Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-chip

Paolo Gai, Giuseppe Lipari, Marco Di Natale
ReTiS Lab, Scuola Superiore di Studi e Perfezionamento S. Anna – Pisa,
{pj,lipari,marco}@sssup.it

Abstract

The research on real-time software systems has produced algorithms that allow to effectively schedule system resources while guaranteeing the deadlines of the application and to group tasks in a very short number of non-preemptive sets which require much less RAM memory for stack. Unfortunately, up to now the research focus has been on time guarantees rather than the optimization of RAM usage. Furthermore, these techniques do not apply to multiprocessor architectures which are likely to be widely used in future microcontrollers.

This paper presents a fast and simple algorithm for sharing resources in multiprocessor systems, together with an innovative procedure for assigning preemption thresholds to tasks. This allows to guarantee the schedulability of hard real-time task sets while minimizing RAM usage. The experimental part shows the effectiveness of a simulated annealing-based tool that allows to find a near-optimal task allocation. When used in conjunction with our preemption threshold assignment algorithm, our tool further reduces the RAM usage in multiprocessor systems.

1. Introduction

Many embedded systems are becoming increasingly complex in terms of functionality to be supported. From an analysis of future applications in the context of automotive systems [9] it is clear that a standard uniprocessor microcontroller architecture will not be able to support the needed computing power even taking into account the IC technology advances.

To increase computational power in real-time systems there are two possible ways: increase the processor speed or increase the parallelism of the architecture. The first option requires the use of caching or deep pipelining which suffer from serious drawbacks in the context of real-time embedded systems. Therefore, the best option and the future of many embedded applications seems to rely on the adoption of multiple-processor-on-a-chip architectures.

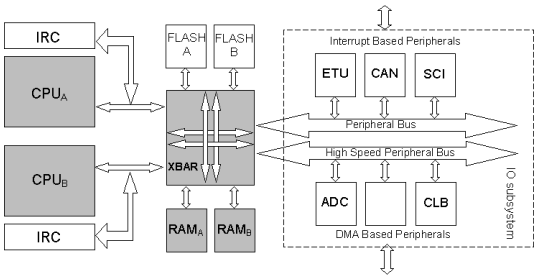

Figure 1. The Janus Dual Processor system

The Janus system, (see the scheme of Figure 1) developed by ST Microelectronics in cooperation with Parades [9], is an example of a dual-processor platform for power train applications featuring two 32-bit ARM processors connected by a crossbar switch to 4 memory banks and two peripheral buses for I/O processing. The system has been developed in the context of the MADESS[1] project. The applications must satisfy a very demanding requirement: in addition to real-time predictability, the OS and the application must use the smallest possible amount of RAM memory. RAM is extremely expensive in terms of chip space and impacts heavily on the final cost.

In the design of the kernel mechanisms for the ERIKA kernel [10], it had been clear from the beginning that the choice of the real-time scheduling discipline influences both the memory utilization and the system overhead: for example, selecting a non-preemptive scheduling algorithm can greatly reduce the overall requirement of stack memory whereas using a preemptive algorithm could increase the processor utilization.

The idea behind this work is based on the concept of non-interleaved execution. As explained in Section 4, using a protocol called Stack Resource Policy (SRP) [1],

[1] `http://www.madess.cnr.it/Summary.htm`

task executions are perfectly nested: if task A preempts task B, it cannot happen that B executes again before the end of A. In this way, it is possible to use a single stack for all the execution frames of the tasks.

Next, comes the following observation: if task preemption is limited to occur only between selected task groups, it is possible to bound the maximum number of task frames concurrently active in the stack, therefore reducing the maximum requirement of RAM space for stack (which is the only way the OS can limit RAM requirements).

Although this idea is not new (see [18]), we extended it along many directions. More specifically, a complete methodology for minimizing the memory utilization of real-time task sets, communicating through shared memory, in uniprocessor and multiprocessor systems is presented in this paper. First, the uniprocessor case is considered, and the following results are presented: a novel scheduling algorithm, called **SRPT**, that allows the use of one single stack for all the real-time tasks under dynamic priority scheduling (Earliest Deadline) schemes; an optimization procedure for assigning the scheduling parameters (preemption thresholds and grouping of tasks in non-preemptive sets) so to minimize the maximum stack size without jeopardizing the schedulability of the task set.

Then, the previous results are extended to multiprocessor systems. In particular, we developed: a novel scheduling algorithm called **MSRP**, that allows real-time tasks, allocated on different processor, to communicate/interact through shared memory; each task is statically allocated to one processor, and all tasks on one processor share the same stack; an optimization procedure for assigning tasks to processors and for assigning the scheduling parameters, so to minimize the overall stack size.

The remaining sections are organized as follows. Section 2 presents some previous related work. Section 3 contains the definitions and the assumptions. Section 4 introduces the SRP and Preemption Thresholds mechanisms on which our work is based. Section 5 discusses our integration of SRP and Preemption thresholds on top of an EDF scheduler. Section 6 contains the discussion on how to optimize memory and CPU resources in uniprocessor systems. Section 7 discusses the MSRP Scheduling Algorithm. Section 8 contains the description of our Simulated Annealing approach to the task allocation problem. Section 9 ends the paper with the discussion on the experimental results for single and multiprocessor systems.

2. Related work

The idea of assigning each task a preemption threshold and to group tasks in non-preemptive sets has been formulated by Saksena and Wang [18]. The mechanism has been implemented (in a proprietary form) in the SSX kernel from REALOGY [6] and the ThreadX kernel from Express Logic [8].

The algorithms presented in this paper are based on the Stack Resource Policy (SRP), a synchronization protocol presented by Baker in [1]. The SRP is similar to the Priority Ceiling Protocol of Sha, Lehoczky and Rajkumar (see [19]), but has the additional property that a task is never blocked once it starts executing.

The problem of scheduling a set of real-time tasks with shared resources on a multiprocessor system is quite complex. One of the most common approaches is to statically allocate tasks to processors and to define an algorithm for inter-processor communication. Following this approach, the problem can be divided into two sub-problems: define a scheduling algorithm plus a synchronization protocol for global resources; and provide an off-line algorithm for allocating tasks to processors.

