Applying Priorities to Memory Allocation

Sven G. Robertz
Department of Computer Science
Lund University
Box 118, SE-221 00 Lund, Sweden
sven@cs.lth.se

ABSTRACT

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. The proposed mechanisms can also be used to increase performance of systems with automatic memory management by limiting the amount of garbage collection work.

A way of introducing priorities for memory allocation in a Java system without making any changes to the syntax of the language is proposed and this has been implemented in an experimental Java virtual machine and verified in an automatic control application.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and application-based systems—*Real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*Memory management, run-time environments*

General Terms

Reliability, Performance

Keywords

Memory allocation, real-time garbage collection, embedded systems, robustness

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, or component, 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* (NC) 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 [6]. 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 [8].

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 processes
3. Low priority processes and incremental garbage collection

ISMM'02, June 20-21, 2002, Berlin, Germany.

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 requirement 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.

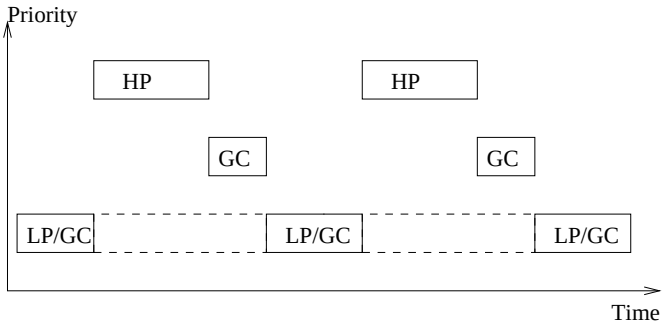

Figure 1: Dividing the CPU time between processes. The system consists of one periodic high priority process (*HP*) and one low priority process (*LP*). Whenever a high priority process is suspended, and no other HP process is eligible for execution, the garbage collector (*GC*) is run. GC work is also 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 or the amount of memory available to the application 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 being 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 target platform properties like memory size.

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 requirements.

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 [9, 14, 11].

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 increasing the amount of CPU time available to the application. This can, in turn, improve the application's performance by, e.g., allowing more advanced algorithms to be used.

Furthermore, in a real-time GC system, such as the one devised by Henriksson [6], 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 performed, but the less important task of the HP process is not. Hence, for the quality of service of the system, performance can be tuned in a more flexible and appropriate manner.

[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 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. Here, this is taken one 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.

Particularly, in automatically managed memory systems, where we have the problem with floating garbage[2], 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 that.

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 accomplish that by introducing an allocation limit for non-critical allocations; if there is less free, or *allocatable*[3], 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 (critical) 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 critical 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. Non-critical memory allocations should only be allowed if they won't cause the amount of allocatable memory to drop below R_{HP}.

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

[3]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.

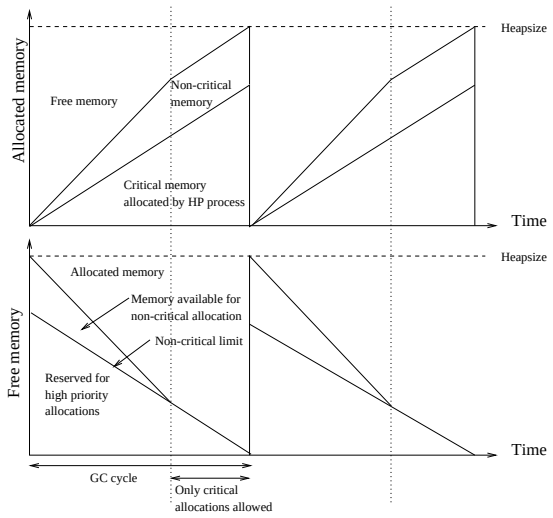

Figure 2: Schematic illustration of the limit for non-critical allocations. The dotted lines indicate the times where the non-critical limit is equal to the amount of allocatable memory, i.e., when the system starts to deny non-critical allocation requests.

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 an 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. DETAILED DESCRIPTION

This section describes the suggested approach in more detail. We discuss how the garbage collection cycle length can be calculated, how the decisions about when to deny non-critical memory allocation requests are taken, how the scheduling can be done and finally we give an example of how such a system may work.

