Design Automation for Embedded Systems, 7, 53–87, 2002.

# Stack Size Minimization for Embedded Real-Time Systems-on-a-Chip*

PAOLO GAI pj@sssup.it
*ReTiS Lab, Scuola Superiore di Studi e Perfezionamento S. Anna - Pisa*

GIUSEPPE LIPARI lipari@sssup.it
*ReTiS Lab, Scuola Superiore di Studi e Perfezionamento S. Anna - Pisa*

MARCO DI NATALE marco@sssup.it
*ReTiS Lab, Scuola Superiore di Studi e Perfezionamento S. Anna - Pisa*

**Abstract.** The primary goal for real-time kernel software for single and multiple-processor on a chip systems is to support the design of timely and cost effective systems. The kernel must provide time guarantees, in order to predict the timely behavior of the application, an extremely fast response time, in order not to waste computing power outside of the application cycles and save as much RAM space as possible in order to reduce the overall cost of the chip. 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 small 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 on 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 innovative scheduling and optimization algorithms that effectively solve the problem of guaranteeing schedulability with an extremely little operating system overhead and minimizing RAM usage. We developed a fast and simple algorithm for sharing resources in multiprocessor systems, together with an innovative procedure for assigning a preemption threshold to tasks. These allow the use of a single user stack. The experimental part shows the effectiveness of a simulated annealing-based tool that allows to find a schedulable system configuration starting from the selection of 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.

**Keywords:** Multiprocessor scheduling, operating systems, real-time, stack size minimization.

## 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 [10] 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.

This work was supported in part by the CNR MADESS II Project.

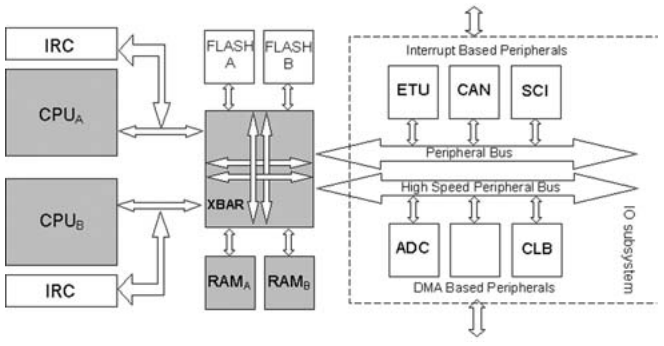


*Figure 1.* The Janus Dual Processor system.

There are two possible ways to increase the computational power in real-time systems: 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: caching makes it very hard or impossible to determine the worst case execution times of programs; deep pipelining is not effective because of the large number of stalls caused by reactions to asynchronous events. Parallelism at the instruction level (VLIW architectures) requires large silicon areas and drastically increases code size. Therefore, the best option and the future of many embedded applications seem to rely on the adoption of multiple-processor-on-a-chip architectures.

The Janus system, (see the scheme of Figure 1) developed by ST Microelectronics in cooperation with Parades [10], 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 running on the new single-chip platform require predictable and fast real-time kernel mechanisms. Furthermore, they 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. RAM is extremely expensive in terms of chip space and impacts heavily on the final cost (it is often necessary to re-design part of the application just to save a few RAM bytes).

In the design of the kernel mechanisms for the ERIKA kernel [11], 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 can increase the processor utilization.

For this reason, we found very important to exploit the different combinations and configurations of scheduling algorithms and services and to develop new ones in order to find the best kernel mechanisms for minimizing the memory requirements without jeopardizing the timing constraints.

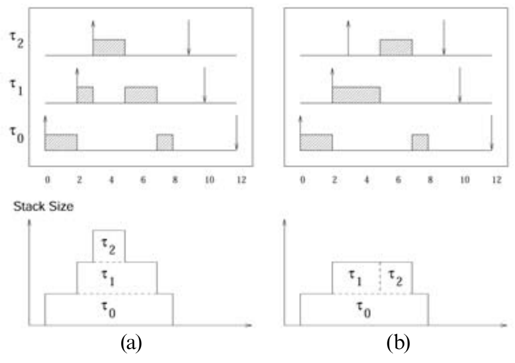


*Figure 2.* Two different schedules for the same task set: (a) full-preemptive schedule; (b) preemption is disabled between $\tau_1$ and $\tau_2$.

The overall RAM requirement is the sum of many factors, global variables, dynamically allocated memory (heap memory), operating system variables and stack space requirements for local variables and function call parameters. Of all these factors, the stack space requirements can be a significant factor and the only one that can be possibly reduced by a selection of operating system-level policies (dynamic memory allocation should not be used in hard real-time systems and is explicitly forbidden by the OSEK automotive standards).

The idea behind this work is based on the concept of non-interleaved execution. As explained in Section 4.1, using a protocol called Stack Resource Policy (SRP) [2], 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. An example of this behavior is depicted in Figure 2(a) where three periodic tasks $\tau_0$, $\tau_1$ and $\tau_2$ are scheduled by SRP. In the upper part of the figure, the ascending arrows denote the task activations, whereas the descending arrows denote the task deadlines. If the executions of tasks A and B were to be interleaved (for example scheduling the tasks in the order B, A, B, A) then, upon resuming its execution, task B would find the top of the stack occupied by the stack frame of task A. Therefore, a single stack cannot be used when executions are interleaved. In the lower part, the system stack size is plotted against the time.

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). In the example of Figure 2(b), preemption is disabled between $\tau_2$ and $\tau_1$ and, consequently, only two task frames can be active at the same time: thus we can decrease the amount of memory to be reserved for the system stack.

Although this idea is not new (see [25], [22]), we extended it along many directions. First, we consider dynamic priority scheduling instead of fixed priority. In particular, the algorithms presented in this paper are based on the Earliest Deadline First (EDF) scheduling algorithm, which achieves better processor utilization with respect to fixed priority schedulers. Second, the objective of our algorithm is not to find the smallest number of non-preemptive groups that keep the set schedulable, but rather to assign tasks to non-preemptive groups in such a way that the overall requirement of stack space is minimized (which means introducing the stack requirements of each task as a factor). Third, tasks are not considered independent but are allowed to interact through mutually exclusive critical sections (i.e., shared memory) and finally, the above methodology has been extended to multiprocessor systems, where tasks allocated on different processors can interact through shared memory.