Solutions have been proposed in the literature for both sub-problems. The *Multiprocessor Priority Ceiling Protocol* (MPCP) has been proposed by Rajkumar in [17] for scheduling a set of real-time tasks with shared resource on a multi-processor. It extends the Priority Ceiling Protocol [19] for global resources. However, it is rather complex and does not guarantee that the execution of tasks will not be interleaved (tasks cannot share the same stack). Moreover, no allocation algorithm is proposed.

The problem of allocating a set of real-time tasks to m processors has been proved NP-hard in [12] and [7], even when tasks are considered independent. Several heuristic algorithms have been proposed in the literature [4, 16], but none of them explicitly considers tasks that interact through mutually exclusive resources.

In this paper, we bring contributions to both sub-problems. In Section 7, we propose an extension of the SRP protocol to multiprocessor systems. In Section 8 we propose a simulated annealing based algorithm for allocating tasks to processors.

3. Basic assumptions and terminology

Our system consists of a set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of real time tasks to be executed on a set $\mathcal{P} = \{P_1, \ldots, P_m\}$ of processors. First, we consider the case of a uniprocessor, and then extend the results to the case of multi-processor systems. The subset of tasks assigned to processor P_k will be denoted by $T_{P_k} \subset \mathcal{T}$. A real time task τ_i is a infinite sequence of jobs (or instances) $J_{i,j}$. Every job is characterized by a release time $r_{i,j}$, an execution time $c_{i,j}$ and a deadline $d_{i,j}$.

A task can be periodic or sporadic. A task is *periodic* if the release times of two consecutive jobs are

separated by a constant *period*; a task is *sporadic* when the release times of two consecutive job are separated by a variable time interval, with a lower bound, also called minimum interarrival time.

Without loss of generality, we use the same symbol θ_i to indicate the period of a periodic task and the minimum interarrival time of a sporadic task τ_i. In the following a task will be characterized by a worst case execution time $C_i = \max\{c_{i,j}\}$ and a period θ_i. We assume that the relative deadline of a task is equal to θ_i: thus, $d_{i,j} = r_{i,j} + \theta_i$.

Tasks can access mutually exclusive resources through critical sections. Let $\mathcal{R} = \{\rho^1, \ldots, \rho^p\}$ be the set of shared resources. The k–th critical section of task τ_i on resource ρ^j is denoted by ξ^j_{ik} and its maximum duration is denoted by ω^j_{ik}.

4. Background

4.1. Stack Resource Policy (SRP)

The Stack Resource Policy was proposed by Baker in [1] for scheduling a set of real-time tasks on a single processor. It can be used togheter with the Rate Monotonic (RM) scheduler or with the Earliest Deadline First (EDF) scheduler. According to the SRP, every real-time (periodic and sporadic) task τ_i is assigned a priority p_i and a static preemption level λ_i, such that the following essential property holds:
τ_i is not allowed to preempt τ_j, unless $\lambda_i > \lambda_j$.
Under EDF and RM, the previous property is verified if preemption levels are inversely proportional to the periods of tasks.
Every resource ρ^k is assigned a static[2] *ceiling* defined as: $\text{ceil}(\rho^k) = \max_i\{\lambda_i \mid \tau_i \text{ uses } \rho^k\}$. Finally, a dynamic *system ceiling* is defined as

$$\Pi_s(t) = \max[\{ceil(\rho^k) \mid \rho^k \textit{ is currently locked}\} \cup \{0\}].$$

Then, the SRP scheduling rule states that: *"a job is not allowed to start executing until its priority is the highest among the active jobs and its preemption level is greater than the system ceiling"*. The SRP ensures that once a job is started, it cannot be blocked until completion; it can only be preempted by higher priority jobs. However, the execution of a job $J_{i,k}$ with the highest priority in the system could be delayed by a lower priority job, which is locking some resource, and has raised the system ceiling to a value greater than or equal to the preemption level λ_i. This delay is called *blocking time* and denoted by B_i. Given the maximum blocking time for each task, it is possible to perform a schedulability test, depending on the scheduling algorithm.

In [1] Baker proposed the following schedulability condition for the EDF scheduler:

$$\forall i,\ 1 \le i \le n \quad \sum_{k=1}^{n} \frac{C_k}{\theta_k} + \frac{B_i}{\theta_i} \le 1 \qquad (1)$$

The maximum local blocking time for each task τ_i can be calculated as the longest critical section ξ^k_{jh} accessed by tasks with longer periods and with a ceiling greater than or equal to the preemption level of τ_i.

$$B_i = \max_{\tau_j \in \mathcal{T}, \forall h} \{\omega^k_{jh} \mid \lambda_i > \lambda_j \wedge \lambda_i \le \text{ceil}(\rho^k)\}. \qquad (2)$$

The Stack Resource Policy has several interesting properties. It prevents deadlock, bounds the maximum blocking times of tasks, reduces the number of context switches and can be easily extended to multi-unit resources. From an implementation viewpoint, it allows tasks to share a unique stack. In fact, a task never blocks its execution: it simply cannot start executing if its preemption level is not high enough. Moreover, the implementation of the SRP is straightforward as there is no need to implement waiting queues.

However, SRP does not scale to multiprocessor systems. In section 7 we will propose a possible extension of the SRP to be used in multi-processor systems.

4.2. Preemption Thresholds

Given a non-interleaved execution of the application tasks, the use of a preemptive scheduling algorithm makes the maximum number of task frames on the stack equal to the number of priority levels, whereas using a non-preemptive algorithm there can be only one frame on the stack. However, a non-preemptive algorithm in general is less responsive and could produce an infeasible schedule. Hence, the goal is to find an algorithm that selectively disables preemption in order to minimize the maximum stack size requirement while respecting the schedulability of the task set.

