Memory Resource Management for Real-Time Systems

Audrey Marchand, Patricia Balbastre, Ismael Ripoll, Miguel Masmano and Alfons Crespo*
Department of Computer Engineering
Technical University of Valencia, Spain

Abstract

Dynamic memory storage has been widely used for years in computer science. However, its use in real-time systems has not been considered as an important issue, and memory management has not receive much consideration, whereas today's real-time applications are often characterized by highly fluctuating memory requirements. In this paper we present an approach to dynamic memory management for real-time systems. In response to application behavior and requests, the underlying memory management system adjusts resources to meet changing demands and user needs. The architectural framework that realizes this approach allows adaptive allocation of memory resources to applications involving both periodic or aperiodic tasks. Simulation results demonstrate the suitability of the proposed mechanism.

1 Introduction

Dynamic storage allocation (DSA) algorithms have played an important role in the modern software engineering paradigms and techniques (as object oriented paradigm). Additionally, its utilization allows it to increase the flexibility and functionalities of the applications. There exists in the literature a large number of works and references to this particular issue. However, in the real-time community the use of dynamic memory techniques has not been considered as an important issue because spatial and temporal worst case for allocation and deallocation operations were insufficiently bounded.

Recently a new algorithm for dynamic memory allocation (TLSF) [21] that solves the problem of the worst case bound maintaining the efficiency of the allocation and deallocation operations allows the reasonable use of dynamic memory management in real-time applications. The proposed algorithm, with a constant cost $\Theta(1)$, opens new possibilities with respect to the use of dynamic memory in real-time applications. There exists an increasing number of emerging applications using high amounts of memory, such as multimedia systems, video streaming, video surveillance, virtual reality, scientific data gathering, or data acquisition in control systems.

Nowadays, new real-time applications require more flexibility being of major importance the ability to adjust system resources to load conditions. The system resources (CPU, memory, energy, network, disk, etc.) that an application can use, can be adapted to the global needs. One important issue is the bandwidth assigned to each task or group of tasks and how it can be dynamically changed when execution conditions are modified.

When considering CPU use, resource-based algorithms have been developed to characterize the timing requirements and processor capacity reservation requirements for real-time applications ([22, 3, 2, 16, 12]). Some works based on a job skipping scheme [13, 18] and providing flexible task models, have also been introduced. More recently, enhanced scheduling results based on the EDL (Earliest Deadline as Late as Possible) algorithm [6] have been proposed in [19] to optimize resource allocation in a job skipping context. In other works, as [11, 1, 9, 15], the focus has been oriented on networks. In [Ghosh04], an integrated framework for QoS optimization and scheduling for dynamic real-time sys- tems having multiple resource constraints was considered. It assumes an optimization criteria based on the maximisation of the global utility.

Besides the high number of papers related to resource management, the memory management has been completely forgotten. There are no papers, as far as the authors know, that consider the memory as a resource that can be adjusted depending on the system requirements. The reasons that can justify this lack can be summarised as:

- So far, these algorithms were designed to provide good average response times, whereas real-time applications will instead require the response times of the memory allocation and deallocation primitives to be bounded. For this reason, most real-time developers and researchers avoid using dynamic memory, or using it in a restricted way.

*This work was supported by the Spanish Government Research Office (CICYT) under grant DPI2005-09327-C02-02

- Applications can handle the need of dynamic memory with ad-hoc algorithms. The application allocates a large amount of memory that is handled by the application using small chunks of memory inside the allocated memory.
- Additionally, dynamic memory management can introduce an important amount of memory fragmentation, that may result in an unreliable service when the application runs for large periods of time. It can produce system degradation and inefficient use of resources.

However, the availability of a new dynamic memory allocator with not only bounded (constant) and fast response time, but also low and predictable fragmentation will permit us to consider the memory use as a resource in the system. Security, fault-tolerant real-time applications, robust systems can obtain benefits from memory management.

In [26], the authors pointed out the need of resource reservation in many application domains, such as, aerospace, multimedia, and real-time control systems. They state a set of propositions related to operating system services to provide quality of service to applications. They also ask for a more efficient memory management and an interface of the memory services at RTOS level.

1.1 Contributions of this paper

In this paper we present a new approach to consider the use of dynamic memory in real-time applications and to define the memory pool as a bandwidth abstraction where similar policies used for CPU or network can be applied. Specifically, the main contribution of the paper is to consider memomy as a first-class resource and to provide the schedulability conditions for systems with periodic and aperiodic tasks with known maximum resource usage. It implies:

- The definition of a new real-time task model which copes with both timing and memory constraints and the specification of a Memory Resource Manager that, jointly with a Dynamic Memory Allocator, efficiently assigns memory to tasks.
- The definition of an on-line acceptance test for dynamic memory requests that acts jointly with an acceptance test for CPU.

In the same way as processor scheduling, the proposed model allows to cope with the following situations: 1) Periodic tasks asking for additional memory to increase their quality during execution. 2) Sporadic tasks requiring enough memory resources to be executed. This is granted by means of an acceptance test that applies the memory model in the worst case situation for the execution interval.

1.2 Outline

The rest of the paper is organised as follows: Next section describes the work carried out in memory management. Section 4 presents a multimedia application that can be used as reference for the memory requirements. Section 3 states the basic requirements of real-time applications that use dynamic memory. In section 4, we describe a motivation example. Section 5 defines the task model for periodic and aperiodic tasks using dynamic memory. Section 6 and 7 presents the architecture, definitions and rules of a memory resource manager, and the memory analysis to guarantee dynamic changes in the requirements. A task set example using dynamic memory is detailed in Section 8. Section 9 describes the experimental evaluation of the proposed model. Finally, Section 10 states our conclusions and future lines of work.