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 as 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 [25], [22] in the context of the fixed priority scheduling of independent tasks in uniprocessor systems. The mechanism has been implemented (in a proprietary form) in the SSX kernel from REALOGY [7] and the ThreadX kernel from Express Logic [9], both targeted to embedded system with small code requirements. In this paper, this idea is extended to dynamic scheduling and to non-independent task sets. Moreover, the optimization algorithm presented in [22] has been improved by considering the stack requirements of each task in order to minimize the overall stack requirement rather than the number of task groups.

The algorithms presented in this paper are based on the Stack Resource Policy (SRP), a synchronization protocol presented by Baker in [2]. It allows to share resources in uniprocessor systems with a predictable worst case blocking time. The SRP is similar to the Priority Ceiling Protocol of Sha, Lehoczky and Rajkumar (see [23]), but has the additional property that a task is never blocked once it starts executing. Moreover, SRP can be used on top of an Earliest Deadline scheduler and can be implemented with very little overhead.

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 (i.e., resources that are shared between different processors);

— 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 [21] in for scheduling a set of real-time tasks with shared resources on a multi-processor. It extends the Priority Ceiling Protocol [23] 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 [16] and [8], even when tasks are considered independent. Several heuristic algorithms have been proposed in the literature [5], [13], [20], [15], 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. This solution allows tasks to use local resources under the SRP policy and to access global resources with a predictable blocking time without interfering with the local execution order. This mechanism, when used in conjunction with preemption thresholds and the creation of non-preemptive task groups allows to perform time guarantees minimizing the requirement for RAM space.

In Section 8 we propose a simulated annealing based algorithm for allocating tasks to processors. The algorithm tries to find an optimal task allocation, that is the allocation that keeps the task set schedulable and groups tasks in order to use the minimal amount of RAM.

## 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 we extend the results to the case of multi-processor systems. The subset of tasks assigned to processor $P_k$ will be denoted by $\mathcal{T}_{P_k} \subset \mathcal{T}$.

A real time task $\tau_i$ 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, called minimum interarrival time.

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

Tasks communicate through shared memory. A portion of memory shared by two or more tasks is referred as a *mutually exclusive resource* (or simply *resource*). 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 $\tau_i$ on resource $\rho^j$ is denoted by $\xi^j_{ik}$ and its maximum duration is denoted by $\omega^j_{ik}$.

All memory is statically allocated and no dynamic memory allocation is permitted. This is not a great limitation in embedded real-time systems, since typical general purpose dynamic allocation schemes are not adequate for real-time system programming.

A *scheduling algorithm* is an algorithm that decides which task is to be executed at each instant. In this paper we will consider the *Earliest Deadline First* scheduling algorithm and its variants, which selects the task with the earliest deadline. The sequence of execution of the tasks is called *schedule*. A schedule is *feasible* when all the jobs finish before their deadlines. A task set is *schedulable* with a given scheduling algorithm if every produced schedule is feasible. For a given scheduling algorithm, a *schedulability* test is used to guarantee a-priori that a task set is *schedulable*.

## 4. Background

### *4.1. Stack Resource Policy (SRP)*

The Stack Resource Policy was proposed by Baker in [2] for scheduling a set of real-time tasks on a uniprocessor system. It can be used together 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 $\tau_i$ must be assigned a priority $p_i$ and a static preemption level $\lambda_i$, such that the following essential property holds:

$$\text{task } \tau_i \text{ is not allowed to preempt task } \tau_j, \text{unless } \lambda_i > \lambda_j.$$

Under EDF and RM, the previous property is verified if preemption levels are inversely proportional to the periods of tasks:

$$\forall\, \tau_i \quad \lambda_i \propto \frac{1}{\theta_i}.$$

When the SRP is used together with the RM scheduler, each task is assigned a static priority that is inversely proportional to its period. Hence, under RM, the priority equals the preemption level. Instead, when the SRP is used with the EDF scheduler, in addition to the static preemption level, each job has a priority that is inversely proportional to its absolute deadline.

Every resource $\rho^k$ 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 $\lambda_i$. 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 [2] Baker proposed the following schedulability condition for the EDF scheduler:

$$\forall i, \quad 1 \leq i \leq n \quad \sum_{k=1}^{i} \frac{C_k}{\theta_k} + \frac{B_i}{\theta_i} \leq 1. \tag{1}$$

A tighter condition, proposed in [17], is the following:

$$\forall i, \quad 1 \leq i \leq n \quad \forall L, \quad \theta_i \leq L \leq \theta_n \quad L \geq \sum_{k=1}^{i} \left\lfloor \frac{L}{\theta_k} \right\rfloor C_k + B_i. \tag{2}$$

In all cases, the maximum local blocking time for each task $\tau_i$ can be calculated as the longest critical section $\xi_{jh}^k$ accessed by tasks with longer periods and with a ceiling greater than or equal to the preemption level of $\tau_i$.

$$B_i = \max_{\tau_j \in \mathcal{T}, \forall h} \{\omega_{jh}^k \mid \lambda_i > \lambda_j \wedge \lambda_i \leq \mathrm{ceil}(\rho^k)\}. \tag{3}$$

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 easily be 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, one problem with the SRP is the fact that it does not scale to multiprocessor systems. In Section 7 we propose an extension of the SRP to be used in multiprocessor systems.

### *4.2. Preemption Thresholds*

Given a non-interleaved execution of the application tasks (obtained, for example, by using the SRP), 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, [22], [25], developed the concept of *Preemption Threshold*: each task $\tau_i$ is assigned *a nominal priority* $\pi_i$ and a preemption threshold $\gamma_i$ with $\pi_i \leq \gamma_i$. 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 [22], we introduce the following definitions:

DEFINITION 1 *Two tasks* $\tau_i$ *and* $\tau_j$ *are* ***mutually non-preemptive*** *if* $(\pi_i \leq \gamma_j) \wedge (\pi_j \leq \gamma_i)$.