Based on this idea, Wang and Saksena, [18] developed the concept of *Preemption Threshold*: each task τ_i is assigned *a nominal priority* π_i and a preemption threshold γ_i with $\pi_i \le \gamma_i$. When the task is activated, it is inserted in the ready queue using the nominal priority; when the task begins execution, its priority is raised to its preemption threshold; in this way, all the tasks with priority less than or equal to the preemption threshold of the executing task cannot make preemption. According to [18], we introduce the following definitions:

Definition 1 *Two tasks τ_i and τ_j are* ***mutually non-preemptive*** *if $(\pi_i \le \gamma_j) \wedge (\pi_j \le \gamma_i)$.*

[2] In the case of multi-units resources, the ceiling of each resource is dynamic as it depends on the current number of free units.

Definition 2 *A set of tasks $G = \{\tau_1, \tau_2, \ldots, \tau_m\}$ is a* ***non-preemptive group*** *if, for every pair of tasks $\tau_j \in G$ and $\tau_k \in G$, τ_j and τ_k are mutually non–preemptive.*

By assigning each task the appropriate preemption threshold, we can reduce the number of preemptions in the system without jeopardizing the schedulability of the tasks set. Given an assignment of preemption thresholds, the task set can be partitioned into *non–preemptive groups*. Obviously, a small number of groups results in a lower requirement for the stack size.

In the following, we will show how it is possible to efficiently implement the Preemption Threshold mechanism using the SRP, and extend it to be used under EDF.

5. Integrating Preemption Threshold with the SRP

Our approach is based on the observation that the threshold values used in the Preemption Threshold mechanism are very similar to the resource ceilings of the SRP. In the SRP, when a task accesses a critical section, the system ceiling is raised to the maximum between the current system ceiling and the resource ceiling. In this way, an arriving task cannot preempt the executing task unless its preemption level is greater than the current system ceiling. This mechanism can be thought as another way of limiting preemptability.

Thus, if we want to make task τ_i and task τ_j mutually non-preemptive, we can let them share a *pseudo-resource* ρ^k: the ceiling of resource ρ^k is the maximum between the preemption levels of τ_i and τ_j. At run time, instances of τ_i or τ_j will lock ρ^k when they start executing and hold the lock until they finish.

Suppose task τ_i needs a set of pseudo-resources $\rho^1, \ldots, \rho^h$. When τ_i starts execution, it locks all of them: in the SRP, this corresponds to raising the system ceiling to $\max_k \text{ceil}(\rho^k)$. We define this value as the *preemption threshold* γ_i of task τ_i. Now, the problem of finding an optimal assignment of thresholds to tasks is equivalent to finding the set of pseudo-resources for each task. In the remaining of this paper, we will indicate this modification of the SRP as *SRPT* (SRP with Thresholds).

Since SRPT can be thought as an extension of the SRP that add pseudo-resources compatibles with the traditional SRP resources, it can be easily shown that SRPT retains all the properties of SRP.

The feasibility test for SRPT is given by Equation (1), except for the computation of the blocking time, that is: $B_i = max(B_i^{local}, B_i^{pseudo})$, where B_i^{local} and B_i^{pseudo} are respectively the blocking time due to local resources and the blocking time due to pseudo-resources.

Blocking due to local resources. Assuming relative deadlines equal to periods, the maximum local blocking time for each task τ_i can be calculated using Equation (2). This can be easily proved: supposing the absence of pseudo-resources, the SRPT reduces to the SRP, and the blocking times can be calculated using equation 2.

Blocking due to pseudo-resources. A task τ_i may experience an additional blocking time due to the non-preemptability of lower priority tasks. This blocking time can be computed as follows:

$$B_i^{pseudo} = \max_{\tau_j \in T_{P_i}} \{C_j \,|\, \lambda_i > \lambda_j \wedge \lambda_i \leq \gamma_j\}$$

The non-preemptability of lower task is due to the use of pseudo-resources. The formula of B_i^{pseudo} is another way of writing formula 2, because: γ_i is $\max_k ceil(\rho^k) = ceil(\rho^{k'})$ where $k' \in \{k : \gamma_i = ceil(\rho^k)\}$ and C_i is the critical section duration for resource k' (remember that pseudo-resources are locked when an instance starts and is unlocked when an instance finish; moreover, we can consider only the k' critical section for each task since they all have length equal to C_i and $\forall\, k, ceil(\rho^k) \leq ceil(\rho^{k'}) = \gamma_i$.

The SRPT presents two main advantages: it seamlessly integrates access to mutually exclusive resources and preemption threshold with a very little implementation effort and with no additional overhead, and it permits to implement the preemption threshold mechanism on top of EDF. The last issue can lead to further optimizations: the EDF scheduling algorithm has been proven optimal both in the preemptive [14, 2, 3] and in the non-preemptive[3] version [11]; furthermore, in [13] the authors claim that EDF+SRP is an optimal algorithm for scheduling sporadic task sets with shared resources. Since EDF is optimal, it is more likely that a given assignment of preemption thresholds produces a feasible schedule. Therefore, we expect a better chance to trade processor utilization with a reduction in the maximum stack space requirement by reducing preemption.

6. Optimizing stack usage in Uniprocessors

In this section we present an algorithm that allows the optimization of the total stack space requirement of a set of tasks using the SRPT protocol on uniprocessor systems. To simplify the presentation, we do not consider here the use of shared resources. The complete algorithm for multiprocessors will be presented in Section 8.

[3] The non-preemptive version of the EDF algorithm is optimal for sporadic task sets among all the non-idle (work conserving) non-preemptive scheduling algorithms.

The algorithm requires each task to be characterized by its worst case execution time C_i, its period θ_i, its maximum stack requirement (in bytes) s_i, its priority π_i and its preemption level λ_i. At the end of the optimization algorithm, each task τ_i will be assigned a preemption threshold γ_i and will be inserted in a non-preemptive group G_k. The goal of the optimization algorithm is:
step 1 to find an optimal assignment of preemption thresholds to tasks, and **step 2** to find an optimal set of non-preemptive groups that minimizes the total stack size, maintaining the feasibility of the schedule.

