation limit for non-critical allocations; if there is less free, or *allocatable*[4], memory than this limit, no non-critical allocations may be done. This limit will vary over time; at the start of a GC cycle, we have to reserve memory for all the HP memory allocations needed during this GC cycle and then, as the HP process runs and does its allocations, the amount of reserved memory is reduced accordingly. Figure 2 shows schematically how the amount of allocated, reserved and free memory varies over a GC cycle.

When deciding whether to grant or deny a non-critical memory request, we look at how much allocatable memory there is, and how much memory we need to reserve for the HP process so that all its remaining memory allocations during this GC cycle will succeed. Let $n$ be the number of HP periods in a GC cycle, and $m_{HP}$ the amount of memory allocated during each period by the HP process. Then, $i$ HP periods into a GC cycle we need to reserve $R_{HP_i} = (n - i)\ m_{HP}$ bytes for the remaining HP periods during this GC cycle.

[2]In systems with manual memory management it would be trivial to keep track of the amount of live non-critical memory, since objects are explicitly deallocated. The only problem here is the possibility of fragmentation.

[3]Floating garbage is memory that is no longer reachable from the application but has not yet been reclaimed by the garbage collector.

[4]Allocatable memory is memory that is immediately available for allocation. We prefer the term allocatable memory to free memory since, depending on the memory allocator or garbage collection algorithm used, the term free memory may be difficult to define or even irrelevant. E.g., in a non-compacting system, the amount of free memory may be much larger than the amount of allocatable memory due to fragmentation.

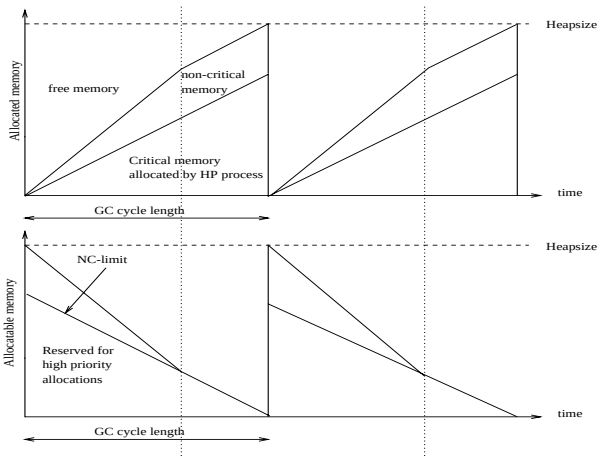


**Figure 2: A schematic figure showing the limit for non-critical allocations. The dotted lines indicate the times where the non-critical limit is equal to the amount of allocatable memory, i.e., when the system starts to deny non-critical allocation requests.**

Non-critical memory allocations should only be allowed if they won't cause the amount of allocatable memory to drop below $R_{HP}$.

## 4.2 Fixed GC cycle length

In order to be able to guarantee that the HP process always will get the memory it requests, we need to make sure that the GC always keeps up with the application. I.e., after each invocation of the HP process, the GC must do enough GC work so that all the allocations during the next HP process invocation will succeed. Given the amount of memory allocated by the HP process each period and the amount of memory reserved for HP allocations we can calculate the GC cycle time expressed in number of HP process periods. We call this time the nominal GC cycle time.

To ensure that no HP allocation fails, we need to complete each GC cycle within this time, even if the actual amount of allocations done during the current GC cycle are less than the worst case. Otherwise, the situation may arise that there is allocatable memory left, but not enough for another complete HP process invocation. If a HP process is started at that time, it will require more memory than currently available and thus, that HP process will be delayed by panic garbage collection.

# 5. NON-CRITICAL MEMORY IN JAVA

The main objective when implementing these ideas in a Java environment was that no changes to the syntax of the Java language should be made, and that programs written for our system should work on any Java platform (but, of course, without the added sematics of non-critical memory allocations.)

Our proposed approach is to use the exception mechanism of Java, so we define a special exception class, `NoNonCriticalMemoryException`, with the added semantics that all allocations that are done in a block which catches

that exception are non-critical. Figure 3 shows a simple program which does both critical and non-critical memory allocations. This program will run on any Java platform with the only addition of an (empty) exception class.

```
void example(){
        Object aCriticalObject = new Object();
        foo(aCriticalObject); // do something important
        try{
                Object aNonCriticalObject = new Object();
                foo(aNonCriticalObject);
                doSomething();
                        // do something
                        // if the non-critical
                        // allocation was successful
        } catch(NoNonCriticalMemoryException e){
                        // non-critical allocation failed
        }
}
```

**Figure 3: Small example program. The allocation of `aCriticalObject` is always done, but the allocation of `aNonCriticalObject` may be denied. If the allocation fails, a `NoNonCriticalMemoryException` is thrown and may be handled in the catch-clause.**

Non-criticality is transitive, i.e., memory allocations done in a method that is called from a non-critical region, like the calls `foo(aNonCriticalObject)` and `doSomething()` in Figure 3, are also non-critical. Note, however, that the first call to `foo()`, `foo(aCriticalObject)` is *not* non-critical since the call is not made from a non-critical block. This behaviour is preferable since an auxilliary function could be called both from critical and non-critical regions of the same program.