2 Related work

Whereas languages perform an explicit memory allocation, the memory reclamation (deallocation) can be invoked explicitly or implicitly. Languages with explicit memory reclamation possess a freeing operator that take the address of the block to deallocate. Languages with implicit memory reclamation do not possess memory-freeing operators. Instead, an automatic reclamation mechanism (garbage collector) is engaged either when a value is no longer referenced or at a specified time. In this paper, we consider explicit memory reclaiming, so the application is in charge of the deallocation of memory blocks.

There are several dynamic storage allocation strategies that have been proposed and analysed under different real or synthetic loads. In [27] a detailed survey of dynamic storage allocation was presented. It has been considered the main reference since then.

Although it is possible to find a large number of papers about dynamic memory allocation, there are not many considering real-time constraints. In [24] a study of some allocators under real-time constraints is performed. In [21] and [20] a new allocator (TLSF - Two Level Segregated Fits) for real-time applications was presented and its temporal and spatial characteristics evaluated. The conclusions of these works showed that the worst-case execution time is asymptotically constant (not more than 170 processor instructions) whereas the experimental evaluation of the fragmentation is not more than 20% of the required memory for the application when real-time and non real-time loads were used.

On the other hand, very few of these works consider memory as a resource. In [8], the authors study the problem of FPGA based tasks using banks of memory. They try to minimise the number of memory banks that are shared by applications. In [10], the authors propose a memory

reservation scheme which permits any application to reserve a portion of the total system memory pages in exclusive. A reservation mechanism between applications is also defined.

Although this paper is focused on explicit memory reclaiming, we want to point out some works using implicit memory reclaiming mechanisms (garbage collector). There is a significant number of papers related to memory reclaiming and real-time. In [17] a deep analysis of the real-time garbage collection techniques is presented.

It is important to mention the efforts done to perform garbage collection under real-time constraints. One of the aspects is to reduce the garbage collector blocking time by performing incrementally or concurrently so as to achieve bounded and low worst-case latency [4, 14, 5]. In [25], a novel approach consisting in applying priorities to memory allocation is presented.

3 Application requirements

From the memory allocation point of view, the basic requirements of real-time applications that use dynamic memory are:

- **Bounded response time**. The worst case execution time of the allocation and deallocation operations has to be known in advance and be independent of application data. This is the main requirement that must be meet.
- **Fast response time**. Although having a bounded response time is a must, the response time has to be fast to be usable.
- **Memory allocations have to be always bounded**. Non-real time applications can receive a null pointer or are just be killed by the OS when the system runs out of memory. Although it is obvious that it is not possible to always grant all the memory requested, the DSA algorithm has to minimise the chances of exhausting the memory pool by minimising the fragmentation and wasted memory.

Currently, there are allocators that fullfil first two requirements (TLSF [21] and Half-fit [23]). The third requirement has to be analysed in more detail. While TLSF provides excellent results on fragmentation (at least as good as the best allocators), Half-fit allocator obtains a very high fragmentation. This implies that TLSF allocator is the most interesting option to implement dynamic memory allocation for real-time systems. We assume in the rest of the paper that TLSF is the allocator used.

In order to achieve the third requirement, we propose to organise the memory management as a reservation mechanism. Each task allocating memory has to declare its needs in terms of space (amount of memory) and time (holding time). While periodic activities can specify the maximum amount by period and the maximum number of periods that hold it, non periodic activities can define the maximum amount of memory by activation assuming that the memory is released at the end of the activation.

The sum of these needs define the worst-case amount of used memory by the application or *maximum live memory* refered in the paper as M^L. Additionally, dynamic memory allocation presents a level of memory fragmentation or *wasted memory*[1]. Following the processor model, this wasted memory could be considered as spatial overhead. In the paper this wasted memory will be refered as M^w. Moreover, the allocator uses a data structure to organise the available free blocks (M^{ds}). Considering these aspects, the total amount of memory needed to fulfill the application requirements (M^T) can be expressed as:

$$M^T = M^L + M^w + M^{ds}$$

While data structures M^{ds} can be considered negligible (less than 10 Kbytes in TLSF), fragmentation M^w is still an unsolved issue. From the point of view of real-time systems which have to operate over very long periods, fragmentation can play an important role in system degradation.

3.1 How to grant memory requests

Whereas some allocators achieve low fragmentation levels, others have very high fragmentation. Although theoretical analysis are very pessimistic, experimental analysis using TLSF with different load classes showed worst case experimental fragmentation less than 20% [7, 20]. Fragmentation is measured as the maximum amount of memory used by the allocator relative to the maximum amount of allocated memory. In other words, a 20% value means that we need a total memory (M^T) 20% higher than the maximum amount of bytes allocated and not freed by the application (M^L) considering that data structures are negligible.

The nondeterministic behavior of the fragmentation can be arised in terms of probability. For instance, if the total memory is designed with an overhead of 30% of the Live_memory, we can say that any request not exceeding Live_memory will always be satisfied. This approach needs a formal analysis depending on the block size requested and the holding time. This approach is beyond the scope of this paper and it is stated as future work.

[1] Although the term "wasted memory" describes better the inability to use some parts of the memory, historically the term "fragmentation" has been used.