The algorithm selects a possible assignment of preemption thresholds and tests the feasibility of the scheduling using Equation (1). Our optimization algorithm works as follows: tasks are ordered by decreasing preemption level λ; we use the algorithm described in [18] to explore the space of possible threshold assignments[4]: starting with the task having the highest preemption level, we try to raise the preemption threshold γ of each task, until the task set remains schedulable according to Equation (1). Then, given a feasible assignment of preemption thresholds, we partition the task set into non-preemptive groups and compute the maximum stack size. Our algorithm differs from the one in [18] in the final optimization objective: while the algorithm in [18] tries to minimize the number of non-preemptive groups, our algorithm accounts for the stack usage of each task and tries to minimize the total amount of required stack. In fact, there are cases in which the minimum overall stack requirement does not correspond to the minimum number of groups.

The algorithm that is used to *partition the task set into preemption groups* is more complex and can be only outlined as follows:
Step 1: Tasks are ordered by increasing preemption thresholds; ties are broken in order of decreasing stack requirements.
Step 2: The algorithm starts by finding the *maximal group* for each task. A maximal group for task τ_i is the biggest non-preemptive group that can be created using τ_i as a *representative task*. A *representative task* for a non-preemptive group is the task having the smallest threshold among all tasks in the group. Maximal groups are computed with the algorithm shown in Figure 2.
Step 3: Then, the algorithm calls a recursive function that allocates all tasks to non-preemptive groups using the information computed in the previous step. The function, called **create_group()**, recursively computes all solutions consisting in the partitioning of tasks into a set of non-preemptive groups G_i.

The function **create_group(g, min_stack, sum)** is the core of the procedure. Its pseudo code description is outlined in Figure 3. At this point each task is assigned a new index which corresponds to its position in the order of preemption thresholds (starting from 0).

[4]Since EDF is optimal, there is no need to find an initial priority assignment for the task set.

```
foreach τ_i in T {
  M_i = emptylist;
  foreach τ_j in {τ_k : τ_k ∈ T and k > i}
    if (λ_j ≤ γ_i)  insert(M_i, j);
}
```

Figure 2. Finding the maximal groups.

When called, the function computes a set of new groups starting from G_g where `g` is the index of the group's representative task; `min_stack` points to the current minimum for the overall stack requirements and `sum` to the (partial) stack requirement for the solution being computed.

The first time the function is called, `g` has the value `0` (the algorithm starts from the task with the lowest threshold), `min_stack` refers a variable containing the sum $\Sigma_i s_i$ of all stack requirements (the worst case stack requirement), and `sum` equals 0 (no task allocated to any group yet). No group G_i has been computed yet.

Lines 9-31 are used to selectively extract a subset of M_g that will be inserted into G_g for testing the optimality of a solution. The subsets that are tried as candidate for G_g are *all* the possible subset of M_g, plus the representative task τ_g.

Please also note that in the function **creategroup()** the index `i` **always** refers to the position in list M_g rather than G_g.

At line 7 the group G_g is initialized (the representative task τ_g is inserted in its group). The `i` variable (initialized at the index of the first element in M_g line 8) is used to mark the next index of the candidate representative for a new group.

Lines 10-12 insert all the unallocated tasks belonging to M_g and following τ_i into the group G_g. Line 13 computes the maximum stack requirement of the non-preemptive group G_g, and line 14 adds it to the temporary accumulator `new_sum`.

If there remains a task to be allocated, line 16 finds a task that will be the representative for the next group, and line 17 calls recursively the **creategroup()** function with `g` set to that task index, and `sum` set to the current stack usage.

The rest of the function implements the cleanup before a new iteration or backtracking for searching a new solution. This means removing some tasks from the current group and setting up new representative tasks for a different group partitioning. If the current group G_g is not composed by its representative task only, line 19 removes all tasks from the tail of G_g until the stack requirement of the group decreases (the task

with the largest stack requirement is removed). Next, the pointer i is set to the position of the task following the last task extracted from G_g (line 21) to allow task τ_i to be skipped at the next iteration, and to become a representative task in the following recursive call. If all tasks have been assigned to a group then a new candidate solution has been computed and must be evaluated as a candidate optimum (lines 24-26). If the current group is the last group, it is emptied (line 28), since it is pointless to split it.

```
 1: creategroup(int g, int *min_stack, int sum)
 2: {
 3:   task_index i;
 4:   int m, newsum;
 5:   bool notYetDone = true;
 6:
 7:   initialize(G_g);
 8:   i = firstelement(M_g);
 9:   do {
10:     foreach j in {k : k ≥ i in list M_g}
11:       if (τ_j not already allocated)
12:         insertlast(G_g,τ_j);
13:     m = findMaximumStackUsage(G_g);
14:     newsum = sum + m;
15:     if (there are task to be allocated) {
16:       f = findFirstFreeTask(g);
17:       creategroup(f, min_stack, newsum);
18:       if (G_g != {τ_g}) {
19:         i = lastelement(G_g);
20:         removeFromTail(G_g);
21:         i = nextelement(M_g,i);
22:       } else notYetDone = false;
23:     } else
24:       if (newsum < *min_stack) {
25:         NewOptimumFound();
26:         *min_stack = newsum;
27:       }
28:       Remove_all_tasks(G_g);
29:       notYetDone = false;
30:     }
31:   } while (notYetDone);
32:   remove(G_g);
33: }
```

Figure 3. The create_group() recursive function.

The implementation of the algorithm is slightly more complex, since a lot of effort is spent in order to optimize the search by pruning the search tree and back-tracking before reaching a higher cost solution.

In the worst case, the complexity of the algorithm is exponential in the number of tasks. However, since the number of groups in the optimal solution is small, the number of combinations to evaluate is limited. Thanks to the efficiency of the pruning, the number of solutions is further reduced. In our experiments, the average number of explored solutions (leafs) is quite low even for large task sets (<160). For typical embedded systems, where the number of tasks rarely exceeds 20, the problem is tractable with modern computers.

7. Sharing Resources in Multiprocessors

When considering **multiprocessor** symmetric architectures, we wish to keep the nice properties of EDF and SRP, that is high processor utilization, predictability and perfectly nested task executions on local processors. Unfortunately, the SRP cannot be directly applied to multiprocessor systems.

In this section, we first propose an extension of the SRP protocol to multi-processor systems and a schedulability analysis for the new policy. In the next section, we propose a simulated annealing based algorithm for allocating tasks to processors that minimizes the overall memory requirements.