The exception class `NoNonCriticalMemoryException` is an unchecked exception in order to make such transitivity possible without having to litter the code with `try` and `catch` clauses. An instance of this class can be statically allocated to avoid wasting memory.

We have made an experimental implementation using the IVM (Infinitesimal Virtual Machine) [4], a very compact real-time Java virtual machine currently being developed at the Department of Computer Science, Lund University. Currently, we explicitly turn non-critical allocations on and off using a native method `IVM.setMemoryPriority()`. This is not fundamentally different from our proposed approach since the `setMemoryPriority()` calls could be inserted automatically by the class loader as the exception catching table is set up (much in the same way as `monitorenter` and `monitorexit` are generated for synchronized blocks). This is, we believe, a better approach than dynamically checking whether `NoNonCriticalMemoryException` is caught at each allocation, since such a run-time check would be more expensive.

# 6. EXPERIMENTAL RESULTS

We have implemented these ideas in a simple control system. For the experiments, we used a lab process with a ball on a beam. The angle of the beam is controlled in order to roll the ball to a given position on the beam, see Figure 4.

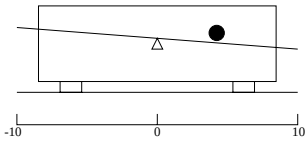


**Figure 4: The ball-on-beam process. The beam can be rotated to roll the ball to the desired position. The position of the ball is in the interval $[-10, 10]$.**

The control was done by a Java application consisting of three threads; a user interface (low priority), a reference generator (high priority) and a controller (high priority). In addition to doing the actual control, the controller thread sends log data back to the user interface thread.

## 6.1 Avoiding out-of-memory situations

We have encountered two scenarios where non-critical memory allocations can help making sure that a change to a previously working system doesn't risk breaking it: increasing the sampling rate of the controller and reducing the amount of memory available to the application.

When the sampling rate is increased, the controller both uses a greater part of the CPU time and allocates log data at a higher rate until we get to a point where the user interface thread doesn't get the CPU time needed for consuming all the log data and the application runs out of memory and fails. By making the log data allocations non-critical, this cannot happen and the control is not affected.

Reducing the available memory[5] will, obviously, at some point cause the application to fail. However, by making the allocation of log data non-critical, the minimum memory requirement for the application may be significanty reduced compared to the original version.

The following traces illustrate the first scenario. In these experiments, the period of the reference generator and the controller was both 20 ms, and a log data object about 60 bytes. Figure 5 shows a run of the ball-on-beam system without non-critical memory. The high allocation rate causes a large GC workload and the UI process is starved, eventually leading to failure. Figure 6 shows the same system where the allocation of log data has been made non-critical. The majority of the log data allocations are still made, but the allocation is kept at a sustainable level. Figure 7 shows a close-up of Figure 6 where you can see the non-critical behaviour more clearly.

## 6.2 Improving performance

Our experiments also indicate that it is possible to achieve better control performance by limiting the amount of non-critical memory allocations. The plots in Figure 8 show two runs of the ball-on-beam application without and with non-critical memory allocations enabled, respectively.

In the version without non-critical allocations, the high al-

[5]This could occur either by actually running the system on a smaller platform or, perhaps more likely, by adding more threads to the system.