4 Motivation example

In this section, we restrict our discussion to the multimedia applications context as a motivational example. The objective is to underline the fact that operating systems need to improve resource management in order to be able to ensure real-time processing needed by multimedia applications. Up to now, processor scheduling was the most important factor to improve to achieve this, while data memory aspects of multimedia systems were neglected. However, they are especially important in applications such as video streaming or data acquisition in which enormous amounts of data are processed, thus involving hard memory constraints for the application. This is the issue we propose to illustrate here, with the description of a real-time video play-out application (Figure 1).

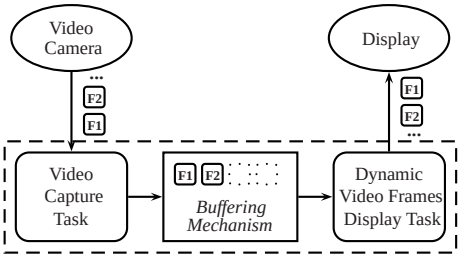

Figure 1. A real-time video application

Video data is first captured and digitized through a video capture device such as a video camera. Then, the video capture task receives the incoming video frames to be processed, and transfer them periodically to the buffering mechanism, with a period T_{input}. The buffering mechanism consists in storing frames in buffers. For convenience, we assume that the successive frames are numbered 1, 2, etc. A second task named "Dynamic Video Frames Display" consumes frames from the buffering mechanism, with a period T_{output}, and send the acquired video frames to a display device.

We assume that the buffers used for the implementation of the buffering mechanism reside in main memory, which allow fast data access required for efficient play-out. Note that each video frame takes up a large amount of storage space. Hence, the number of video frames that can be stored in memory buffers is limited by the size of the application memory itself. As a matter of fact, we point out here the fact that the system has to ensure that the maximal usage of data memory will never exceed the available memory size. The underlying system requirement is then the following: both processor and memory utilization have to be taken into consideration with the same weight.

Considering this system specification, we have to turn to the problem of the modeling of the application requirements. The leading solution of this problem is to define a suitable task model which will cope with both timing and memory constraints. Every task requires an amount of memory for data processing which is dynamically allocated during task execution. In addition, the example above shows that tasks can perform both memory allocations and memory deallocations (the "Video Capture Task" fills frame buffers involving memory allocations whereas the "Dynamic Video Frames Display Task" releases frames from buffers involving memory frees). However, from the system point of view, note that it does not matter to know which task has reserved or freed data memory. Indeed, the important issue necessary to meet the memory requirements relies on the knowledge of both the size of the allocated memory (here the size of the buffer mechanism) and the time during which data remain in memory.

In this work, we are focusing on memory as the resource to be managed. Because users need only to specify what the application requirements are and not how to achieve them, the underlying memory management details are effectively hidden from the user and application developers. In the following sections, we define the task model used and we examine the architectural design of the memory resource management component we have developed.

5 Task model and assumptions

In this section, we formally define the task model motivated in section 4. Let $\tau = \{\tau_1, ..., \tau_n\}$ be a periodic task system. It is assumed that a periodic task, requiring dynamic memory, requests each period an amount of memory. This amount of memory is allocated as result of one or several dynamic memory requests. Allocated memory is freed after some time interval by the same or other task. Note that in this model, it is not relevant which task frees memory, once it has been allocated the relevant aspect is the holding time.

Taking into account this behavior, each task $\tau_i \in \tau$ has the following temporal parameters: a worst-case computation time C_i, a period T_i, a relative deadline D_i, and an additional parameter M_i which describes the dynamic memory needs of task τ_i. Thus, a real-time set of periodic tasks consists in $\tau_i = (C_i, T_i, D_i, M_i)$. We will assume that all tasks meet their deadlines, i.e the system is CPU-schedulable.

In addition, M_i can be described by a 2-tuple considering the maximum amount of memory G_i^{max} requested each period, and the maximal time h_i^{max} during which allocations are persistent in memory (expressed in terms of numbers of periods of task τ_i). $M_i = (G_i^{max}, h_i^{max})$.

Note that $M_i^{max} = G_i^{max} * h_i^{max}$ can be considered as the maximum budget, in terms of memory, a task τ_i can have allocated. This task can obtain this maximum amount requesting blocks lower or equal than G_i^{max} at each period.

Under normal conditions, a task with M_i^{max} memory allocated has to free memory before requesting more memory or reusing allocated memory. This is done at most every h_i^{max} activations of task τ_i. Also, we assume that every task τ_i claims the memory it needs for executing at the release time of its requests. Figure 2 shows an example of the memory requests performed by a periodic task τ_i.

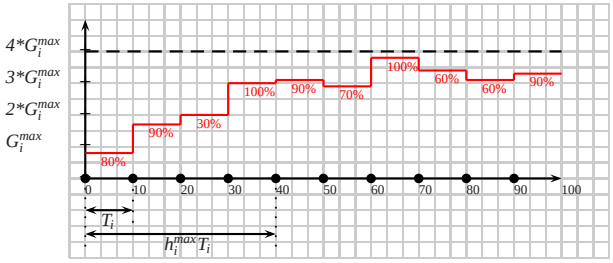

Figure 2. Memory requests of a periodic task

We assumed here that the allocations of task τ_i persist in memory during maximum h_i^{max} periods. What we can see is that after a transient period of time equal to $h_i^{max}T_i$, the amount of memory used by the task "oscillates". This is due to the balance of the memory allocations and frees performed by the task. For instance, at time $t = 40$, task τ_i frees an amount of 80%*G_i^{max} while it requests an amount of 90%*G_i^{max}. That is why we observe an increase of (90%-80%)*G_i^{max}=10%*G_i^{max} of memory use.