7.1. Multiprocessor Stack Resource Policy

Suppose that tasks have already been allocated to processors. Depending on this allocation, resources can be divided in *local* and *global* resources. A local resource is used only by tasks belonging to the same processor, whereas a global resource is used by task belonging to different processors.

We concentrate our efforts on the policy for accessing global resources. If a task tries to access a global resource and the resource is already locked by some other task on another processor, there are two possibilities: the task is suspended (as in the MPCP algorithm), or the task performs a busy wait (also called *spin lock*). We want to maintain the properties of the SRP: in particular, we want to let all tasks belonging to a processor to share the same stack. Hence, we choose the second solution. However, the *spin lock time* is wasted time and should be reduced as much as possible (the resource should be freed as soon as possible). For this reason, when a task executes a critical section on a global resource, its priority is raised to the maximum priority on that processor and the critical section becomes non-preemptable.

In order to simplify the implementation of the algorithm, the amount of information shared between processors is minimal. For this reason, the priority assigned to a task when accessing resources does not depend on the status of the tasks on other processors or on their priority. The only global information is the status of the global resources.

The MSRP algorithm works as follows:

Rule 1: For local resources, the algorithm is the same as the SRP algorithm. In particular, we define a preemption level for every task, a ceiling for every local resource, and a system ceiling Π_k for every processor P_k.

Rule 2: Tasks are allowed to access local resource through nested critical sections. It is possible to nest

	C_i	λ_i	ω_{ij}^1	ω_{ij}^2	ts_i	C_i'	B_i^l	B_i^g
τ_1	2	3	0	0	0	2	0	7
τ_2	6	2	2	0	0	6	9	7
τ_3	11	1	9	4	3	14	0	0
τ_4	7	1	0	3	4	11	0	0
τ_5	2	2	0	0	0	2	0	7

Table 1. The example task set.

local and global resources. However, it is not possible to nest global critical sections, otherwise a deadlock can occur.

Rule 3: For each global resource, every processor P_k defines a ceiling greater than or equal to the maximum preemption level of the tasks on P_k.

Rule 4: When a task τ_i, allocated to processor P_k accesses a global resource ρ^j, the system ceiling Π_k is raised to ceil(ρ^j) making the task non–preemptable. Then, the task checks if the resource is free: in this case, it locks the resource and executes the critical section. Otherwise, the task is inserted in a FCFS queue on the global resource, and then performs a busy wait.

Rule 5: When a task τ_i, allocated to processor P_k, releases a global resource ρ^j, the algorithm checks the corresponding FCFS queue, and, in case some other task τ_j is waiting, it grants access to the resource, otherwise the resource is unlocked. Then, the system ceiling Π_k is restored to the previous value.

Example. Consider a system consisting of two processors and five tasks as shown in Figure 4. Tasks τ_1, τ_2 and τ_3 are allocated to processor P_1: task τ_3 uses local resource ρ^1, task τ_2 uses resources ρ^1 and ρ^2 through nested critical sections, and τ_1 does not use any resource. Tasks τ_4 and τ_5 are allocated to processor P_2: task τ_4 uses the global resource ρ^1 and τ_5 does not uses resources. The parameters of the tasks are reported in Table 1. The ceiling for resource ρ^1 is 2. The ceiling for resource ρ^2 on processor P_1 is 3, and on processor P_2 is 2. A possible schedule is shown in Figure 5. At time $t = 3$, task τ_2 is blocked because its preemption level $\lambda_2 = 2$ is equal to the current system ceiling $\Pi_1 = 2$ on processor P_1. At time $t = 5$, task τ_3 locks resource ρ_2 and raises the system ceiling Π_1 to 3. At time $t = 6$, task τ_4 tries to access the global resource ρ^2 which is currently locked by τ_2. Thus, it raises the system ceiling of processor P_2 to 2 and performs a busy wait. At time $t = 7$, both τ_1 and τ_5 are blocked, because the system ceilings of the two processors are set to the maximum. At time $t = 8$, task τ_3 releases the global resource ρ^2 and task τ_4 can enter the critical section on ρ^2. At the same time, the system ceiling of processor P_1 is set back to 2, and task τ_1 can make preemption.

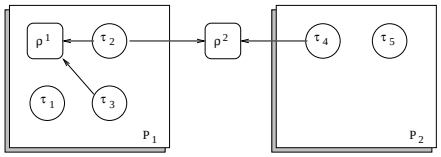

Figure 4. Structure of the example.

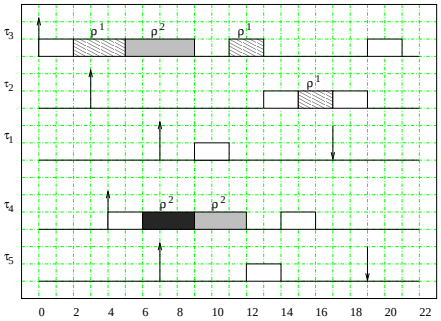

Figure 5. Example of schedule produced by the MSRP on two processors.

7.2. Schedulability analysis of the MSRP

First, we give an upper bound on the time that task τ_i, allocated to processor P_k, can spend waiting for a global resource ρ^j. In the following, we refer to this time as *spin lock time* and denote it as $spin(\rho^j, P_k)$.

Lemma 1 *The spin lock time that every task allocated to processor P_k needs to spend for accessing a global resource $\rho^j \in \mathcal{R}$ is bounded from above by:*

$$spin(\rho^j, P_k) = \sum_{p \in \{\mathcal{P} - P_k\}} \max_{\tau_i \in T_p, \forall\, h} \omega_{ih}^j.$$

Basically, the spin lock time increments the duration ω_{ih}^j of every global critical section ξ_{ih}^j, and, consequently, the worst case execution time C_i of τ_i. Moreover, it also increments the blocking time of the tasks allocated to the same processor with a preemption level greater than λ_i.

We define $totalspin_i$ as the maximum total spin lock time experienced by task τ_i. From the previous lemma,

$$totalspin_i = \sum_{\xi_{ih}^j} spin(\rho^j, P_i)$$

We also define the *actual worst case computation time* C_i' for task τ_i as the worst case computation time plus the total spin lock time:

$$C_i' = C_i + totalspin_i$$

Now, we demonstrate that the MSRP maintains the same basic properties of the SRP, as shown by the following theorems.