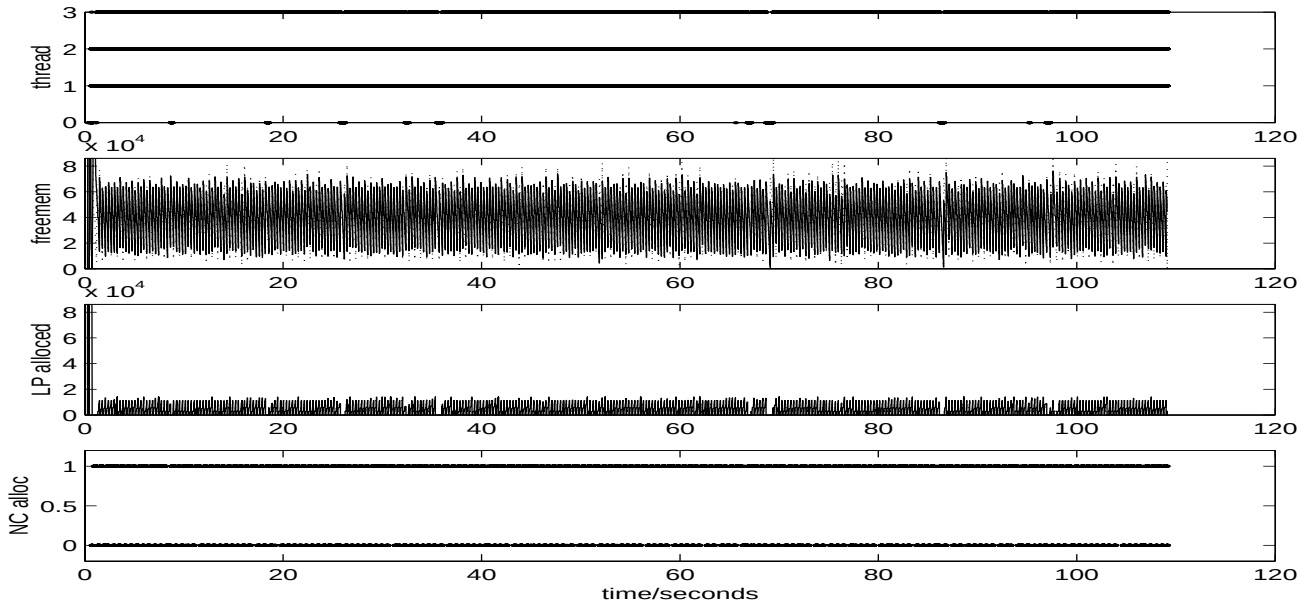


**Figure 6: A run of the ball-on-beam system with log-data allocations made non-critical. In the thread plot you see that the UI thread gets CPU time throughout the run. The third plot shows the amount of memory allocated by low priority processes during this cycle. The fourth plot shows if non-critical allocations succeed or not; high level means success and low level is deny.**

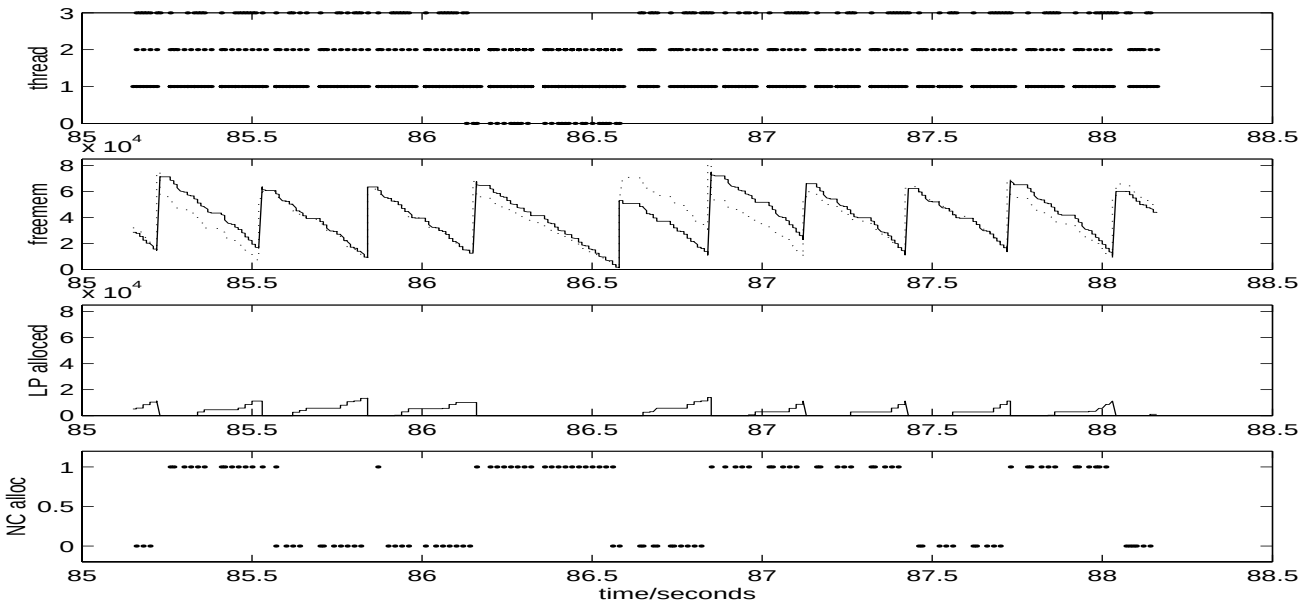


**Figure 7: Closeup to show the non-critical memory behaviour.The dotted line in the freemem plot is the non-critical limit.**