Considering all the periodic tasks that use dynamic memory, the total amount of memory needed by τ can be expressed as:

$$M^L = \sum_i M_i^{max}$$

In the case of aperiodic tasks, we assume that each task activation, it has to allocate some amount of memory in one or more allocation requests. The allocated memory is used by the task during its execution and freed before the end of the activation. Aperiodic tasks are characterized by $A_i = (r_i, c_i, M_i)$ where r_i is the release time, c_i is the worst-case computation time and M_i the maximum amount of memory needed by A_i to execute. We assume that R_i is the response time of A_i. Note that its value depends on the aperiodic scheduling strategy. Consequently, M_i can be described by a 2-tuple considering the amount of memory g_i that can be allocated permanently to the task, and the time R_i during which allocations are persistent in memory $M_i = (g_i, R_i)$.

6 Memory Resource Management

Memory resource management provide rules and mechanisms to assign the memory pool (M^T) to the tasks from its initial requirements and considering its dynamic needs.

A memory resource manager (MRM) uses a dynamic memory allocator algorithm to serve task requests of memory blocks. Figure 3 represents the scheme of the MRM.

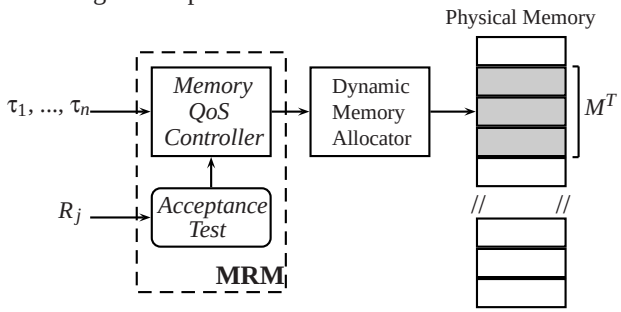

Figure 3. Memory Resource Manager (MRM)

When a real-time task requests dynamic memory, the MRM must decide whether sending the request to the Dynamic Memory Allocator or denying it. The last case occurs when there is not enough memory. Thus, the MRM applies certain rules and implements acceptance tests, since tasks not only share the CPU but also the memory pool. To state the rules managed by the "Memory QoS Controller" module, some definitions are needed:

Definition 1 *At a given time t, a task τ_i can have allocated a total amount of memory of $M_i^U(t)$. The total amount of allocated memory at time t is the sum of all memory allocations assigned to each task:*

$$M^U(t) = \sum_{i=1}^{n} M_i^U(t)$$

Definition 2 *At a given instant t, the MRM has an amount of available memory for serving future task memory allocation requests given by:*

$$M^F(t) = M^T - M^U(t)$$

Definition 3 *The maximum total amount of memory that τ_i can have allocated at time t can not exceed M_i^{max}. Thus, $M_i^U(t) \leq M_i^{max}$.*

Definition 4 *The memory utilisation factor of τ_i at t can be stated as:*

$$U_i^M(t) = \frac{M_i^U(t)}{M_i^{max}}$$

Definition 5 *The global utilisation factor of memory is given by:*

$$U^M = \max_t \left(\frac{1}{n} \sum_{i=1}^{n} U_i^M(t) \right)$$

being n the number of tasks.

Based in the previous definitions, the MRM define a set of rules to assign dynamic memory to tasks. It is assumed that a task requires only one dynamic memory request per activation.

Rule 1 *A memory block of size m is allocated to a task τ_i by the MRM if:*

$$U_i^M(t) + \frac{m}{M_i^{max}} \leq 1$$

Rule 2 *A task can, at any activation, deallocate any block of memory of size m. In this case the memory utilization factor of the task is decremented by the requested amount and the memory available is increased by the same amount.*

Rule 1 is a sufficient but not necessary acceptance test for real-time dynamic memory requests. At this point the MRM reserves an amount of memory of size M_i^{max} to each task τ_i. If at any time τ_i needs more than its reserved memory, the request is simply denied. Dynamic memory can be more efficiently managed by using reserved but not yet allocated memory of other tasks. The same reasoning can be applied when an aperiodic activity is released and it needs dynamic memory.

Rule 3 *At a given instant t, a periodic task τ_i can temporarily overrun its maximum memory budget M_i^{max}, provided the MRM has enough available memory to serve its request.*

Indeed, we consider that unexpected dynamic memory requests can occur. These can be caused either by memory budget overruns of periodic tasks (see Rule 3), or by arrivals of aperiodic tasks. In both cases, the MRM has to face the resulting surplus memory requirements. These are characterized as follows:

Definition 6 *The surplus memory requirements caused by the occurrence of an aperiodic task or by a memory budget overrun requested by a periodic task can be modeled by the following memory request:*

$$\Delta M_i = (\Delta G_i, L)$$

being ΔG_i the extra amount of memory requested and L the duration (i.e. the number of units of time) of the request.

With respect to the occurrence of an aperiodic task, note that ΔG_i corresponds to the amount of memory g_i required by the task to execute, while L refers either to its response time R_i in the case of soft aperiodic tasks, or to its relative deadline d_i for hard aperiodic tasks.

When an extra memory allocation is requested, the MRM has to take the following decisions:

- Would the available memory be enough to serve this new request along the interval of L duration?
- If not:
  - which is the size of the block that can be allocated successfully during the interval of duration L?
  - which is the interval during which the ΔM units of memory can be successfully allocated?

These questions can be answered by means of an schedulability test for temporal memory requests.