DEFINITION 2 *A set of tasks* $G = \{\tau_1, \ldots, \tau_m\}$ *is a* ***non-preemptive group*** *if, for every pair of tasks* $\tau_j \in G$ *and* $\tau_k \in G$, $\tau_j$ and $\tau_k$ *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.

After assigning preemption thresholds, the task set can be partitioned into *non–preemptive groups*. However notice that a given preemption threshold assignment may correspond to more than one partition of non-preemptive groups. The maximum number of tasks that can be on the stack at the same time corresponds to the cardinality of the partition of non-preemptive groups.

Obviously, a small number of groups results in a lower requirement for the stack size.

In the following section, 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. In Section 6 we will present an optimization algorithm for reducing the stack size in single processor systems.

## 5. Integrating Preemption Thresholds 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 $\tau_i$ and task $\tau_j$ mutually non-preemptive, we can let them share a *pseudo-resource* $\rho^k$: the ceiling of resource $\rho^k$ is the maximum between the preemption levels of $\tau_i$ and $\tau_j$. At run time, instances of $\tau_i$ or $\tau_j$ will lock $\rho^k$ when they start executing and hold the lock until they finish.

Suppose task $\tau_i$ needs a set of pseudo-resources $\rho^1, \ldots, \rho^h$. When $\tau_i$ 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* $\gamma_i$ of task $\tau_i$. 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 compatible 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 one of Equations (1) and (2), 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.

### 5.1. Blocking Due to Local Resources

Assuming relative deadlines equal to periods, the maximum local blocking time for each task $\tau_i$ can be calculated using Equation (3). 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).

### 5.2. Blocking Due to Pseudo-Resources

A task $\tau_i$ may experience an additional blocking time due to the non-preemptability of lower priority tasks. This blocking time can be computed as follows:

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

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 (3), because:

— $\gamma_i$ is $\max_k \textit{ceil}(\rho^k) = \textit{ceil}(\rho^{k'})$ where $k' \in \{k : \gamma_i = \textit{ceil}(\rho^k)\}$

— $(C_i)$ is the critical section duration for resource $k'$ (remember that pseudo-resources are locked when an instance starts and is unlocked when an instance finishes; moreover, we can consider only the $k'$ critical section for each task since they all have length equal to $C_i$ and $\forall\, k, ceil(\rho^k) \leq ceil(\rho^{k'}) = \gamma_i$).

EXAMPLE Consider the example of Figure 2(b). The three tasks $\tau_0$, $\tau_1$ and $\tau_2$ are sporadic, and their computation times and minimum interarrival times are respectively $C_0 = 3$, $\theta_0 = 12$, $C_1 = 3$, $\theta_1 = 8$, $C_2 = 2$, $\theta_2 = 6$. By definition of preemption level, we have $\lambda_0 < \lambda_1 < \lambda_2$. We want to make $\tau_1$ and $\tau_2$ mutually non preemptive, so we introduce a pseudo-resource $\rho^o$. Every time $\tau_1$ (or $\tau_2$) starts executing, it locks $\rho^o$, and holds the lock until it finishes. The ceiling of $\rho^o$ is $\max(\lambda_1, \lambda_2) = \lambda_2$. By definition of preemption threshold, $\gamma_1 = \gamma_2 = ceil(\rho^o) = \lambda_2$, whereas $\gamma_0 = \lambda_0$.

In this way, we have two preemption groups, the first consists of tasks $\tau_1$ and $\tau_2$, the second contains only $\tau_0$. Hence, the blocking time of $\tau_2$ is:

$$B_2 = B_2^{pseudo} = C_1 = 3$$

and, substituting in Equation (1), the system results schedulable.

The algorithm works as follows [see Figure 2(b)]:

— At time $t = 0$, task $\tau_0$ is activated and starts executing. The system ceiling $\Pi_s$ is equal to $\gamma_0$.

— At time $t = 2$, task $\tau_1$ arrives, and since its priority is the highest and $\Pi_s = \gamma_0 < \lambda_1$, it preempts task $\tau_0$. Now the system ceiling is equal to $\Pi_s = \gamma_1$.

— At time $t = 3$, task $\tau_2$ arrives, and even though it has the highest priority, its preemption level $\lambda_2$ is not higher than the current system ceiling. Hence, according to SRP it is blocked, and $\tau_1$ continues to execute.

— At time $t = 5$, task $\tau_1$ finishes, and the system ceiling returns to the previous value $\gamma_0$. At this point, task $\tau_2$ can start executing. The system ceiling is raised to $\gamma_2$.

Notice that, if $\tau_0$ is also included in the same preemption group as $\tau_1$ and $\tau_2$, the system remains schedulable and the stack size can be further reduced. □

The SRPT presents two main advantages:

— It seamlessly integrates access to mutually exclusive resources and preemption thresholds with a very little implementation effort and with no additional overhead;

— 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 [18], [3], [4] and in the non-preemptive[3] version [14]; furthermore, in [17] 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.

It is clear that in our methodology we are sacrificing task response time versus memory size. However, the response time of some task could be critical and should be maintained as low as possible. In this case, it is possible to reduce the relative deadline of that task to increase its responsiveness. For simplifying the presentation, we do not consider here the case of tasks with deadline smaller than the period. For further details, please refer to [12].

## 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. The algorithm presented in this section implicitly uses pseudo resources to raise the threshold of a task. To simplify the presentation, we do not consider here the use of shared resources. Shared resources can be taken into account using the techniques presented in Section 4.2. The complete algorithm for multiprocessors will be presented in Section 8.

