Providing Memory QoS Guarantees for Real-Time Applications

A. Marchand, P. Balbastre, I. Ripoll and A. Crespo
Universidad Politécnica de Valencia, Spain
audrey.marchand@univ-nantes.fr
{patricia, iripoll, acrespo}@disca.upv.es

Abstract

Nowadays, systems often integrate a variety of applications whose service requirements are heterogeneous. Consequently, systems must be able to concurrently serve applications which rely on different constraints. This raises the problem of the dynamic distribution of the system resources (CPU, memory, network, etc.). Therefore, an integrated Quality of Service (QoS) management is needed so as to efficiently assign resources according to the various application demands. Within this paper, we focus on a dynamic approach of QoS management for memory resource allocation based on the Skip-Over model. We detail our solution and we show how it improves the service of task memory requests while providing them guarantees. Quantitative results using the TLSF allocator have been performed in order to evaluate the memory failure probability with and without memory QoS manager.

Keywords: Real-time, Quality of Service, Memory Management.

1 Introduction

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. Up to now, all the efforts have been focused on CPU, energy and network management, while memory has not been considered as a dynamic resource for the real-time community.

Recently a new algorithm for dynamic memory allocation (TLSF) [13] 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.

Viewing the memory as a dynamic resource to be shared among real-time tasks, implies to manage this resource and give some kind of guarantees. CPU scheduling techniques can be adapted and used to deal with memory management. In [11] is defined a memory management system that adjusts memory resources to meet changing demands and user needs.

When considering CPU use, several scheduling policies have been proposed to perform the CPU adaptation using different points of view. In particular, resource-based algorithms have been developed to characterize the timing requirements and processor capacity reservation requirements for real-time applications ([14, 2, 1, 7, 5]). Some works based on a job skipping scheme [6, 9, 12] and providing flexible task models, have also been introduced.

1.1 Summary and contributions

Memory allocation is the problem of maintaining an application's heap space by keeping track of allocated and freed blocks. The decision to be made by the memory allocator is where to place the requested block in the heap. The allocator has no information about when the blocks will be freed after they are allocated. The order of these requests is entirely up to the application.

This paper proposes a framework to minimise the number of fails in memory requests. The methodology proposed is based in skippable tasks, specifically, we adapt the Skip-Over model used in CPU scheduling to manage memory overruns.

2 Skip-over based CPU overload management

Different techniques have been proposed to deal with CPU overload management. To represent such quality of service constraints, Hamdaoui and Ramanathan in [6] proposed a model called *(m,k)-firm* deadlines. It guarantees that a statistical number of deadlines will be met, by using a *distance-based* priority scheme to increase the priority of an activity in danger of missing more than m deadlines over a window of k requests. If $m = k$, the system becomes a hard-deadline system.

This problem is solved for the special case $m = k - 1$ for which the (m,k) model reduces to the *Skip-Over* model [9]. The skip-over scheduling algorithms skip some task invocations according to a skip factor. The overload is then reduced, thus exploiting skips to increase the feasible periodic load.

In what follows, we focus on the significant Skip-Over approach. Known results about the feasibility of periodic task sets under this model are also recalled.

2.1 Model description

The Skip-Over model [9] deals with the problem of scheduling periodic tasks which allow occasional deadline violations (*i.e.* skippable periodic tasks), on a uniprocessor system. A task τ_i is characterized by a worst-case computation time C_i, a period T_i, a relative deadline equal to its period, and a skip parameter s_i. This parameter represents the tolerance of this task to miss deadlines. That means that the distance between two consecutive skips must be at least s_i periods. When s_i equals to infinity, no skips are allowed and τ_i is a hard periodic task. Every instance of a task is either red or blue [9]. A red task instance must complete before its deadline whereas a blue task instance can be aborted at any time.

Two scheduling algorithms were introduced about ten years ago by Koren and Shasha in [9]. The first one proposed is the Red Tasks Only (RTO) algorithm. Red instances are scheduled as soon as possible according to Earliest Deadline First (EDF) algorithm [10], while blue ones are always rejected. The second algorithm introduced is the Blue When Possible (BWP) algorithm which is an improvement of the first one. Indeed, BWP schedules blue instances whenever their execution does not prevent the red ones from completing within their deadlines. In other words, blue instances are served in background relatively to red instances.