7 Schedulability analysis for memory allocation

At any time t, the MRM has to check that the allocated memory of all tasks is less than or equal to M^L (Rule 1). Whenever periodic tasks do not exceed their M_i^{max}, the global utilisation factor of memory remains below 1 and the system is schedulable from the memory resources point of view. Let's suppose that a task (periodic or aperiodic) requests some extra amount of memory $\Delta M_i = (\Delta G_i, L)$. The rules presented in section 6 are not valid to know if this request can be successfully allocated. We have to take into account the future memory requests of the task set in $[t, t+L[$.

Rule 4 *Let $\Delta M_i = (\Delta G_i, L)$ be a memory request occurring at t. The new task set $\tau \cup \Delta M_i$ is memory-schedulable if and only if :*

$$M^U(t) + M^R(t, t+L) + \Delta G_i \leq M^L$$

being $M^R(t, t+L)$ the total memory requests of periodic tasks in $[t, t+L[$.

7.1 Available memory calculation in $[t_1, t_2[$

Total memory requests of periodic tasks in $[t_1, t_2[$ can be calculated as:

$$M^R(t_1, t_2) = \sum_{i=1}^{n} (\mathcal{A}_i(t_1, t_2) - \mathcal{F}_i(t_1, t_2)) \quad (1)$$

where :

- $\mathcal{A}_i(t_1, t_2)$ is the amount of memory that is being to be allocated in $[t_1, t_2[$ to τ_i (in the worst case),
- $\mathcal{F}_i(t_1, t_2)$ is the amount of memory that is being to be freed in $[t_1, t_2[$ by τ_i.

7.1.1 $\mathcal{A}_i(t_1,t_2)$ calculation

In the worst case, τ_i will request the maximum amount of memory by period, that is, G_i^{max}. Then, we have:

$$\mathcal{A}_i(t_1,t_2) = \left\lceil \frac{t_2 - \left\lceil \frac{t_1}{T_i} \right\rceil * T_i}{T_i} \right\rceil * G_i^{max} \quad (2)$$

where $\left\lceil \frac{t_2 - \left\lceil \frac{t_1}{T_i} \right\rceil * T_i}{T_i} \right\rceil$ represents the number of activations of τ_i in $[t_1,t_2[$.

7.1.2 $\mathcal{F}_i(t_1,t_2)$ calculation

$\mathcal{F}_i(t_1,t_2)$ is calculated as:

$$\mathcal{F}_i(t_1,t_2) = \mathcal{F}_i^{alloc}(t_1,t_2) + \mathcal{F}_i^{next}(t_1,t_2) \quad (3)$$

where :

- $\mathcal{F}_i^{alloc}(t_1,t_2)$ represents the amount of memory yet allocated to τ_i at t_1 that will be freed in $[t_1,t_2[$,
- $\mathcal{F}_i^{next}(t_1,t_2)$ represents the amount of memory that will be both allocated and freed by τ_i in $[t_1,t_2[$. In this case, the amount of memory allocated and next freed corresponds to the worst case ($100\%G_i^{max}$).

$\mathcal{F}_i^{alloc}(t_1,t_2)$ **calculation.** We assume that the allocated memory for any task τ_i is known (the operating system, or the allocator keeps track of this information) and recorded in an array G of size equal to h_i^{max}. Therefore:

$$\mathcal{F}_i^{alloc}(t_1,t_2) = \sum_{k=0}^{q-1} G[k] * G_i^{max} \quad (4)$$

where $q = \min\left(\left\lceil \frac{t_2 - \left\lceil \frac{t_1}{T_i} \right\rceil * T_i}{T_i} \right\rceil, h_i^{max} \right)$ and $k \leq 1$.

$\mathcal{F}_i^{next}(t_1,t_2)$ **calculation.** The amount of memory requested by τ_i in the worst case in $[t_1,t_2[$ and freed also in $[t_1,t_2[$ is:

$$\mathcal{F}_i^{next}(t_1,t_2) = \left(\left\lceil \frac{t_2 - \left\lceil \frac{t_1}{T_i} \right\rceil * T_i}{T_i} \right\rceil - q \right) * G_i^{max} \quad (5)$$

where $\left(\left\lceil \frac{t_2 - \left\lceil \frac{t_1}{T_i} \right\rceil * T_i}{T_i} \right\rceil - q \right)$ represents the number of deallocations in $[t_1,t_2[$ of future memory requests in $[t_1,t_2[$.

7.2 Acceptance tests

Once we know how to calculate the state of the memory in a certain interval of time, we are going to present the acceptance tests as theorems for each kind of tasks, periodic, hard aperiodic and soft aperiodic. Moreover, when the memory request can not be accepted during a certain interval, we will present an algorithm to calculate the interval that can successfully allocate the request.

THEOREM 1 *Let $\tau_i(C_i,T_i,D_i,M_i)$ be a periodic task that is CPU-schedulable, and $\Delta M_i = (\Delta G_i, L)$ the extra memory needs expressed by the task. τ_i is memory-schedulable if and only if:*

$$M^U(t) + M^R(t,t+L) + \Delta G_i \leq M^L \quad (6)$$

This theorem can be rewritten for soft and hard aperiodic tasks. Assuming that the aperiodic task is CPU-schedulable (again, this depends on the server used to schedule aperiodic activities), we have to evaluate if there exists enough memory to serve the aperiodic task.