Theorem 1 *Once a job starts executing it cannot be blocked, but only preempted by higher priority jobs.*

Note that a job can be delayed before starting execution by the fact that the system ceiling is greater than or equal to its preemption level. This delay is called *blocking time*. The following theorem gives an upper bound to the blocking time of a task.

Theorem 2 *A job can experience a blocking time at most equal to the duration of one critical section (plus the spin lock time, if the resource is global) of a task with lower preemption level.*

It is noteworthy that the execution of all the tasks allocated on a processor is perfectly nested (because once a task starts executing it cannot be blocked), therefore all tasks can share the same stack.

For simplicity, the blocking time for a task can be divided into blocking time due to local and global resources. In addition, if we consider also the preemption threshold mechanism, we have to take into account also the blocking time due to the pseudo-resources:

$$B_i = max(B_i^{local}, B_i^{global}, B_i^{pseudo})$$

where B_i^{local}, B_i^{global} and B_i^{pseudo} are:

Blocking time due to local resources: This blocking time is equal to the longest critical section ξ_{jh}^k among those (of a task τ_j) with a ceiling greater than or equal to the preemption level of τ_i:

$$\begin{aligned} B_i^{local} &= \max_{j,h,k}\{\omega_{jh}^k \mid (\tau_j \in T_{P_i}) \wedge (\rho^k \text{ local to } P_i) \\ &\quad \wedge (\lambda_i > \lambda_j) \wedge (\lambda_i \leq ceil(\rho^k))\} \end{aligned}$$

Blocking time due to global resources: Assume the task τ_i, assigned to processor P_i, is blocked by a task τ_j ($\lambda_j < \lambda_i$) which is assigned to the same processor P_i, and which is waiting for, or it is inside to, a global critical section ξ_{jh}^k. In this case, the blocking time for task τ_i is,

$$\begin{aligned} B_i^{global} &= \max_{j,h,k}\{\omega_{jh}^k + spin(\rho^k, P_i) \mid (\tau_j \in T_{P_i}) \\ &\quad \wedge (\rho^k \text{ global}) \wedge (\lambda_i > \lambda_j)\} \end{aligned}$$

Blocking time due to pseudo resources: As explained in the previous sections, this blocking time is due to the fact that a task τ_i can be mutually non–preemptive with other tasks on the same processor: here, the only difference with the SRPT is that we have to consider the *actual worst case execution time* instead of the worst case execution time.

$$B_i^{pseudo} = \max_{\tau_j \in T_{P_i}} \{C'_j \mid \lambda_i > \lambda_j \wedge \lambda_i \leq \gamma_j\}$$

Theorem 3 *Suppose that tasks on processor P_k are ordered by decreasing preemption level. The schedulability test is as follows:*

$$\forall\, P_k \in \mathcal{P} \qquad T_{P_k} = \{\tau_1, \ldots, \tau_{n_k}\} \;\; \forall i = 1, \ldots, n_k$$
$$\sum_{l=1}^{i} \frac{C'_l}{\theta_l} + \frac{B_i}{\theta_i} \leq 1$$

Please note that the blocking factor influences only one element of the guarantee formula, whereas the spin lock time influences both the blocking time and the worst case execution time. This implies that, when designing an allocation algorithm, one of the goals is to reduce the spin lock time as much as possible. Another noteworthy observation is that, using the MSRP, each processor works almost independently from the others. In particular, it is possible to easily apply this algorithm to non–homogeneous multiprocessor systems. For the task set of the previous example, the total spin lock time ts_i, the actual worst case execution time C'_i, the local and global blocking times are reported in Table 1. The MSRP has many advantages over the MPCP. Unlike MPCP, with the MSRP it is possible to use one single stack for all the tasks allocated to the same processor. Moreover, the MPCP is more complex and difficult to implement than the MSRP. In fact, the MSRP does not need semaphores or blocking queues for local resources, whereas global resources need only a FIFO queue (an efficient implementation can be found in [5]. Finally, the MSRP, like the SRP, tends to reduce the number of preemptions in the systems, hence there is less overhead. However, this comes at the cost of a potentially dangerous spin lock time.

8. Optimizing stack usage in Multiprocessors

Given a task allocation, the policies and algorithms presented in this paper allow to search for the optimal assignment of preemption thresholds to tasks and to selectively group tasks in order to reduce RAM consumption. However, the final outcome depends on the quality of the decisions taken in the task allocation phase. Moving one task from one processor to another can change the placement of (some of) the shared resources accessed by it (some global resources become local and vice versa) and the final composition of the non–preemptive groups on each processor. Unfortunately, the task allocation problem has exponential complexity even if we limit ourselves to the simple case of deadline-constrained scheduling.

A simulated annealing algorithm is a well-known solution approach to this class of problems. Simulated annealing techniques (SA for short) have been used in

[20, 18] to find the optimal processor binding for real-time tasks to be scheduled according to fixed-priority policies, in [15] to solve the problem of scheduling with minimum jitter in complex distributed systems and in [18] to assign preemption thresholds when scheduling real-time tasks with fixed priorities on a uniprocessor. In the following we show how to transform the allocation and scheduling problem which is the subject of this paper into a form that is amenable to the application of simulated annealing. Our solution space S consists of all possible assignments of tasks to processors. We are interested in those task assignments that produce a feasible schedule and, among those, we seek the assignment that has minimum RAM requirements. Therefore we need to define an objective function to be minimized and the space over which the function is defined.

The SA algorithm searches the solution space for the optimal solution as follows: a transition function TR is defined between any pair of task allocation solutions $(A_i, A_j) \in S$ and a neighborhood structure S_i is defined for each solution A_i containing all the solutions that are reachable from A_i by means of TR. A starting solution A_0 is defined and its cost (the value of the objective function) is evaluated. The algorithm randomly selects a neighbor solution and evaluates its cost. If the new solution has lower cost, then it is accepted as the current solution. If it has higher cost, then it is accepted with a probability exponentially decreasing with the cost difference and slowly lowered with time according to a parameter which is called temperature.