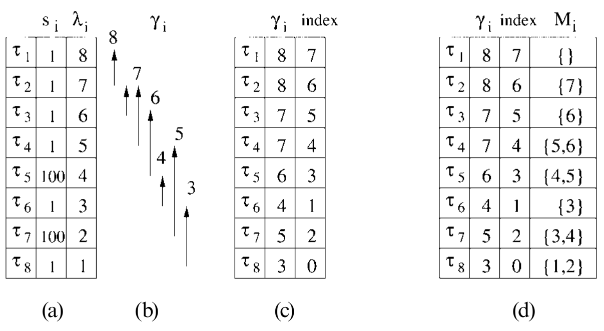
The algorithm requires each task to be characterized by its worst case execution time $C_i$, its period $\theta_i$, its maximum stack requirement (in bytes) $s_i$ and its preemption level $\lambda_i$. At the end of the optimization algorithm, each task $\tau_i$ will be assigned a preemption threshold $\gamma_i$ and will be inserted in a non-preemptive group $G_k$. The goal of the optimization algorithm is:

**Step 1.** To find an assignment of preemption thresholds to tasks, maintaining the feasibility of the schedule; and

**Step 2.** To find the set of non-preemptive groups that minimizes the maximum required stack size.

Notice that unfortunately a preemption threshold assignment does not determine univocally a set of non-preemptive groups. Hence, after assigning the preemption threshold we still do not know the maximum number or tasks that can be on the stack at the same time and how much memory must be allocated for the stack. For this reason, we need to perform step 2.

Our optimization algorithm works as follows:


8
7
6
5
4
3
(a)
(b)
(c)
(d)

(a)

| | $s_i$ | $\lambda_i$ |
|---|---|---|
| $\tau_1$ | 1 | 8 |
| $\tau_2$ | 1 | 7 |
| $\tau_3$ | 1 | 6 |
| $\tau_4$ | 1 | 5 |
| $\tau_5$ | 100 | 4 |
| $\tau_6$ | 1 | 3 |
| $\tau_7$ | 100 | 2 |
| $\tau_8$ | 1 | 1 |

(c)

| | $\gamma_i$ | index |
|---|---|---|
| $\tau_1$ | 8 | 7 |
| $\tau_2$ | 8 | 6 |
| $\tau_3$ | 7 | 5 |
| $\tau_4$ | 7 | 4 |
| $\tau_5$ | 6 | 3 |
| $\tau_6$ | 4 | 1 |
| $\tau_7$ | 5 | 2 |
| $\tau_8$ | 3 | 0 |

(d)

| | $\gamma_i$ | index | $M_i$ |
|---|---|---|---|
| $\tau_1$ | 8 | 7 | {} |
| $\tau_2$ | 8 | 6 | {7} |
| $\tau_3$ | 7 | 5 | {6} |
| $\tau_4$ | 7 | 4 | {5,6} |
| $\tau_5$ | 6 | 3 | {4,5} |
| $\tau_6$ | 4 | 1 | {3} |
| $\tau_7$ | 5 | 2 | {3,4} |
| $\tau_8$ | 3 | 0 | {1,2} |

*Figure 3.* An example: The minimum total stack size does not correspond to the minimum number of non-preemptive groups: a) Initial task set, b) computation of the preemption thresholds, c) reordering, and d) computation of the maximal groups.

- Tasks are ordered by decreasing preemption level;
- *Step 1*: We use the algorithm described in [22] 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 $\gamma_i$ of each task $\tau_i$, to the maximum level that allows to preserve the schedulability of all tasks (i.e., incrementing the preemption threshold is allowed only if the task set remains schedulable). The algorithm stops when a further increment on any task makes the system not schedulable.
- *Step 2*: Given a feasible assignment of preemption thresholds, we partition the task set into non-preemptive groups and compute the maximum required stack size.

Our algorithm differs from the one in [22] in the final optimization objective: while the algorithm in [22] 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 [22] where the maximum overall stack requirement does not correspond to the minimum number of groups, as shown in the example of Figure 3. In this example there are 8 tasks, all having a stack frame size of 1 except $\tau_5$ and $\tau_7$ which have a stack frame size of 100 [Figure 3(a)]. The partition in non-preemptive groups provided by algorithm OPT-PARTITION in [22] is $(G_1 = \{\tau_1, \tau_2\})$, $(G_2 = \{\tau_3, \tau_4, \tau_5\})$, $(G_3 = \{\tau_6, \tau_7, \tau_8\})$, which leads to a total stack size of 201. However, note that task $\tau_7$ and task $\tau_5$ are mutually non-preemptive (in fact, $\lambda_5 < \gamma_7$ and $\lambda_7 < \gamma_5$). If we consider groups $(G_1 = \{\tau_1\})$, $(G_2 = \{\tau_2, \tau_3\})$, $(G_3 = \{\tau_4, \tau_5, \tau_7\})$, $(G_4 = \{\tau_6, \tau_8\})$ the total stack requirement is 103. Note that, in this case, by using the solution with the minimum number of groups, we would have overestimated the maximum stack requirement by a factor of 2.

Figure 3(b) shows the result of the *preemption threshold assignment phase* for our example. The preemption thresholds of all tasks (except $\tau_1$) are raised to the values in the column marked as $\gamma_i$. The algorithm used to partition the task set into preemption groups (*step 2*) is described in the remaining of this section. First some definitions:

DEFINITION 3 *A* ***representative task*** *for a non-preemptive group is the task having the the smallest threshold among all the tasks in the group.*

In the following, $G_i$ will denote a non-preemptive group with representative task $\tau_i$.

DEFINITION 4 *A* ***maximal group*** *for $\tau_i$ is the biggest non-preemptive group that can be created having $\tau_i$ as a representative task.*

In the following, we denote with $M_i$ the maximal group for task $\tau_i$ minus $\tau_i$. For example, if the maximal group for task $\tau_1$ is $\{\tau_1, \tau_2, \tau_3\}$, we denote $M_1 = \{\tau_2, \tau_3\}$.

With these definitions, the algorithms in *step 2* is the following:

— Tasks are ordered by increasing preemption thresholds, ties are broken in order of decreasing stack requirements. For clarifying the algorithm, after ordering we rename each task using an increasing *index* [see the example in Figure 3(c)]; this index will be used from now on for identifying the tasks. Thus, if $\tau_i$ is assigned index $j$, in the following it will be referred to as $\tau_j$. Hence, according to the new ordering,

$$i < j \Rightarrow \gamma_i < \gamma_j \vee \left(\gamma_i = \gamma_j \wedge s_i \geq s_j\right)$$

where $s_i$ is the stack requirement for task $\tau_i$

— The algorithm starts by finding the set $M_i$ for each task $\tau_i$. In Figure 3(d), the $M_i$ are computed for each task and are shown in the last column.