2.2 Feasibility of skippable periodic task sets

Liu and Layland in [10] have shown that a task set $\{\tau_i; 1 \leq i \leq n\}$ is schedulable if and only if its *cumulative processor utilization* (ignoring skips) is no greater than 1, *i.e.*,

$$\sum_{i=1}^{n} \frac{C_i}{T_i} \leq 1. \quad (1)$$

Koren and Shasha proved that determining whether a set of periodic occasionally skippable tasks is schedulable is NP-hard [9]. However, they have shown the following necessary condition for schedulability for a given set $\Gamma = \{\tau_i(C_i, T_i, s_i)\}$ of periodic tasks that allow skips:

$$\sum_{i=1}^{n} \frac{C_i(s_i - 1)}{T_i s_i} \leq 1. \quad (2)$$

In [3], Caccamo and Buttazzo introduced the notion of *equivalent utilization factor* defined as follows.

Definition 1 *Given a set $\Gamma = \{\tau_i(C_i, T_i, s_i)\}$ of n periodic tasks that allows skips, the equivalent utilization factor is defined, for any interval $L \geq 0$, as:*

$$U_p^* = \max_{L \geq 0} \frac{\sum_i D(i, [0, L])}{L} \quad (3)$$

where

$$D(i, [0, L]) = (\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{T_i s_i} \rfloor) c_i. \quad (4)$$

$D(i, [0, L])$ stands here for the minimal computation demand of Γ over the interval L.

The same authors also provided a necessary and sufficient condition for guaranteeing a feasible schedule of a set of skippable tasks which are deeply-red (*i.e.* all tasks are synchronously activated and the first $s_1 - 1$ instances of every task τ_i are red) [4]:

Theorem 1 *A set Γ of skippable periodic tasks, which are deeply-red, is schedulable if and only if*

$$U_p^* \leq 1. \quad (5)$$

3 Skip-Over based memory overload management

3.1 New task model and notations

In this section, we formally define the task model used. 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.

Taking into account this behaviour, 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, the dynamic memory needs M_i and an additional parameter s_i which gives the tolerance of this task to memory failures. Thus, a real-time set of periodic tasks consists in $\tau_i = (C_i, T_i, D_i, M_i, s_i)$.

In addition, M_i can be described by a 2-tuple $M_i = (g_i, h_i)$ considering the maximum amount of memory g_i requested each period, and the maximal time h_i during which allocations are persistent in memory (expressed in terms of numbers of periods of task τ_i). Consequently, the amount of memory used by the application in the worst-case is given by $\sum_i h_i g_i$.

3.2 Memory feasibility of skippable periodic task sets

As an analogy to the processor demand criteria [8], we turned to another form of schedulability test: the *memory demand* criteria.

Definition 2 *Given a set* $\Gamma = \{\tau_i(C_i, T_i, M_i, s_i)\}$ *of* n *skippable periodic tasks with memory constraints, the equivalent memory utilization factor is defined as:*

$$M^* = \max_{L \geq 0} \sum_i D(i, [0, L]) \tag{6}$$

where

$$D(i, [0, L]) = (\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{T_i s_i} \rfloor - \lfloor \frac{L - T_i h_i}{T_i} \rfloor + \lfloor \frac{L - T_i h_i}{T_i s_i} \rfloor) g_i.$$

Proof: Let denote by $D(i, [0, L[)$ the total memory demand within $[0, L[$ for task τ_i. First, let us evaluate the amount of memory requested by a task τ_i over the interval $[0, L[$. Within any interval $[0, L[$, the number of periods observed for every task τ_i is equal to $\lfloor \frac{L}{T_i} \rfloor$, thus involving a total demand for memory allocations equal to $\lfloor \frac{L}{T_i} \rfloor g_i$. According to the Skip-Over definition, every task τ_i is allowed to skip one instance every s_i task activations. Thus, for every task τ_i, the total skipped memory allocations within $[0, L[$ is $\lfloor \frac{L}{T_i s_i} \rfloor g_i$. Let us now evaluate the amount of memory released by a task τ_i over the interval $[0, L[$. Without skips, this amount would be only equal to $\lfloor \frac{L - T_i h_i}{T_i} \rfloor g_i$, taking into account the fact that task τ_i does not perform any memory releases within the interval $[0, T_i h_i[$. However, every skippable periodic task τ_i does not release any memory every $T_i s_i$ periods. Hence, we have to withdraw from the previous quantity, an amount of memory corresponding to skipped task instances (*i.e.* non-allocated memory), which is equal to $\lfloor \frac{L - T_i h_i}{T_i s_i} \rfloor g_i$. Consequently, the total amount of memory remaining at time instant $t = L$ for task τ_i is $D(i, [0, L]) = (\lfloor \frac{L}{T_i} \rfloor - \lfloor \frac{L}{T_i s_i} \rfloor - \lfloor \frac{L - T_i h_i}{T_i} \rfloor + \lfloor \frac{L - T_i h_i}{T_i s_i} \rfloor) g_i$. It follows that the maximal memory utilization is given by $M^* = \max_{L \geq 0} \sum_i D(i, [0, L])$. ■