5.1 Calculating the GC cycle length

Since we want to be able to make guarantees that the application never will run out of memory while still having hard real time constraints, we need a simple model so that we can make e.g., schedulability analysis. This is done by using a fixed GC cycle time which is calculated at application design-time.

The GC cycle time, the allocation rate of the HP process and the amount of memory available for non-critical allocation all affect each other and there are several ways to calculate the cycle length. One approach is to define how much memory should be reserved for HP allocations each GC cycle, M_{HP}. If the HP process allocates m_{HP} each period we get the GC cycle length expressed in HP periods:

$$T_{GC} = n \cdot T_{HP};\ n = \frac{M_{HP}}{m_{HP}}$$

Here, the GC cycle length will be the same regardless of how much total memory the system has and changes to the amount of memory will only affect how much non-critical allocation that can be made.

Another way is to define the ratio of memory reserved for HP processes to non-critical memory. This has the advantage that the application will behave in the same way, with respect to non-critical allocations, independent of how much memory the system it's running on has. This is preferable since while non-critical allocation cannot cause an out of memory situation, they add to the amount of GC work that has to be done and thus affect the schedulability analysis. Using the ratio of critical to non-critical memory instead of a fixed amount for one of the quantities has the property that the (amortized) amount of GC work per allocated object is independent of the total size of the memory — the memory size only affects the length of the GC cycles. Thus, this approach reduces the platform dependency of the schedulability analysis.

5.2 Live memory and floating garbage

In all calculations we must account for the amount of memory that lives across GC cycle boundaries and floating garbage that may exist in the worst case. This can be viewed as a reduction of the (usable) heap size with a constant. If this isn't taken into account, there will be less available memory at the start of each GC cycle than we have calculated with and the application will run out of memory.

Less obviously, it is also a problem if there is *more* allocatable memory at the start of a GC cycle than in the worst case, since this leads to the amount of memory available for non-critical allocations becoming too large, which could cause problems later. Therefore, we need to compensate for this, so that we always assume the worst case (i.e., we reserve a portion of memory to allow the amount of live memory or floating garbage to increase in the future).

With this taken into consideration, the least amount of free memory required for non-critical allocations during period i can now be expressed as

$$L_{NC_i} = (n-i)m_{HP} + f(A_{start}, C)\ ;\ 1 \le i \le n$$

where A_{start} is the amount of allocated memory at the start of this cycle, C the maximum amount of live and floating objects, and

$$f(x,y) = \begin{cases} y - x & , x < y; \\ 0 & , x \ge y; \end{cases}$$

5.3 GC for the low priority processes

When we add LP processes to the system, they will also allocate memory but the GC work corresponding to their allocations will be done at allocation time using traditional incremental GC. When LP allocations are done, the actual GC cycle time will be less than the nominal cycle time. In a traditional incremental garbage collector, this happens naturally; the extra GC work done by the LP process advances the current GC cycle.

In our system where GC work is triggered by time, however, we have to explicitly shorten the current GC cycle. Furthermore, the new, shorter cycle time still has to be a whole number of HP process periods to ensure that there always is enough allocatable memory for one full HP process invocation. This is done by decreasing the current cycle time by l HP periods, where

$$l = \left\lceil \frac{A_{LP}}{m_{HP}} \right\rceil$$

and A_{LP} is the amount of memory allocated by the low priority processes. Thus, if the nominal GC cycle length is n HP periods, the effective GC cycle length due to LP memory allocations will be n' HP periods, where $n' = n - l$.

Note that this should only affect the effective GC cycle length (i.e., the scheduling) and not the NC limit calculations. If we were to adjust the NC limit accordingly when the GC cycle was shortened, it would be possible for non-critical allocations in a HP process to "steal" the GC work done for a critical allocation in a LP process, and that is not what we want.

On the other hand, we do need to change the NC limit due to the actual LP allocations made, because if we don't, we would effectively reduce the amount of memory available for NC allocations. This may seem counter-intuitive but bear in mind that the purpose of the NC limit is to limit the amount of non-critical allocations and has nothing to do with controlling the critical allocations in the low priority processes.

As described above, when an allocation is made in a LP process, the corresponding GC work is done incrementally and the GC cycle is shortened so that there still will be memory for a whole number of HP process activations. Also, when a LP allocation is done, the amount of allocatable memory is decreased and in order to maintain the same amount of memory available to non critical allocations we have to reduce the NC limit with the same amount as the size of the LP allocation.