THEOREM 2 *Let $A_i(r_i,c_i,g_i,R_i)$ be a soft aperiodic task that is CPU-schedulable, and $\Delta M_i = (g_i,R_i)$ the surplus memory requirement generated by its occurrence. A_i is memory-schedulable if and only if:*

$$M^U(r_i) + M^R(r_i,R_i) + g_i \leq M^L \quad (7)$$

THEOREM 3 *Let $S_i(r_i,c_i,d_i,g_i)$ be a hard aperiodic task that is CPU-schedulable, and $\Delta M_i = (g_i,d_i)$ the surplus memory requirement generated by its occurrence. S_i is memory-schedulable if and only if:*

$$M^U(r_i) + M^R(r_i,d_i) + g_i \leq M^L \quad (8)$$

When previous theorems fail, it can be still possible to allocate the requested extra memory calculating the interval L_{sched} that satisfies the theorems. Consequently, the greatest interval in which memory requirements could be satisfied, taking into account the extra memory requirements $\Delta M_i = (\Delta G_i, L)$, can be determined iteratively thanks to the algorithm presented in Listing 1.

It is worth to observe that M^R is a non-decreasing step function that changes its value only at every task release. Thus, it is sufficient to evaluate it at time instants such that

$$\forall t \qquad t = kT_i \quad 1 \leq i \leq n$$

The algorithm starts evaluating the available memory at time t_1 that is the point in which the extra amount of memory is requested and goes forward as long as there is enough memory to serve the request ΔM_i. It will never reach $t_1 + L$,

Listing 1. Suitable interval algorithm

```
1  function Find_suitable_interval (t1, ΔGi, L) is
2    M^R = 0;
3    for t2 in t1 .. (t1 + L) loop
4      ∀τi ∈ τ
5        M^R(t1,t2) += Ai(t1,t2) − Fi(t1,t2);
6      exit when (M^U(t1) + M^R(t1,t2) + ΔGi ≥ M^L);
7      Lsched = t2 − t1;
8    end loop;
9    return Lsched;
10 end Find_suitable_interval ;
```

since L is the interval that do not satisfy the schedulability tests presented in the above theorems. Moreover, the same result is obtained if the algorithm starts at $t_1 + L$ and goes backwards until t_1. This property can be used to implement a faster algorithm starting the search in $t_1 + \frac{L}{2}$ (choosing the nearest task release), and applying the bisection algorithm to rapidly approach to the feasible interval. This algorithm can be used with both periodic or aperiodic tasks. We only have to set ΔM_i, that is the pair $(\Delta G_i, L)$ with the appropriate values. The complexity of the algorithm is linear in the number of tasks (O(n)).

Note that when we estimate $M^U(t)$ in the future, we have to assume the worst case. In this way, after a period equal to the size of vector G multiplied by the period of the task considered, i.e $h_i^{max} T_i$, we reach the point at which the task frees $100\% G_i^{max}$ and requests $100\% G_i^{max}$. Hence, $M^U(t)$ is a function that reaches a maximum value and it holds this value forever:

$$\max_t \left(M^U(t)\right) = M^U(t_{limit})$$

This point t_{limit} can be calculated as:

$$t_{limit} = \max_i \left(\left(\left\lceil \frac{t_1}{T_i} \right\rceil + h_i^{max} - 1 \right) * T_i \right)$$

8 Example

Let us consider a task set $\mathcal{T} = \{\tau_1, \tau_2\}$ of two periodic tasks whose parameters are described in Table 1.

Task	C_i	T_i	G_i^{max}	h_i^{max}	M_i^{max}
τ_1	5	10	9681	4	38724
τ_2	2	5	3546	6	21276

Table 1. A basic periodic task set with memory requirements

In this example, we do not consider the effects of fragmentation. The memory requests performed by tasks τ_1 and τ_2 up to time $t = 45$ are depicted in Figure 4:

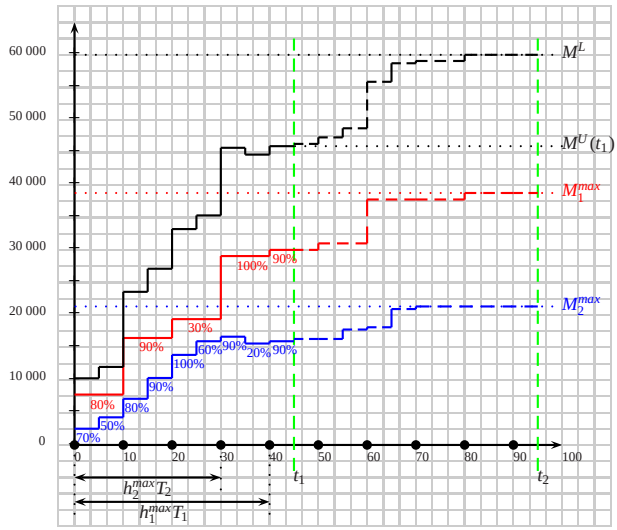

Figure 4. Memory requests of task set $\mathcal{T}$

Let us assume that at time $t = 45$ a soft aperiodic activity $A_j = (45, 3, 1275, 95)$ occurs. To know whether the task can be accepted or not, let us compute the maximum amount of memory that tasks τ_1 and τ_2 can allocate within the interval between the time the aperiodic request occurs and the one it finishes its execution. The memory requests of the tasks within the interval $[45, 95]$ are then estimated in the worst-case (*i.e* $100\% G_i^{max}$ per period). They are depicted in Figure 4.