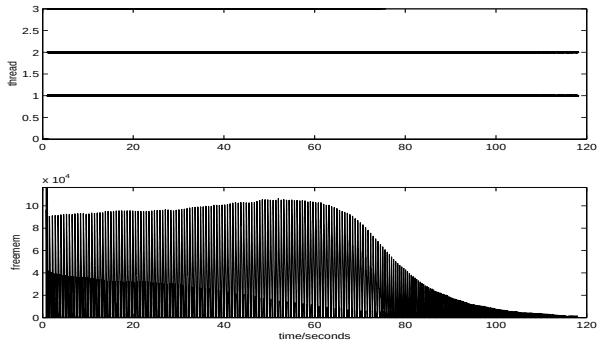


**Figure 5: A sample run of the ball-on-beam system without non-critical memory. The UI thread *(3)* doesn't get enough CPU time to consume all plot data that is produced. After $t = 75$ it is totally starved by the GC. Then, less and less memory is available and more and more CPU time is spent doing (panic) GC. In the first half of the run the controller *(1)* and reference generator *(2)* threads run unimpeded, and the control was OK until $t = 90$. After that the amount of panic stop-the-world GC caused so long delays that the controller dropped the ball. The CPU load is almost 100% and the idle thread *(0)* is not run except in the very beginning. The reason that the maximum amount of allocatable memory increases in the middle is that when the GC cycles get shorter there is less floating garbage.**

location rate occasionally forces the garbage collector to do a full garbage collection cycle in order to reclaim enough memory to satisfy the allocation needs. This delays the high priority controller process so that it misses its deadline which, in turn, degrades the control performance.

When the allocation of log data is made non-critical, the allocation is kept below the safe limit and the system runs as designed, with more consistent control performance.

# 7. RELATED WORK

## 7.1 Memory Management in Real-Time Java

There are two specifications for real-time Java; The Real-Time Specification for Java (RTSJ) [2] and the Real-Time Core Extensions (RTCE) [1]. They both try to solve the real-time garbage problem by avoiding it, using region based approaches to memory management for the real-time threads. The non-real-time threads do their memory allocation on a heap with traditional garbage collection.

RTSJ uses *scoped memory areas* for the high priority threads. Objects allocated in scoped memory areas are not garbage collected but instead the whole memory area is reclaimed when the program exits the scope in which the memory area was allocated.

The access restrictions associated with scoped memory (e.g., objects allocated on the heap may not reference objects in scoped memory, and real time threads aren't allowed to access the heap[6]) make inter-thread communication more difficult. Real-time threads, however, may share scoped memory areas.

[6]Since the heap is garbage collected, real-time threads with hard time constraints must be of the type *NoHeapRealTimeThread* in order to avoid interference from the garbage collector.

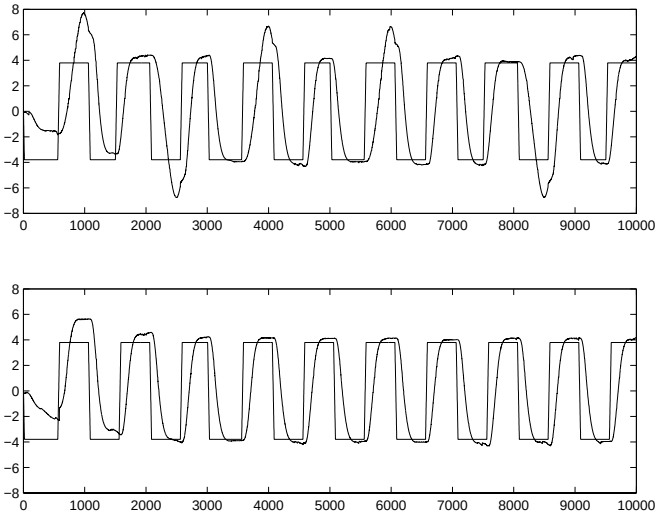

**Figure 8: Plots showing the reference value and the measured position for the ball-on-beam process. The upper plot shows the system without non-critical memory allocations and the lower plot shows the system where the allocation of plot data is non-critical. The irregular behaviour in the upper plot, around samples 2500, 4000, 6000, and 8500, is caused by the controller process being delayed by the garbage collector due to the program running out of allocatable memory and forcing a complete garbage collection cycle.**

In RTCE, real-time objects are allocated in *core memory*, and may not access objects on the garbage collected *baseline* heap. Objects on the heap may, with some restrictions, access core objects through special method calls. Core objects are allocated in an *allocation context*. When an allocation context is released, all objects in it may be eligible for reclamation but, since there might be references from the baseline heap, the actual reclamation is done by the baseline garbage collector when all of the objects in the allocation context are unreachable. Thus, a non-real-time garbage collector is used to reclaim the memory used by the real-time processes.