Theorem 2 *A set* Γ *of skippable periodic tasks, which are deeply-red, is memory-schedulable if and only if*

$$M^* \leq M^T - M^w - M^{ds}. \tag{7}$$

where M^w represents the wasted memory due to fragmentation and M^{ds} the data structures needed by the allocatorto organise the available free blocks.

4 R-MRM Implementation Framework

The *R-MRM (Robust-Memory Resource Controller)* implementation framework is the enabling technology for efficiently managing memory-constraint tasks using Skip-Over principles. It provides users with an operational framework to manage real-time applications.

The implementation framework partly relies on the framework previously proposed in [11]. R-MRM is a component that mediates between tasks and the dynamic memory allocator as depicted in Figure 1.

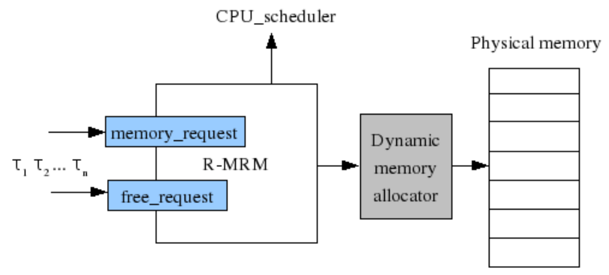

Figure 1. R-MRM external interaction view

R-MRM offers two kinds of operations that a task can perform: `memory_request` and `free_request`. The `memory_request` function involves the memory size requested and a deadline for this request.

When a task wants to free memory previously allocated, it calls the `free_request` function. The pseudo-code of the `memory_request` and `free_request` operations are shown in Listings 1 and 2.

The R-MRM component embeds three kinds of policies to address all its functionalities: a memory granting policy, a rejection policy and a recovery policy. All these policies

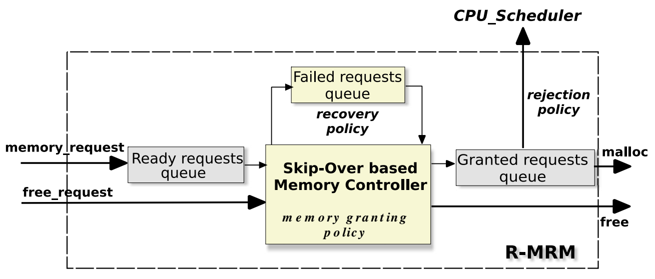

Figure 2. R-MRM internal view

Listing 1. Memory_request

```
function memory_request(size, deadline) is
  if (memory_granting_policy)
     insert in Granted requests queue
     malloc(size);
  else
     if (red_task)
       rejection_policy ;
       insert in Granted requests queue;
       malloc (size);
     else
       insert in Failed requests queue
       -- the calling task is blocked
     end if ;
  end if ;
end memory_request;
```

Listing 2. Free_request

```
function free_request (int ptr) is
  free (ptr);
  re-evaluate Failed requests queue;
end free_request ;
```

are implemented by the "Skip-Over based Memory Controller" sub-component in a centralized manner. It implements the necessary functions to keep track of the available memory in the system, manages the *Failed and Granted request queues* and controls the timer associated to the deadline of a blue blocked task in the *Failed requests queue*.

4.1 Framework Policies

In the following, we consider the case of a real-time application whose heap space has been properly sized according to condition (7) of Theorem 2. We are going to see in more detail the three policies aforementioned.