In order to evaluate quantitatively the worst-case memory requests in $[45, 95]$, we have to calculate the amount of memory allocated ($\mathcal{A}_i(45, 95)$) and freed ($\mathcal{F}_i(45, 95)$) by tasks, using equations 2 and 3. Results are summarized in Table 2. Hence, we obtain that the total memory requests of the task set $\mathcal{T}$ in $[45, 95]$ is equal to:

$$M^R(45, 95) = \sum_{i=1}^{2} \left(\mathcal{A}_i(45, 95) - \mathcal{F}_i(45, 95)\right) = 14032$$

Task	$\mathcal{A}_i(45,95)$	$\mathcal{F}_i(45,95)$	
		$\mathcal{F}_i^{alloc}(45,95)$	$\mathcal{F}_i^{next}(45,95)$
τ_1	48405	30011	9681
τ_2	35460	15957	14184

Table 2. Memory use of τ_1 and τ_2 within $[45, 95]$

Furthermore, at time $t = 45$ the total memory already allocated by tasks τ_1 and τ_2 is given by (see Figure 4):

$$M^U(45) = 0.775M_1^{max} + 0.75M_2^{max} = 45968$$

As there are no more tasks in the system and assuming that the total reserved memory only takes into account the periodic task set $\mathcal{T}$, we can say that:

$$M^L = \sum_{i=1}^{2} M_i^{max} = 60000$$

Therefore, according to Theorem 2, we can accept A_j if and only if:

$$\begin{aligned} g_j &\leq M^L - M^U(45) - M^R(45,95) \\ &\leq 60000 - 45968 - 14032 \leq 0 \end{aligned}$$

Therefore, we conclude that A_j can not be scheduled from the memory resources point of view. However, as we can see in Figure 4, if the response time of A_j is shortened some memory can be allocated to A_j.

Hence, let us determine the greatest interval L_{sched} smaller than $[45,95[$ in which the aperiodic request is memory-schedulable. For that purpose we will use the algorithm previously described in section 7 (see Listing 1) with the following input values: $\Delta G_i = g_j$ and $L = R_i$, i.e $\Delta G_i = 1275$ and $L = 50$.

Table 3 shows the subsequent intervals evaluated at each step of the algorithm. In this example, 7 steps are necessary to find the greatest interval in which the memory requirements of the aperiodic request are satisfied.

Step	t_2	L_{sched}	Available memory in $[45,t_2[$
0	45	0	14032
1	50	5	13677
2	55	10	12709
3	60	15	11291
4	65	20	4160
5	**70**	**25**	**1323**
6	75	30	968

Table 3. Iterative memory computations

Consequently, we can say that the aperiodic request A_j is memory-schedulable provided its response time is shortened so that the request executes within $[45,70[$. Since the response time of the aperiodic request has been shortened, the CPU-schedulability test has to be re-evaluated in order to know if the new task set is CPU-schedulable.

9 Experimental evaluation

This section describes an experimental evaluation of the proposal with two periodic and one sporadic tasks using dynamic memory. Task parameters are shown in Table 4. Time units are given in miliseconds.

Task	C_i	T_i	G_i^{max}	h_i^{max}	G_i^{avg}	G_i^{stdev}	h_i^{min}
τ_1	4	20	6144	20	2048	100	14
τ_2	5	36	4096	14	1024	50	9
A_1	9	-	12275	-	2048	200	-

Table 4. Task set with memory requirements

Parameters G_i^{avg} and G_i^{stdev} define a normal distribution (average and standard deviation) used by task τ_i to request memory blocks that will be used during an interval randomly generated as a uniform distribution between h_i^{max} and h_i^{min} periods. The maximum heap size (M^L) is the sum of the maximum live_memory of each task (180 Kb). Assuming a 30% of fragmentation, the Heap size is $M^T = 234Kb$.

In this scenario, at time 3600, τ_2 is interested in using more memory (increasing the quality of the information) and asks for the MRM to use an increment of 60% of its G_i^{max} during 40 periods. The MRM can only guarantee 23 of the 40 requested periods. At the end of the 23 accepted periods, τ_2 requests the 17 remaining periods, but MRM only accepts 15 of them. Finally, a third negotiation occurs to complete the 40 requested periods which are guaranteed. Figure 5 shows this scenario. Horizontal lines represent the worst case memory used by τ_1, τ_2 and the heap size. Also, the evolution of the maximum address reached by the allocator is plotted in the figure. Note that in the example, tasks never reach its maximum memory use. This is due to the simulation model. Block sizes and holding times are obtained from a normal and uniform distribution whose maximum values correspond to the worst cases.

At time 9000, the sporadic task A_1 is ready to be executed. At this moment, the acceptance test is invoked. First, the processor acceptance test returns the response time which is used as holding time for the memory acceptance test. So, it asks for the availability of the needed memory (12275 bytes) which is granted by the MRM during the execution interval.

10 Conclusions and future work

This paper focused on the development of an approach to dynamic memory management for real-time systems. By using a dynamic resource management strategy tied to application requirements, we can ensure that tasks will have the appropriate amount of memory when needed. In addition to these memory guarantees, the memory management component allows the dispatching of the available memory so as to serve dynamic aperiodic requests or to provide periodic tasks with more memory. Simulation results showed

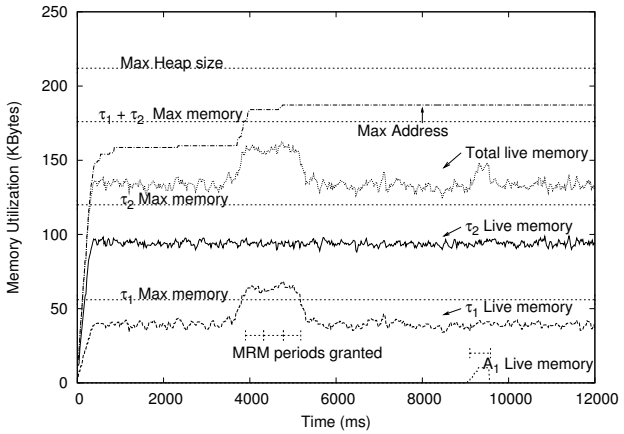

Figure 5. Memory requests of task set $\mathcal{T}$

that the proposed mechanism provides good results for large randomly generated task sets. Future work include extending this approach to dynamic memory management for quality of service provisioning. Moreover, we can further pursue solutions that can effectively address the problem of dynamically taking into account both CPU and memory requirements, thus linking the memory resource management component with the system task schedulers.