— Then, the algorithm calls a recursive function that allocates all tasks to non-preemptive groups.

The function, called **creategroup()**, recursively computes all possible partitions of the tasks into non-preemptive groups, and computes the maximum stack requirement for each partition. The minimum among these requirements will be the maximum memory for the stack that we need to allocate in our system.

Enumerating all possible partitions in non-preemptive groups clearly takes exponential time. We claim that the problem of finding the required stack memory size, given a preemption threshold assignment, is NP-hard. This claim is supported by the observation that the problem is somewhat similar to a bin-packing problem, which is known to be NP-hard.

Hence, a great effort has been devoted in trying to reduce the mean complexity of Algorithm **creategroup** by pruning as soon as it is possible all the solutions that are clearly non-optimal. In the following, we give a description of Algorithm **creategroup**, and proof sketches of its correctness.

```
int ministack = $\sum s_i$;
$F = \mathcal{T}$;
$\forall i \quad G_i = \{\}$;

creategroup ($\tau_g$ , sum) {
  int newsum;
  $G_g$.insert($\tau_g$);
  $\tau_i = M_g$.queryFirst();
  end = false;
  do {
      $\forall \tau_j \in M_g$ if ($\tau_j \in F$ and $j \geq i$) {
          $G_g$.insert($\tau_j$)
          $F$.remove($\tau_j$);
      }
      newsum = sum + stackUsage($G_g$);
      if (!empty($F$)) {
          if (condition1) {
             $\tau_f = F$.removeFirst();
             if (condition2) creategroup($\tau_f$ , newsum);
             else {
                   $F$.insert($\tau_f$);
                   end = true;
             }
          }
          if ($G_g \neq \{\tau_g\}$ and !end {
             while ($G_g \neq \tau_g$ and condition3) {
                   $\tau_h = G_g$.removeLast();
                   $F$.insert($\tau_h$);
             }
             $\tau_i = M_g$.queryNext($\tau_h$);
          } else end = true;
      } else {
          if (newsum < minstack) {
             minstack = newsum;
             NewCandidate()
          }
          end = true
      }
  } while (!end);
  $F$.insertAll($G_g$);
  $G_g$.removeAll();
```

*Figure 4.* The creategroup() recursive function.

The pseudo-code for **creategroup** is shown in Figure 6. The algorithm is recursive: at each level of recursion a new non-preemptive group is created. The following global variables are used: `minstack` contains the value of the candidate optimal solution and is initialized to the sum of the stack requirements of all tasks; $F$ is the set of tasks that have not been allocated yet; $G_1, \ldots, G_n$ are the non-preemptive groups that will be created by the function; $M_1, \ldots, M_n$ are the maximal groups. The parameters of

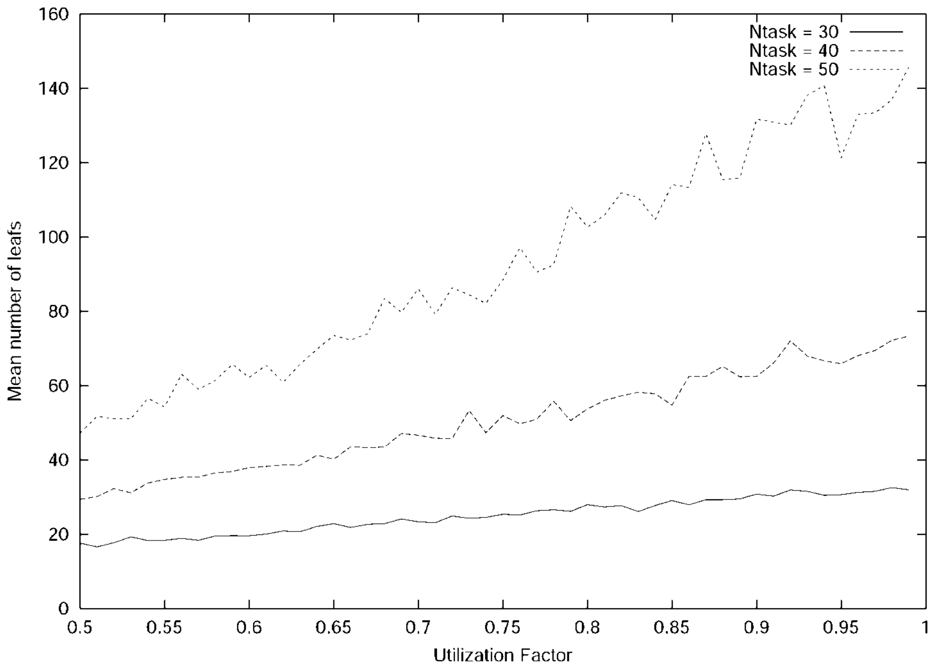


*Figure 5.* Mean number of *cuts* for different task set sizes.

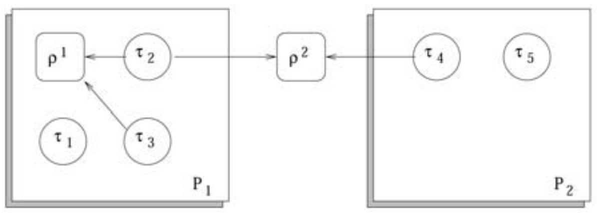


*Figure 6.* Structure of the example.

**creategroup** are $\tau_g$, which is the representative task based on which a new group $G_g$ will to be created and `sum` that is the sum of the stack requirement of the already created groups.

The key point in understanding how **creategroup** works is that the space of candidate solutions is explored in a well defined order: in particular, all the task sets mentioned so far are ordered by increasing preemption thresholds (ties are broken in order of increasing stack requirements).

**creategroup** is first invoked with $\tau_1$ (that is the task with the lowest preemption threshold) and $sum = 0$. When invoked (at the $k$-th level of recursion), function

**creategroup** builds a non-preemptive group for task $\tau_g$ by inserting all tasks from $M_g$ that are not allocated yet (lines 3–10).

Now, if there are still tasks to be allocated (line 12), **creategroup** tries to recursively call itself in order to compute the next group. However, this recursion is not performed if we are sure that no better solution can be found in this branch (Condition 1 at line 13 and Condition 2 at line 15).