4.1.1 The memory granting policy

Let t be the current time which coincides with the arrival of a memory request I. Upon arrival, request $I(d, g, h)$ is characterized by its deadline d, its maximum amount of memory g and the maximal time h during which allocation is persistent in memory. We assume that several memory requests are present in the Granted request queue at time t. Let denote by $\mathcal{I}(t) = \{I_i(d_i, g_i, h_i), i = 1 \text{ to } req(t)\}$ the memory request set supported by the machine at t. Then, the acceptance problem to solve when any memory request I occurs, is reduced to the test of a necessary and sufficient condition:

Theorem 3 *Memory is granted to request I if and only if, considering the request set $\mathcal{I}(t) \cup I$, we have:*

$$\sum_{i=1}^{req(t)+1} h_i g_i \leq M^T - M^w - M^{ds} \tag{8}$$

If there is enough memory available then the allocation is granted, otherwise the request will undergo a recovery process aiming at attempting the request later on.

4.1.2 The recovery policy

If the memory granting policy determines that there is not enough memory to serve the request, the task is temporarily put into the queue named "Failed requests queue" (see Figure 2), waiting there to make another attempt at being accepted. According to the Skip-Over model, tasks in this queue are only blue and can exit the queue in the following cases:

- When it is freed (by other task or tasks) the sufficient amount of memory to serve its request. In this case, the request can be granted and the task exits the Failed requests queue.
- When the deadline is reached. Then, the task exits the Failed requests queue with a failure.

4.1.3 The rejection policy

The problem of the rejection decision consists merely on determining which blue task has to be rejected. The criterion set for rejection consists on identifying the blue task having the least actual failure factor.

Hence, we propose as a metric the Task Failure Factor (TTF_i) as the ratio between the number of failures observed for task τ_i since initialization time and its number of activations at time t:

$$TTF_i(t) = \frac{nb\_failures(t)}{\lfloor \frac{t}{T_i} \rfloor} \tag{9}$$

That means that the ready blue task whose failure ratio $TTF_i(t)$ computed from the initialization time is least, is candidate for rejection. Ties are broken in favour of the task with the earliest deadline. Note that this is an arbitrary metric.

5 Simulation Study

In this section, we evaluate how effective the proposed task model and scheduling scheme can solve the problem of guaranteeing memory allocation according to the QoS specification inherently provided by the Skip-Over task model.

5.1 Experiments

To evaluate the actual performance of our solution, we constructed a simulator that models the behaviour of the R-MRM component. Its evaluation was performed by means of four simulation experiments, all operating with the `memory_request` and `free_request` operations presented in Listings 1 and 2. The proposed scenarios (see Table 1) were specially designed to provide a comprehensive and comparative analysis of the proposed approach regarding the memory and QoS requirements previously exposed.

Test	s_i
1	∞
2	10
3	6
4	2

Table 1. Simulation scenarios

s_i parameters have been considered identical for all tasks in order to clearly demonstrate the influence of the QoS specification with respect to the memory failure probability observed.

Experiments were carried out for 100 randomly generated task sets with period T_i uniformly distributed in the range 20..250, maximal amount of memory G_i^{max} uniformly distributed in the range 4096..102400. Deadlines equals to periods. Additional 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.

5.2 Results

Our results are shown in several different ways in Figures 3 to 6. In all cases, the x-axis displays the different percentages of memory amounts provided for the application with respect to the total live memory (*i.e.* $\sum_i h_i g_i$) given by the task specification itself.

Four output parameters have been evaluated: the number of failed requests, the number of retries(*i.e.* the number of times a memory request is re-attempted), the number of solved requests, the number of overruns (this case occurring when a task reached its deadline without having received memory allocation). Figures 3, 4, 5 and 6 show the absolute memory failure probability for each output parameter for the four tested scenario.