In RTCE, there are no limitations on which allocation contexts objects may reference so it is up to the programmer not to release an allocation context when it is still referenced.

RTCE also specifies stack allocation of real-time objects, which are to be automatically reclaimed as the scope is exited. To allocate stack objects, a set of restrictions apply and the reference must explicitly be declared stackable.

Under both of these specifications, the same behaviour as our system can be achieved by using one memory area (or

allocation context) for critical memory and another (or the heap) for the non-critical objects. The drawbacks of these approaches compared to ours are firstly that a much higher responsibility is placed on the programmer and that garbage collection cannot be used for the real-time threads and secondly that the access restrictions make communications between e.g. low and high priority threads more complicated.

## 7.2 Worst case analysis

Good worst case estimates for execution time[8] and memory usage[7] are crucial for making any kind of real-time guarantees. The experimental tool Skånerost[9] developed at our department provides interactive worst case execution time and memory consumption analysis based on timing schema and source code annotations for (currently a subset of) the Java language.

# 8. CONCLUSIONS

We have introduced the idea of applying priorities to memory allocation and shown how this can be used to enhance the robustness of real-time applications. The advantage this approach gives is twofold; firstly, it provides run-time support for prioritizing memory allocations if there is not enough memory for all allocation requests. Secondly, but equally important, it makes it easier to provide hard guarantees since the worst case memory usage calculations only has to be done for the critical parts of the system as non-critical allocations cannot cause the system to fail. Furthermore, we also suggest that the same mechanisms could be used to increase performance by limiting the amount of memory allocation and, consequentially, GC work.

Our approach is based on the notion of non-critical memory allocation requests, which can be used by the programmer to indicate that the memory allocations done in a certain part of the program are less important than the rest. Such non-critical allocations may be allowed to fail if the run-time system decides that that memory could be of better use elsewhere or that the increased garbage collection work would degrade system performance.

We also propose a way of introducing non-critical memory allocation in a Java system without making any changes to the syntax of the Java language and we have implemented this in an experimental Java virtual machine.

Preliminary experiments show that this mechanism is fairly easy to implement and can improve the robustness and performance of a control application by restricting its operation to the critical tasks if the system runs low on memory. It allows the programmer to write a system that performs better if run on a faster and larger system but whose critical tasks won't fail if it is run on a system with less than ideal amount of memory. Instead, the non-critical features of the system will automatically be turned off if there isn't enough memory for them to be safely executed.

# 9. REFERENCES


[1] Real-time core extensions. International J Consortium Specification, 2000.

[2] G. Bollella, et al. *The Real-Time Specification for Java*. Addison-Wesley, 2001.

[3] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 1998.

[4] A. Ive. *Implementation of an Embedded Real-Time Java Virtual Machine Prototype*. Lic. eng. thesis, Department of Computer Science, Lund Institute of Technology, Lund University, 2002. (to be published).

[5] R. Jones and R. Lins. *Garbage Collection. Algorithms for Automatic Dynamic Memory Management*. John Wiley & sons, 1996.

[6] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5), 1986.

[7] P. Persson. Live memory analysis for garbage collection in embedded systems. In *Proceedings of the ACM SIGPLAN 1999 Workshop on Languages, Compilers, and Tools for Embedded Systems (LCTES'99)*, Atlanta, Georgia, May 1999.

[8] P. Persson and G. Hedin. Interactive execution time predictions using reference attributed grammars. In *Proceedings of WAGA'99: Second Workshop on Attribute Grammars and their Applications*, Amsterdam, The Netherlands, March 1999.

[9] P. Persson and G. Hedin. An interactive environment for real-time software development. In *Proceedings of the 33rd International Conference on Technology of Object-Oriented Languages (TOOLS Europe 2000)*, St. Malo, France, June 2000.

[10] L. Sha, R. Rajkumar, and J. P. Lehoczky. Generalized rate-monotonic scheduling theory. *Proceedings of the IEEE*, 82(1), 1994.