Then, the algorithm does backtracking, by extracting tasks from $G_g$ and inserting them back in $F$, in order to explore all possible configurations for this branch (lines 21–26). Condition 3 at line 22 further reduces the number of solution to be checked by pruning some configuration of $G_g$ that cannot improve the optimal solution.

If this is the last level of recursion for this branch (it happens when $F$ is empty), we check whether the current candidate solution is optimal, and, if it is the case, we save the value of this solution in `minstack` and the current group configuration by calling function `NewCandidate()`. Notice that, before returning to the previous recursion level, all tasks are removed from $G_g$ (lines 36–37).

Now we describe the conditions that allow **creategroup** to prune non-optimal branches. Let us define the *required tasks* as the representative tasks of the non-preemptive groups found by Algorithm OPT-PARTITION (described in [22]). These tasks are important because in every possible partition, they will always be in different non-preemptive groups. Hence, our solution is bounded from below by the sum of the stack sizes of the *required tasks*.

**Condition 1** is false if the sum of `newsum` and the size of the stack of the required tasks that have not been allocated yet is greater than or equal to `minstack`.

The correctness of this condition is trivially proven.

**Condition 2** is false when $\gamma_g = \gamma_f$.

THEOREM 1 *If Condition 2 does not hold,* (i.e., $\gamma_g = \gamma_f$), *then any solution with* $\tau_f$ *as representative task of a new non-preemptive group cannot achieve a solution with a lower stack requirement than the already explored solutions.*

*Proof.* First we will prove that $G_f \subset M_g$. Consider $\tau_i \in G_f$, it follows that $\lambda_i \leq \gamma_f$, $\lambda_f \leq \gamma_i$ and $\gamma_f \leq \gamma_i$. Since $\gamma_f = \gamma_g$, it follows that $\lambda_i \leq \gamma_f \leq \gamma_g$, and $\lambda_g \leq \gamma_g = \gamma_f \leq \gamma_i$. As a consequence, $\tau_i$ and $\tau_g$ are mutually non preemptive, and $\tau_i$ is in $M_g$. Hence, $G_f \subset M_g$.

Since the current iteration that has $\tau_g$ as a representative task visits all the subsets of $M_g$, and $G_f \subset M_g$ it follows that any configuration produced by calling recursively the **creategroup** algorithm with representative task $\tau_f$ leads to an overall stack usage that is bounded from below by the following expression:

$$\texttt{stackUsage}(G_g) + \texttt{stackUsage}(G_f) \geq \\ \texttt{stackUsage}(G_g \cup G_f) \geq \texttt{stackUsage}(M_g).$$

However, $M_g$ is a branch that has already been explored (it is the first branch that algorithm **creategroup** explores). Hence, the theorem follows.

**Condition 3** is true when the task with the maximum stack requirement in $G_g$ has been removed, that is when the stack usage of $G_g$ has been reduced.

THEOREM 2 *All the branches in which the task with the maximum stack requirement has not been removed from $G_g$ cannot lead to a better solution than the already explored ones.*

*Proof Sketch.* The basic idea is based on the fact that solutions are explored in a given order. In particular, the first solution is the *greedy* solution, where $G_g$ is the biggest non-preemptive group that can be built.

When considering the next configuration for $G_g$, it must have a lower stack requirement than the previous one. In fact, all solutions in which $G_g$ has the same stack requirement are bounded from below by the first solution explored. Hence, Condition 3 forces the removal of tasks from $G_g$ until its stack requirement is reduced.

We do not report the complete proof of this theorem due to space constraints. The interested reader can refer to [12] for the complete proof. □

As an example, a typical run of the algorithm on the task set of Table 1 will work as follows:

— To find the first solution, three recursive calls are needed, creating groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_4, \tau_5\}$, and $G_6 = \{\tau_6, \tau_7\}$, with a total stack of 201. This first solution is equal to that found by the algorithm OPT-PARTITION proposed in [22].

— Then, group $G_6$ is rolled back. Task 5 is removed from group $G_3$, and $\tau_i$ is set to the next task ($\tau_5$ will not be reconsidered for inclusion in the next group configuration). The recursive calls produce groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_4\}$, $G_5 = \{\tau_5, \tau_6\}$, $G_7 = \{\tau_7\}$.

— Group $G_7$ is rolled back and $\tau_6$ is removed from group $G_5$. The recursive calls produce groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_4\}$, $G_5 = \{\tau_5\}$, $G_6 = \{\tau_6, \tau_7\}$.

— Then, groups $G_6$ and $G_5$ are rolled back, and $\tau_4$ is removed from $G_3$. Now $\tau_i$ is moved past $\tau_4$, and $\tau_5$ is re-inserted into $G_3$. Next, $\tau_4$ and $\tau_7$ are chosen as representative tasks giving groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_5\}$, $G_4 = \{\tau_4, \tau_6\}$, $G_7 = \{\tau_7\}$.

— Again, $G_7$ is emptied and $\tau_6$ is removed from $G_4$. Group $G_6$ is created, giving groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3, \tau_5\}$, $G_4 = \{\tau_4\}$, $G_6 = \{\tau_6, \tau_7\}$.

— At this point, groups $G_6$ and $G_4$ are removed; $\tau_5$ is also removed from $G_3$. Then, groups $G_4$ and $G_7$ will be created, giving groups $G_0 = \{\tau_0, \tau_1, \tau_2\}$, $G_3 = \{\tau_3\}$, $G_4 = \{\tau_4, \tau_5, \tau_6\}$, $G_7 = \{\tau_7\}$.

— After some other non optimal solutions, the first recursive call of **creategroup()** will remove $\tau_2$ from $G_0$, letting the creation of group $G_2 = \{\tau_2, \tau_3, \tau_4\}$ that will bring the algorithm to the optimal solution.

As already mentioned, the complexity of the algorithm is exponential in the number of tasks. However, since the number of groups in the optimal solution is often small, the number of combinations to evaluate is limited. Thanks to the efficiency of the pruning, the number of solutions is further reduced. In Figure 6 the average number of explored solutions (leafs) is plotted against the load of the system and for different number of tasks: the resulting average number is quite low even for large task sets. We conclude that, for typical embedded systems in the domain of automotive applications where the number of tasks is relatively small, 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 (MSRP)*

Suppose that tasks have already been allocated to processors. Depending on this allocation, resources can be divided into *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);

— the task performs a busy wait (also called *spin lock*).