If we have allocated A_{LP} bytes of memory in the LP processes during this GC cycle, the NC limit can be written

$$L_{NC_i} = (n-i)m_{HP} + f(A_{start}, C) - A_{LP}$$

5.4 Non-critical limit calculations in the real world.

In all the previous calculations in this paper, we have assumed that a GC cycle can easily be divided into a number of HP process periods and that the memory allocations of each period are done instantaneously at the start of the period. This model is well suited for reasoning about systems and off-line analysis but doesn't lend itself well to actual implementation.

In real systems, the high priority processes often have different period times, and real programs do allocations more or less sporadically during their excecution rather than at the start of a well defined period. For these reasons, among others, a NC limit based on the number of elapsed HP periods is not a very practical one for run-time calculations. Instead, we will use the following algorithm:

- At the start of each GC cycle, the amount of memory needed by all the critical allocations by HP processes is calculated[4]. This is the amount of memory reserved for HP allocations (compensated for floating garbage, etc), $R_{HP} = M_{HP} + f(A_{start}, C)$

- Whenever a critical HP allocation is done, R_{HP} is decreased by the size of the allocated object. When an allocation is done by a low priority process, A_{LP} is increased. The non-critical limit is then updated; $L_{NC} = R_{HP} - A_{LP}$.

- If the amount of allocatable memory is less than or equal to L_{NC}, non-critical allocation requests will be denied.

This way, the NC limit will always be correct, regardless of how much memory the HP processes actually allocates and at what time during their execution they perform the allocations.

Another implementation issue is that our calculations assume that the garbage collector only frees memory at the very end of each GC cycle. This simplifies the non-critical limit calculations as each cycle can be viewed independently but when implementing support for non-critical allocations, care must be taken to assure that this assumption holds.

Mark-sweep collectors, of course needs some attention as they, by nature, free memory continuously during the sweep phase. A copying collector has this behaviour in principle, but still might have to be modified; it does free all memory after the last object has been moved, but this could happen before the full GC cycle time has elapsed.

Thus, in any case the memory manager must be designed so that it does not make any memory available to the allocator until at the start of the next GC cycle. Otherwise, too many non-critical allocations might be allowed in the current cycle, which might cause problems later. This also means that if the GC work metric is conservative and the garbage collector finishes early, the freed memory should not be made available to the allocator until at the start of the next cycle.

5.5 Time-based GC scheduling

Traditionally, incremental garbage collectors have been implemented so that GC work has been triggered by memory allocation, and done in proportion to the amount of allocated memory. I.e., when half of the memory available at the start of the cycle has been allocated, half of the GC work required to complete the cycle has been done and when all the memory has been allocated the GC cycle is completed.

That approach to garbage collection scheduling has drawbacks when it comes to non-critical allocations. Firstly, it may cause an unnecessarily low memory utilisation; If the application does less critical allocations than its worst case the GC cycle will be longer. The limit for non-critical allocations, on the other hand, is not affected, so when the amount of allocatable memory reaches the non-critical limit, no more non-critical allocations are allowed during that GC cycle. Thus, the less critical memory the application allocates, the longer the GC cycle gets and the less non-critical allocations are allowed, which is not what we want.

Secondly, each time a high priority process is scheduled to run, we need to make sure that there is always enough memory for it to do all the allocations it needs during its execution. Otherwise, that HP process would be delayed by panic garbage collection. Such guarantees can indeed be made for systems with allocation-triggered GC [6], but the calculations are quite complex and more suitable for off-line schedulability analysis.

Therefore, we use time, rather than allocation, as the trigger for GC work and do GC work in proportion to how large a fraction of the GC cycle time has elapsed. I.e., when half of the GC cycle time has elapsed, the GC should have done (at least) half the work needed to complete the cycle. This ensures that each GC cycle finishes within the fixed time, even if there is allocatable memory left. Thus, time-triggered GC ensures the same non-critical memory behaviour regardless of how much critical memory the application allocates.

5.6 Example

As an example, we take a system with one high priority process doing both critical and non-critical memory allocations and a set of low priority processes doing critical memory allocations.