References

[1] T. F. Abdelzaher and K. G. Shin. End-host architecture for qos-adaptive communication. In *IEEE Real Time Technology and Applications Symposium*, pages 121–130, 1998.

[2] L. Abeni and G. Buttazzo. Resource reservation in dynamic real-time systems. *Journal of Real-Time Systems*, 27(2):123–167, 1998.

[3] L. Abeni, T. Cucinotta, G. Lipari, L. Marzario, and L. Palopoli. Qos management through adaptive reservations. *Journal of Real-Time Systems*, 29(2-3):131–155, 2005.

[4] D. Bacon and P. Cheng. The metronome: An simpler approach to garbage collection in real-time systems, 2003.

[5] Y. Chang and A. Wellings. Hard real-time hybrid garbage collection with low memory requirements. *IEEE RTSS*, 0:77–88, 2006.

[6] M. Chetto and H. Chetto. Some results of the earliest deadline scheduling algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

[7] A. Crespo, I. Ripoll, and M. Masmano. Dynamic memory management for embedded real-time systems. In *IFIP Conf. on Distributed and Pararallel Embedded Systems*, pages 195–204. Springer, 2006.

[8] K. Danne and M. Platzner. Memory-demanding periodic real-time applications on FPGA computers. In *WiP of the 17th ECRTS*, 6 - 8 July 2005.

[9] M. A. El-Gendy, A. Bose, and K. G. Shin. Evolution of the internet qos and support for soft real-time applications. *Proceedings of the IEEE*, 91(7):1086–1104, 2003.

[10] A. Eswaran and R. Rajkumar. Energy-aware memory firewalling for qos-sensitive applications. In *ECRTS*, pages 11–20, 2005.

[11] V. Firoiu, J. L. Boudec, D. Towsley, and Z. Zhang. Theories and models for internet quality of service, 2002.

[12] C. Hamann, J. Loser, L. Reuther, S. Schonberg, J. Wolter, and H. Hartig. Quality-assuring scheduling: Using stochastic behavior to improve resource utilization. In *22nd IEEE Real-Time Systems Symposium*, pages 119–128, 2001.

[13] M. Hamdaoui and P. Ramanathan. A dynamic priority assignment technique for streams with (m,k)-firm deadlines. *IEEE Transactions on Computers*, 44:1443–1451, 1995.

[14] R. Henriksson. *Scheduling Garbage Collection in Embedded Systems*. PhD thesis, 1997.

[15] E. Hernández-Orallo and J. Vila. Efficient qos routing for differentiated services ef flows. In *ISCC*, pages 91–96, 2005.

[16] K. Jeffay, F. D. Smith, A. Moorthy, and J. Anderson. Proportional share scheduling of operating system services for real-time applications. In *IEEE RTSS*, pages 480–491, 1998.

[17] M. S. Johnstone. *Non-Compacting Memory Allocation and Real-Time Garbage Collection*. PhD thesis, 1997.

[18] G. Koren and D. Shasha. Skip-over: algorithms and complexity for overloaded systems that allow skips. In *RTSS '95*, pages 110–119, Washington, DC, USA, 1995.

[19] A. Marchand and M. Silly-Chetto. Dynamic real-time scheduling of firm periodic tasks with hard and soft aperiodic tasks. *Real-Time Systems*, 32(1-2):21–47, 2006.

[20] M. Masmano, I. Ripoll, and A. Crespo. A comparison of memory allocators for real-time applications. In *JTRES '06*, pages 68–76, New York, USA, 2006. ACM Press.

[21] M. Masmano, I. Ripoll, A. Crespo, and J. Real. TLSF: A new dynamic memory allocator for real-time systems. In *16th ECRTS*, pages 79–88, Catania, Italy, July 2004. IEEE.

[22] C. W. Mercer, S. Savage, and H. Tokuda. Processor capacity reserves for multimedia operating systems. Technical report, Pittsburgh, PA, USA, 1993.

[23] T. Ogasawara. An algorithm with constant execution time for dynamic storage allocation. *2nd Int. Workshop on Real-Time Computing Systems and Applications*, page 21, 1995.

[24] I. Puaut. Real-Time Performance of Dynamic Memory Allocation Algorithms. In *14th ECRTS*, page 41, 2002.

[25] S. G. Robertz. Applying priorities to memory allocation. In *ISMM '02: Proceedings of the 3rd international symposium on Memory management*, pages 108–118. ACM Press, 2002.

[26] L. Steffens, G. Fohler, G. Lipari, and G. Buttazzo. Resource reservation in real-time operating systems - a joint industrial and academic position. In *ARTOSS'03*, pages 25–30, 2003.

[27] P. R. Wilson, M. S. Johnstone, M. Neely, and D. Boles. Dynamic Storage Allocation: A Survey and Critical Review. In *Int. Workshop on Memory Management*, volume 986 of *LNCS*, pages 1–16. Springer-Verlag, 1995.