We want to maintain the properties of the SRP: in particular, we want 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 required is the status of the global resources.

The MSRP algorithm works as follows:

- 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 $\Pi_k$ for every processor $P_k$.
- Tasks are allowed to access local resource through nested critical sections. It is possible to nest local and global resources. However, it is possible to nest global critical sections only if they are accessed in a particular order, otherwise a deadlock can occur. For simplifying the presentation we did not consider nested global critical sections: however, this improvement can be done with little effort.
- For each global resource, every processor $P_k$ defines a ceiling greater than or equal to the maximum preemption level of the tasks on $P_k$.
- When a task $\tau_i$, allocated to processor $P_k$ accesses a global resource $\rho^j$, the system ceiling $\Pi_k$ is raised to ceil$(\rho^j)$ 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.
- When a task $\tau_i$, allocated to processor $P_k$, releases a global resource $\rho^j$, the algorithm checks the corresponding FCFS queue, and, in case some other task $\tau_j$ is waiting, it grants access to the resource, otherwise the resource is unlocked. Then, the system ceiling $\Pi_k$ is restored to the previous value.

EXAMPLE Consider a system consisting of two processors and five tasks as shown in Figure 7. Tasks $\tau_1$, $\tau_2$ and $\tau_3$ are allocated to processor $P_1$: task $\tau_3$ uses local resource $\rho^1$, task $\tau_2$ uses resources $\rho^1$ and $\rho^2$ through nested critical sections, and $\tau_1$ does not use any resource. Tasks $\tau_4$ and $\tau_5$ are allocated to processor $P_2$: task $\tau_4$ uses the global resource $\rho^2$ and $\tau_5$ does not uses resources. The parameters of the tasks are reported in

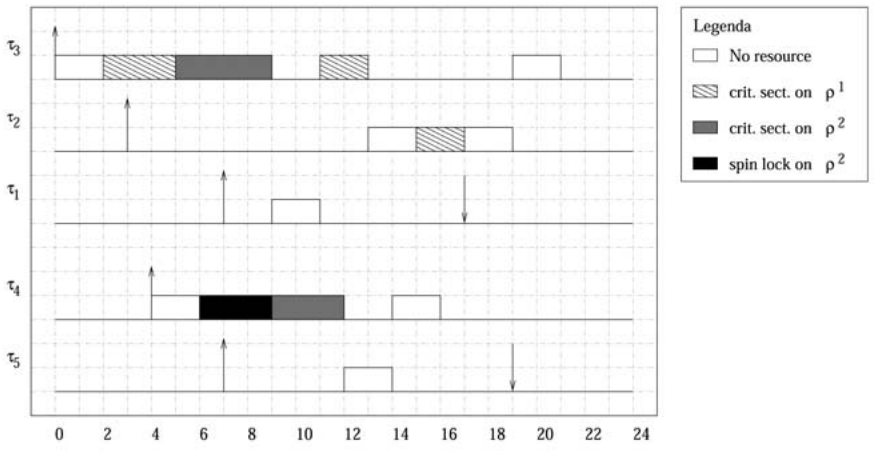


*Figure 7.* An example of schedule produce by the MSRP on two processors.

*Table 1.* The Example Task Set

|  | $C_i$ | $\lambda_i$ | $\omega_{ij}^1$ | $\omega_{ij}^2$ | $ts_i$ | $C_i'$ | $B_i^{local}$ | $B_i^{local}$ |
|---|---|---|---|---|---|---|---|---|
| $\tau_1$ | 2 | 3 | 0 | 0 | 0 | 2 | 0 | 7 |
| $\tau_2$ | 6 | 2 | 2 | 0 | 0 | 6 | 9 | 7 |
| $\tau_3$ | 11 | 1 | 9 | 4 | 3 | 14 | 0 | 0 |
| $\tau_4$ | 7 | 1 | 0 | 3 | 4 | 11 | 0 | 0 |
| $\tau_5$ | 2 | 2 | 0 | 0 | 0 | 2 | 0 | 7 |

Table 1. The ceiling for resource $\rho^1$ is 2. The ceiling for resource $\rho^2$ on processor $P_1$ is 3, and on processor $P^2$ is 2. A possible schedule is shown in Figure 7. Notice that:

— At time $t = 3$, task $\tau_2$ 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 $\tau_3$ locks resource $\rho_2$ and raises the system ceiling $\Pi_1$ to 3.

— At time $t = 6$, task $\tau_4$ tries to access the global resource $\rho^2$ which is currently locked by $\tau_2$. Thus, it raises the system ceiling of processor $P_2$ to 2 and performs a busy wait.

— At time $t = 7$, both $\tau_1$ and $\tau_5$ are blocked, because the system ceilings of the two processors are set to the maximum.

— At time $t = 8$, task $\tau_3$ releases the global resource $\rho^2$ and task $\tau_4$ can enter the critical section on $\rho^2$. At the same time, the system ceiling of processor $P_1$ is set back to 2, and task $\tau_1$ can make preemption.

### *7.2. Schedulability Analysis of the MSRP*

First, we give an upper bound on the time that task $\tau_i$, allocated to processor $P_k$, can spend waiting for a global resource $\rho^j$. 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:*

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

*Proof.* On each processor, only one task can be inside a global critical section or waiting for a global resource. In fact, when a task tries to access a critical section on a global resource, it first raises the system ceiling to the maximum possible value, becoming non-preemptable. Tasks that are waiting on a global critical section are served in a FCFS order: hence, a task allocated to $P_k$ that tries to access $\rho^j$, has to wait for at most the duration of the longest global critical section on $\rho^j$ for each processor $p \neq P_k$. This condition does not depend on the particular task on processor $P_k$. Hence, the lemma follows.

Basically, the spin lock time increments the duration $\omega^j_{ih}$ of every global critical section $\xi^j_{ih}$, and, consequently, the worst case execution time $C_i$ of $\tau_i$. Moreover, it also increments the blocking time of the tasks allocated to the same processor with a preemption level greater than $\lambda_i$.