In figure 3 you see how the amount of allocated and allocatable memory, respectively, varies over three GC cycles. In the first GC cycle, the amount of memory reserved for critical HP allocations (or rather, the non-critical limit) is larger than in the other two. This is because we must compensate for the fact that there is *less* than the maximum amount of allocated memory at the start of the GC cycle (see Section 5.2).

The second GC cycle shows how the system behaves when there are no allocations (and thus no incremental GC work) done by the low priority process. The first and third cycles are shortened since low priority allocations are done.

Since we have a fixed nominal GC cycle length and use time, rather than memory allocation, to trigger GC work the GC cycles may end before all available memory has been allocated. This can happen if the application uses less memory than in the worst case or due to quantization when low priority allocations are made (see section 5.3).

6. 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 semantics 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 4 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.

Non-criticality is transitive, i.e., memory allocations done in a method that is called from a non-critical region, like the calls to `foo(aNonCriticalObject)` and `doSomething()`

[4]The actual calculation of the worst-case memory requirements for each process could be done either manually or at compile time. Another possibility for soft real time systems is that it could be estimated by the run-time system based on measurements from previous GC cycles.

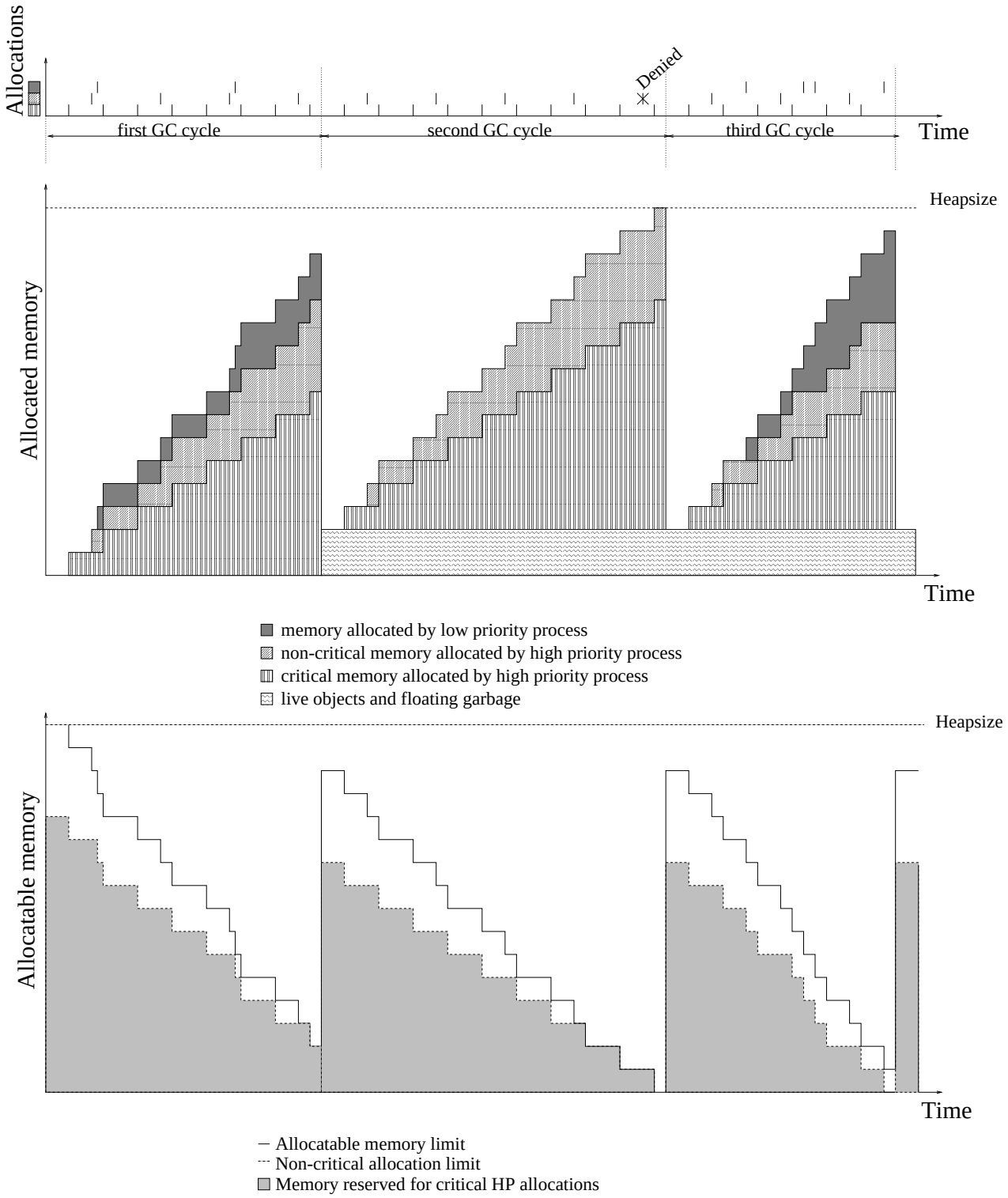

Figure 3: An example showing how the amounts of allocated and allocatable memory vary over time. Allocation requests for non-critical memory are denied when the amount of allocatable memory is less than or equal to the non-critical allocation limit ($R_{HP} - A_{LP}$). This happens at the end of the second GC cycle. Note that the first and third GC cycles are shorter than the nominal length due to low priority memory allocations. Also note how the non-critical limit is lowered when LP allocations are done so that the amount of memory available for non-critical allocations is not changed.

```
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 4: 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.

in Figure 4, 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 auxiliary function could be called both from critical and non-critical contexts.

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) [7], a very compact real-time Java virtual machine. Currently, we explicitly turn non-critical allocations on and off using a native method `IVM.setMemoryPriority()`. This is, however, 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 inserted for synchronized methods).

7. EXPERIMENTAL RESULTS

A simple control system was implemented to test the proposed technique. 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 5.