Our transition function consists in the random selection of a number of tasks and in changing the binding of the selected tasks to randomly selected processors. This simple function allows to generate new solutions (bindings) at each round starting from a selected solution. Some of the solutions generated in this way may be non schedulable, and therefore should be eventually rejected. Unfortunately, if non–schedulable solutions are rejected before the optimization procedure is finished, there is no guarantee that our transition function can approach a global optimum. In fact, it is possible that every possible path from the starting solution to the optimal solution requires going through intermediate non-schedulable solutions.

If non-schedulable solutions are acceptable as intermediate steps, then they should be evaluated very poorly. Therefore, we define a cost function with the following properties:

1) schedulable solutions must always have energy lower than non–schedulable solutions;

2) the energy of non schedulable solutions must be proportional to the maximum excess utilization resulting from the evaluation of formula (3) for non–schedulable tasks;

3) the energy of schedulable solution must be proportional to the worst case overall RAM requirements for stack usage.

If TS is the overall stack requirement, obtained by adding up the stack requirements of all tasks, and OS is the overall stack requirement, evaluated for schedulable sets after the computation of optimal preemption thresholds and task groups (see Section 6), then our cost function is the following:

$$\begin{cases} \max_{\forall \tau_i} \left(\sum_{k=i}^{n} \frac{C'_k}{T_k} + \frac{B_i}{T_i} \right) * TS & \text{non sched. assign.} \\ TS + \Delta * (OS - TS) & \text{sched. assign.} \end{cases}$$

When the assignment is non schedulable, we use the result of the guarantee test (Equation 1) as an index of schedulability. In fact, as the system load, blocking time or spin-lock time increase, the system becomes *less* schedulable. When the assignment is schedulable, the cost function does not depend on processor load but returns a value that is proportional to the reduction of stack with respect to the total stack requirement.

The Δ factor estimates the average ratio between the stack requirements before task grouping and the stack requirements after optimization and is defined as:

$$\Delta = \frac{ncpu * meanstack * meangroups}{ntask * meanstack}$$

where *ncpu* is the number of CPU in the system, *meanstack* is the mean stack value of all tasks, *meangroups* estimates the typical number of preemption groups on a uniprocessor. The Δ factor has been introduced to smooth the steep increase in the cost function when going from schedulable solutions to non–schedulable assignments. This improves the chances for the simulated annealing algorithm to escape from local minima (which might require accepting a non–schedulable solution).

The experimental results (next section) show the effectiveness of our SA-based binding algorithm when simulating task sets scheduled on 4-processor system-on-a-chip architectures.

9. Experimental evaluation

We extensively evaluated the performance of our optimization algorithms on a wide range of task set configurations.

Uniprocessor experiments In every experiment, tasks' periods are randomly chosen between 2 and 100. The total system load U ranges from 0.5 to 0.99, with a step of 0.01: the worst case execution time of every task is randomly chosen such that the utilization factors sums up to U. The number of tasks in the task set ranges from 1 to 100, and the stack frame size is a random variable chosen between 10 and 100 bytes excepts for the experiments of Figure 7 in which the

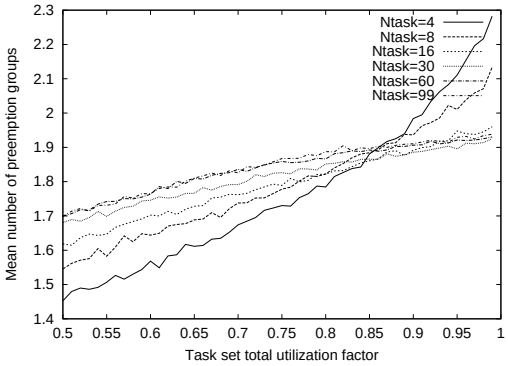

Figure 6. Average number of preemption groups for different task set sizes.

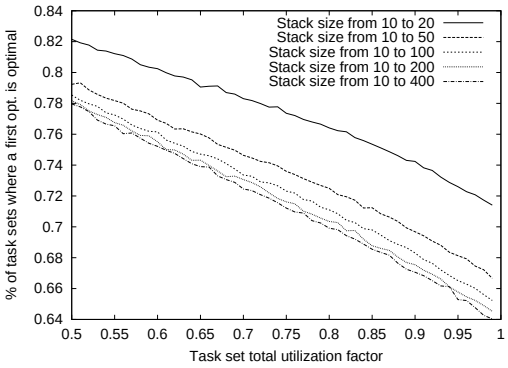

Figure 7. Ratio of improvement given by our optimization algorithm.

stack size ranges between 10 and 400 bytes. In Figure 6 the average number of preemption groups is shown.

Note that the figure has a maximum for NTASK = 4 and U=0.99. As the number of tasks increases, the number of preemption groups tends to 2; this can be explained with the fact that, when the number of tasks grows, each task has a smaller worst case execution time; hence, the schedule produced by a non-preemptive scheduler does not differ significantly from the schedule produced by a preemptive scheduler. On the contrary, with a small number of tasks, the worst case execution time of each task is comparable with the period; hence it is more difficult to find a feasible non-preemptive schedule.

Figure 6 shows how the average number of preemption groups is almost independent of the utilization factor and of the number of tasks, except for a very limited number of tasks (< 10) and a high utilization factor (> 0.8).

The average number of groups is not only constant but also very close to 2. This means that the application of Preemption Threshold techniques, together with EDF, allows a great reduction in the number of preemption levels and great savings in the amount of RAM needed for saving the task stack frames. RAM reduction in the order of 3 to 16 times less the original requirements can easily be obtained.

In Figure 7, we compare the optimization algorithm presented in [18] (which does not take into account the stack frame size of the tasks) and our algorithm, to show the improvement in the optimization results. The figure shows the fraction of experiments where the optimal solution has been found by the original algorithm. The ratio appears as a function of the system load and for different stack sizes. In most cases (from 60% to 80%), the algorithm proposed in [18] finds the optimal partition of the task set in preemption groups. This ratio decreases as the load increases and as the range of the stack size requirements is widened.