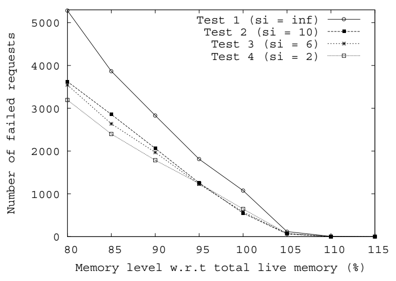

Figure 3. Number of failed requests according to s_i

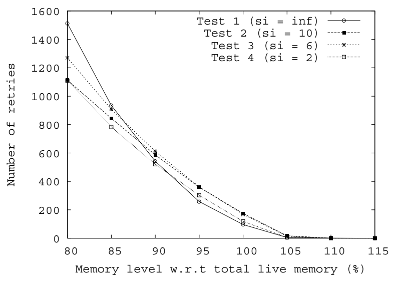

Figure 4. Number of retries according to s_i

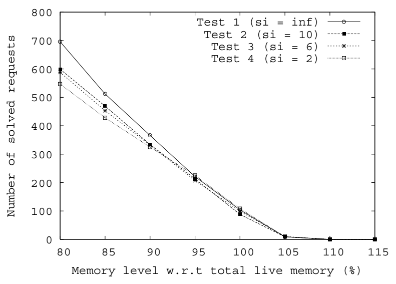

Figure 5. Number of solved requests according to s_i

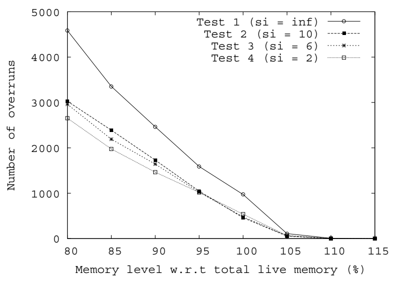

Figure 6. Number of overruns according to s_i

First, notice that all the curves decrease with increasing percentage of memory given to the application, which is a logical behaviour. Note also that for the remarkable point where the amount of memory is exactly equal to the total live memory (*i.e.* 100%), we observe a non-zero memory failure probability. This is due to the spatial overhead induced by the dynamic memory allocator (here the TLSF). For a zero memory failure probability, the amount of memory assigned to the application must be higher to take into account the data structures needed by the dynamic memory allocator for functioning.

On the other hand, as expected, we observe that the memory failure probability for $s_i = \infty$ (*i.e.* no skips allowed) is significantly higher than in the other scenarios. Interestingly, curves for $s_i = 10$, $s_i = 6$ and $s_i = 2$ have almost identical distribution for a memory level greater or equal to 95% of the total live memory.

6 Conclusions

While feasibility and schedulability analysis from the CPU point of view is well understood, memory analysis for real-time systems has received less attention. In this paper we addressed the problem of scheduling real-time task sets with memory constraints. In particular, we presented a memory feasibility analysis for skippable periodic task sets. The memory feasibility test contained in this paper represents the first known result for periodic real-time tasks based on the Skip-Over model. Our main contribution was actually to design a component for a skip-over based memory overload management and to evaluate it. Through the results, we showed to what extend the proposed R-MRM component can minimize the memory failure occurrence probability, while a QoS level (i.e., skip parameter established by the application programmer) is always guaranteed for tasks. The strong point of this approach relies on the memory guarantees provided by the component. We believe that the present approach is promising for enhancing the performance of memory-constraint applications and applying memory analysis in the real-time field.

References

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

[2] 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.

[3] M. Caccamo and G. Buttazzo. Exploiting skips in periodic tasks for enhancing aperiodic responsiveness. In *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97), San Francisco, California*, pages 330–339, 1997.

[4] M. Caccamo and G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems. In *Proceedings of fifth conference on Real-Time Computing Systems and Applications (RTCSA'98), Hiroshima, Japan*, 1998.

[5] 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.

[6] 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.

[7] 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.

[8] K. Jeffay and D. Stone. Accounting for interrupt handling costs in dynamic priority task systems. In *Proceedings of the 14th IEEE Real-Time Systems Symposium (RTSS'93), Raleigh-Durham, NC*, pages 212–221, 1993.

[9] G. Koren and D. Shasha. Skip-over algorithms and complexity for overloaded systems that allow skips. In *Proceedings of the 16th IEEE Real-Time Systems Symposium (RTSS'95), Pisa, Italy*, 1995.

[10] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM*, 20(1):46–61, 1973.

[11] A. Marchand, P. Balbastre, I. Ripoll, M. Masmano, and A. Crespo. Memory Resource Management for Real-Time System. In *Proceedings of the 19th Euromicro Conference on Real-Time Systems, Pisa, Italy*, 2007.

[12] 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.

[13] M. Masmano, I. Ripoll, A. Crespo, and J. Real. Tlsf: A new dynamic memory allocator for real-time systems. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems, Catania, Italy*, 2004.

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