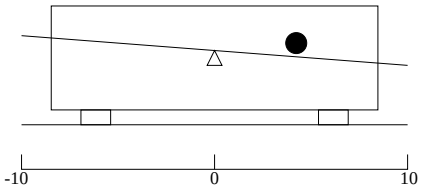

Figure 5: The ball-on-beam process. The beam can be rotated to roll the ball to the desired position. The position of the ball is in the interval $[-10, 10]$.

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.

The garbage collector used is an incremental mark-compact collector using time triggered semi-concurrent GC scheduling. The traces were collected by instrumenting the JVM with logging calls at memory operations and context switches.

7.1 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 significantly 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 6 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.

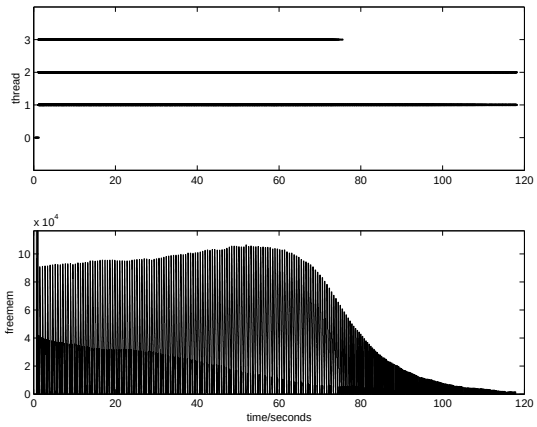

Figure 6: A sample run of the ball-on-beam system without non-critical memory. The UI thread *(3)* doesn't get enough CPU time to consume all plot data that is produced. After $t = 75$ it is totally starved by the GC. Then, less and less memory is available and more and more CPU time is spent doing panic GC.

[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.

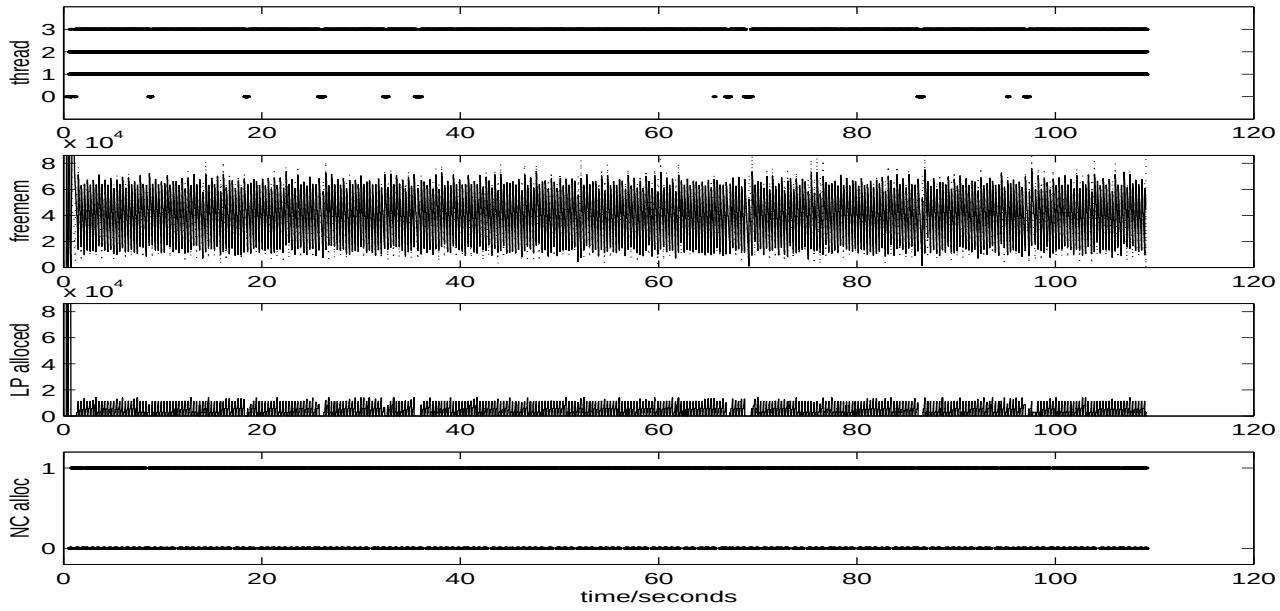

Figure 7: A run of the ball-on-beam system with log-data allocations made non-critical. In the thread plot you see that the UI thread gets CPU time throughout the run. The third plot shows the amount of memory allocated by low priority processes during this cycle. The fourth plot shows if non-critical allocations succeed or not; high level means success and low level is deny.

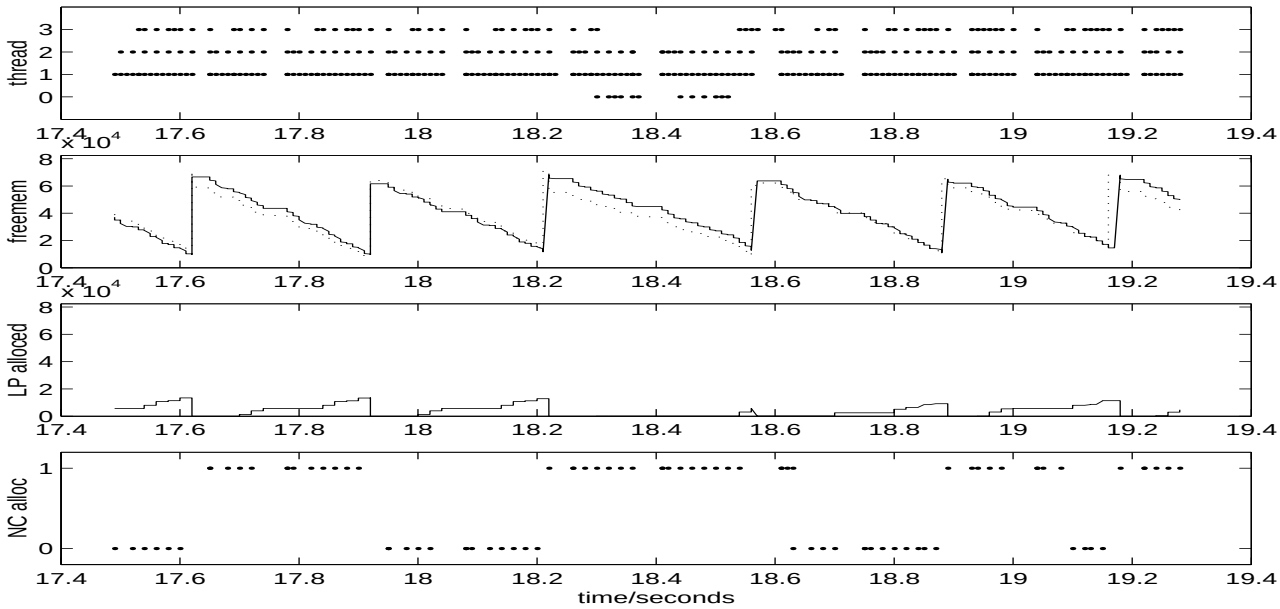

Figure 8: Close-up to show the non-critical memory behaviour.The dotted line in the freemem plot is the non-critical limit. Note how the GC cycles are shortened when low priority allocations are made.

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 frequent panic stop-the-world GC cycles 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.

Figure 7 shows the same system where the allocation of log data has been made non-critical, and the log data allocation is kept at a sustainable level. In this experiment, more than half of the log data allocation requests were allowed. Figure 8 shows a close-up of Figure 7 where you can see the non-critical behaviour more clearly.

7.2 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 9 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 allocation 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.

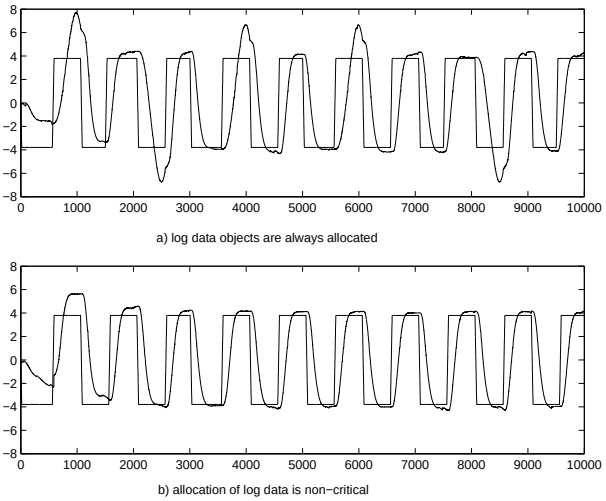

Figure 9: Plots showing the reference value and the measured position for the ball-on-beam process. Plot *a* shows the system without non-critical memory allocations and plot *b* shows the system where the allocation of plot data is non-critical. The irregular behaviour in *a*, around samples 2500, 4000, 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.

8. RELATED WORK

8.1 Memory Management in Real-Time Java