Multiprocessor experiments. In the first set of experiments, we consider 4 CPU, 40 resources, and 40 tasks. Tasks' periods are randomly chosen between 1 and 1000. The total system load U ranges from 2.76 to 3.96, with a step of 0.2. The stack frame size of each task is a random variable chosen between 10 and 100 bytes. Each task has 0 to 4 critical sections that lock randomly selected resources; the sum of the worst case execution times of the critical section accessed by each single task is in the range of 0-20%, 5-25%, 10-30%,15-35%, 20-40% of the task worst case execution time (depending on the simulation, see Figure 7).

In Figure 8 we plot the stack gain ratio between the overall stack requirement before optimization and the stack memory requirement of the solution found by our SA algorithm. In all experimental runs the solution found by our SA routine saves a considerable amount of RAM even when compared to the first schedulable (and optimized for RAM consumption) solution found. The average improvement in 58 runs is 34.6% (min 18%, max 49%).

Running times can be a concern when using a simulated annealing solution. Our algorithm can be run in a few hours on modern computers The execution of the simulated annealing routine takes 6 to 30 hours on an Intel Pentium III 700Mhz to complete the cooling. For example, a typical execution (Total U = 2.76, critical section ratio 0.10 to 0.30) visited 15,900,000 assignments (one every 4 ms) and found 6,855,560 schedulable solutions. These results are quite acceptable considered that task allocation is a typical design time activity.

10. Conclusions and future works

In this paper, we present a solution for scheduling real-time tasks in single and multiple processor systems with minimal RAM requirements. In uniproces-

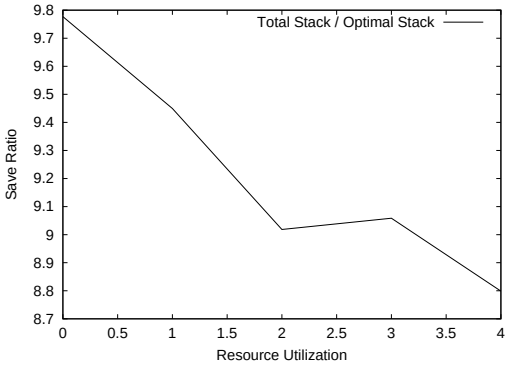

Figure 8. Ratio of improvement given by our multiprocessor optimization algorithm.

sor systems, our solution seamlessly integrates Earliest Deadline scheduling techniques, the Stack Resource Policy for accessing shared resources, plus an innovative algorithm for the assignment of preemption thresholds and the grouping of tasks in non-preemptive sets. Our methodology allows to evaluate the schedulability of task sets and to find the schedulable solution (the task groups) that minimize the RAM requirements for stack. We also provide an extension of the SRP policy to multiprocessor systems and global shared resources (MSRP) and a task allocation algorithm based on simulated annealing. The main contribution of our work consists in realizing that real-time schedulability and the minimization of the required RAM space are tightly coupled problems and can be efficiently solved only by devising innovative solutions. The objective of RAM minimization guides the selection of all scheduling parameters and is a factor in all our algorithms. We plan to implement the algorithms described in this paper in a new version of our ERIKA kernel[5] for the JANUS architecture [9].

References

[1] T.P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[2] S.K. Baruah, A.K. Mok, and L.E. Rosier. Preemptively scheduling hard-real-time sporadic tasks on one processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, pages 182–190, December 1990.

[3] S.K. Baruah, L.E. Rosier, and R.R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic real-time tasks on one processor. *The Journal of Real-Time Systems*, 2, 1990.

[4] A. Burchard, J. Liebeherr, Y. Oh, and S.H. Son. New strategies for assigning real-time tasks to multiprocessor systems. *IEEE Transactions on Computers*, 1995.

[5] T. S. Craig. Queuing spin lock algorithms to support timing predictability. In *Proceedings of the IEEE Real-Time Systems Symposium*, Dec. 1993.

[5] `http://erika.sssup.it/`

[6] Robert Davis, Nick Merriam, and Nigel Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings of the Work in Progress and Industrial Experience Session, Euromicro Conference on Real-Time Systems*, June 2000.

[7] M. L. Dertouzos and Aloysius Ka-Lau Mok. Multiprocessor on-line scheduling of hard-real-time tasks. *IEEE Transactions on software engineering*, 15(12), December 1989.

[8] Inc. Express Logic. http://www.threadx.com. available on Internet.

[9] A. Ferrari, S. Garue, M Peri, S. Pezzini, L.Valsecchi, F. Andretta, and W. Nesci. The design and implementation of a dual-core platform for power-train systems. In *Convergence 2000*, Detroit (MI), USA, October 2000.

[10] Paolo Gai, Giuseppe Lipari, Luca Abeni, Marco di Natale, and Enrico Bini. Architecture for a portable open source real-time kernel environment. In *Proceedings of the Second Real-Time Linux Workshop and Hand's on Real-Time Linux Tutorial*, November 2000.

[11] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 129–139, December 1991.

[12] J. Y. T. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic, real-time tasks. *Performance Evaluation*, 2:237–250, 1982.

[13] Giuseppe Lipari and Giorgio Buttazzo. Schedulability analysis of periodic and aperiodic tasks with resource constraints. *Journal of Systems Architecture*, 46:327–338, 2000.

[14] C.L. Liu and J.W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the Association for Computing Machinery*, 20(1), 1973.

[15] M. Di Natale and J. Stankovic. Scheduling distributed real-time tasks with minimum jitter. *Transaction on Computer*, 49(4), 2000.

[16] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Journal on Real Time Systems*, 9, 1995.

[17] R. Rajkumar. Synchronization in multiple processor systems. In *Synchronization in Real-Time Systems: A Priority Inheritance Approach*. Kluwer Academic Publishers, 1991.

[18] Manas Saksena and Yun Wang. Scalable real-time system design using preemption thresholds. In *Proceedings of the Real Time Systems Symposium*, December 2000.

[19] Lui Sha, Ragunathan Rajkumar, and John P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE transaction on computers*, 39(9), September 1990.

[20] K. Tindell, A. Burns, and A. Wellings. Allocating real-time tasks (an np-hard problem made easy). *Real-Time Systems Journal*, 1992.