There are two specifications for real-time Java; The Real-Time Specification for Java (RTSJ) [4] and the Real-Time Core Extensions (RTCE) [3]. Both try to solve the real-time garbage collection problem by avoiding it. They assume that garbage collection is not feasible in real-time systems and instead propose 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 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.

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 the one proposed in this paper are firstly that a much higher responsibility is placed on the programmer by removing the safety that garbage collection provides, from the most critical parts of the system. Secondly, the access restrictions between the different types of memory make communications between low and high priority threads more complicated.

8.2 Soft references

Our notion of non-critical allocation is somewhat related to the *soft references* found for instance in Java [2] in that they both aim to prevent out of memory errors due to too many objects not absolutely needed for the correct operation of the program. In analogy with the Java terminology, non-critical allocations could be called "soft allocations".

[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.

The difference lies in *when* the system decides that it is running low on memory and starts trying to limit memory usage. With our approach, the decision is taken at allocation time, preventing a low on memory situation from arising. When using soft references, on the other hand, all allocations are carried out, and the decision about when to reclaim softly reachable objects are left to the garbage collector. There is also a difference in the intended usage; soft references were introduced to facilitate the implementation of e.g. caches, where objects' lifetimes are nondeterministic (i.e., you never know whether a cached value will be accessed again in the future or not, but it's best to keep it as long as the memory permits). Thus, while soft references may be used to achieve a similar logical behaviour as our non-critical allocations, the increase in the amount of required GC work when the system is already low on memory makes this use of soft references unsuitable for for real-time applications.

8.3 Worst case analysis

Good worst case estimates for execution time[12] and memory usage[11] are crucial for making any kind of real-time guarantees. The experimental tool Skånerost[13] developed at our department provides interactive worst case execution time and memory consumption analysis based on timing schema[15] and source code annotations for (currently a subset of) the Java language.

9. FUTURE WORK

Our preliminary experiments indicate that run-time support for dividing memory allocations into critical or non-critical can increase both robustness and performance of real-time software. However, more experiments on larger systems and systems with high performance requirements (e.g. low latency) will have to be done.

The semi-concurrent GC scheduling relies on that a priority-based process scheduler is used and is difficult to adapt to a deadline-based scheduler. We believe that the time-based GC scheduling proposed in this paper may be very suitable for making real-time GC scheduling feasible also in a deadline-based system (e.g., EDF).

It would also be interesting to study whether additional advantages may be gained from having an arbitrary number of memory priority levels compared to having just two (critical and non-critical).

9.1 Feedback Scheduling and Quality-of-Service

We believe that these ideas make it easier to handle memory management issues in the feedback scheduling and quality-of-service areas. For instance, by treating memory allocations just like any other resource allocation, it is possible to optimise the trade-off between memory usage and CPU time. I.e., increasing the allocation rate increases the GC workload which, in turn, reduces the application's CPU time and vice versa. It should be studied how this can be used in practise to optimise quality-of-service.

In a feedback scheduling based system, like the one described in [5], bringing the memory system into the loop coupled with the possibility to dynamically change an application's memory allocation behaviour depending on the system's current memory status could be used to get better overall performance. This is a very complex matter and strategies for when to deny non-critical memory allocations in order to optimise performance need further studies.

9.2 Configurable behaviour

Models for controlling when to fail non-critical allocations should be studied. In the logging example the optimal behaviour of the system depends on what the intended use of the log data is; if it is for system identification we want as long consecutive series of data as possible but the amount of time between the series is of less importance. I.e., we want every non-critical allocation request to be granted up to a point where no more non-critical requests are granted during that cycle. On the other hand, if the data is to be used for plotting, we want the samples to be equally spaced, i.e., every n-th NC allocation request should be granted. Furthermore, usually a set of allocations is needed in order to perform a certain task. If the last allocation of such a set is denied, the whole task has to be abandoned for that time. That should also be taken into account when deciding whether to grant or deny an allocation request.

Also, would it be possible to have different profiles to let the programmer choose among to get the one that fits a particular application best? Could such profiles co-exist in one application, i.e., different parts of the application having different non-critical memory policies?

9.3 Non-critical memory using aspects

This paper focuses on embedded real-time systems and the approach, as presented here, relies on the fact that we can modify the memory allocator. For systems without hard real-time constraints, however, we believe that it is possible to achieve the same advantages without having to do any modifications to the Java platform. One way of doing this could be by using aspect oriented software development[1]. The cross-cutting concern in this case is the handling of low-on-memory situations. It should be investigated whether it is suitable to e.g. divide the tasks into critical and non-critical aspects and dynamically weave in the non-critical parts only if the system has enough memory. We believe that it is possible to use e.g., the property-based cross-cutting of AspectJ [10] to insert a test whether an allocation should be done before each call to a constructor.

10. CONCLUSIONS

The idea of applying priorities to memory allocation was introduced and it was shown 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 prioritising 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.

We observe that memory priority and CPU time priority needs to be separated. Our logging example shows that a process having high CPU time priority doesn't necessarily mean that all of its memory allocations are critical.

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 the 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.

Acknowledgments

I wish to thank Roger Henriksson and Klas Nilsson for their valuable input and support during this work and Anders Ive for his help with implementing these ideas in the IVM. I also wish to thank the anonymous reviewers for their suggestions and comments on this paper.

This work has been financially supported by ARTES (A network for Real-Time research and graduate Education in Sweden) and SSF (the Swedish Foundation for Strategic Research). The experiments have been carried out in cooperation with projects financed by VINNOVA.

11. REFERENCES

[1] Aspect-oriented software development web site; http://www.aosd.net.

[2] Java 2 platform, standard edition, API specification. Sun Microsystems. http://java.sun.com.

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

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

[5] A. Cervin. *Towards the Integration of Control and Real-Time Scheduling Design*. Lic. eng. thesis, Department of Automatic Control, Lund Institute of Technology, Lund University, 2000.

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

[7] 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. (in preparation).

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

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

[10] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the Europeean Conference on Object Oriented Programming (ECOOP)*. Springer-Verlag, 2001.

[11] 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.

[12] 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.

[13] 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.

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

[15] A. C. Shaw. Reasoning about time in higher-level language software. *IEEE Transactions on Software Engineering*, 15(7), 1989.