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

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

We also define the *actual worst case computation time* $C'_i$ for task $\tau_i$ 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.

LEMMA 2 *When task $\tau_j$ starts executing:*

1. All the local resources required by the $\tau_j$ are unlocked;

2. All the local resources required by every task that can preempt $\tau_j$ are unlocked.

*Proof.* By contradiction.

1. Suppose that, after $\tau_j$ starts executing, it needs resource $\rho^k$ which is locked by some other task. Since $\tau_j$ is executing, $\lambda_j > \Pi_s$. Moreover, since $\tau_j$ needs $\rho^k$, $ceil(\rho^k) \geq \lambda_j$. But $\rho^k$ is locked, so $\Pi_s \geq ceil(\rho^k) \geq \lambda_j$, and this is a contradiction.

2. Suppose that at time $t$ a task $\tau_H$ preempts $\tau_j$, and that it needs a local resource $\rho^k$ which is locked. By hypothesis, $\lambda_H > \lambda_j > \Pi_s$ (because $\tau_H$ can preempt $\tau_j$) and $\Pi_s \geq ceil(\rho^k)$ (because $\rho^k$ is locked). The lemma follows because $\tau_H$ uses $\rho^k$, that implies $ceil(\rho^k) \geq \lambda_H$.

THEOREM 3 *Once a job starts executing it cannot be blocked, but only preempted by higher priority jobs.*

*Proof.* We prove the theorem by induction. Suppose there are $n$ tasks that can preempt $\tau_j$.

If $(n = 0)$, no task can preempt $\tau_j$. Since when $\tau_j$ started executing $\lambda_j > \Pi_s(t)$, Lemma 2 guarantees that all the resources required by task $\tau_j$ are free, so it can run to completion without blocking.

If $(n > 0)$, suppose that a task $\tau_H$ preempt $\tau_j$. By induction hypothesis $\tau_H$ cannot be blocked, so when it finishes it will release all the resources that it locked, and the task $\tau_j$ will continue until the end since it has all the resources free.

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 4 *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.*

*Proof.* By contradiction. Suppose that a task $\tau_j$ is blocked for the duration of at least 2 critical sections corresponding to resources $\rho^1$ and $\rho^2$. First, suppose that $\rho^1$ is a local resource ($\rho^2$ can be either local or global). It must be the case that there are two lower priority tasks $\tau_{L1}$ and $\tau_{L2}$. The first task locked a resource $\rho^1$ and, while in critical section, it is preempted by $\tau_{L2}$ that locked another resource $\rho^2$. While $\tau_{L2}$ is still in critical section, $\tau_j$ arrives, and it has to wait for $\rho^1$ and $\rho^2$ to be free. This scenario cannot happen, because we have that $ceil(\rho^1) \geq \lambda_j > \lambda_{L2}$ (since $\tau_j$ uses $\rho^1$ and preempted $\tau_{L2}$, and $\lambda_{L2} > ceil(\rho^1)$ (since $\tau_{L2}$ preempted $\tau_{L1}$, when $\tau_{L1}$ locked $\rho^1$).

Now suppose $\rho^1$ is a global resource and consider the previous scenario. When $\tau_{L1}$ locked $\rho^1$, $\tau_{L1}$ become non-preemptable, so $\tau_{L2}$ cannot preempt $\tau_{L1}$.

Finally, note that the length of every global critical section consists of the critical section itself, plus the spin-lock time due to the access to global resources.

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 of 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 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:

### 7.3. Blocking Time Due to Local Resources

This blocking time is equal to the longest critical section $\xi_{jh}^k$ among those (of a task $\tau_j$) with a ceiling greater than or equal to the preemption level of $\tau_i$:

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

### 7.4. Blocking Time Due to Global Resources

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

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

### *7.5. Blocking Time Due to Pseudo Resources*

As explained in the previous sections, this blocking time is due to the fact that a task $\tau_i$ 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 5 *Suppose that tasks on processor $P_k$ are ordered by decreasing preemption level. The schedulability test is as follows:*

$$\forall\, P_k \in \mathcal{P} \quad T_{P_k} = \{\tau_1, \tau_2, \ldots, \tau_{n_k}\} \quad \forall\, i = 1, \ldots, n_k \quad \sum_{l=1}^{i} \frac{C'_l}{\theta_l} + \frac{B_i}{\theta_i} \leq 1. \tag{4}$$

*Proof.* Consider a generic task $\tau_i$ on processor $P_i$. To be schedulable under MSRP with preemption thresholds, it must be schedulable under EDF considering all the blocking times and the spin locks. Hence, a guarantee formula for task $\tau_i$ can be written as

$$\sum_{l=1}^{i-1} \frac{C'_l}{\theta_l} + \frac{C'_i}{\theta_i} + \frac{B_i}{\theta_i} \leq 1$$

where the first part is the bandwidth stolen by tasks that preempt $\tau_i$, $C'_l$ takes into account the wasted bandwidth for the spin-lock time of each preempter task. The second part accounts for the execution time and the spin-lock time of the task to be guaranteed. The third part accounts for the largest blocking time experienced by $\tau_i$ due to the use of resources by lower priority tasks.

In the same way, we can rewrite Equation (2) as follows:

$$\begin{gathered} \forall\, P_k \in \mathcal{P} \quad T_{P_k} = \{\tau_1, \tau_2, \ldots, \tau_{n_k}\} \quad \forall\, i, \quad 1 \leq i \leq n_k \\ \forall\, L, \quad \theta_i \leq L \leq \theta_{n_k} \quad L \geq \sum_{l=1}^{i} \left\lfloor \frac{L}{\theta_l} \right\rfloor C'_l + B_i. \end{gathered} \tag{5}$$

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.

EXAMPLE 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 main differences between the MSRP and the MPCP are the following:

— Unlike MPCP, with the MSRP it is possible to use one single stack for all the tasks allocated to the same processor.

— 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 [6]).

— The MSRP, like the SRP, tends to reduce the number of preemptions in the system, hence there is less overhead. However, this comes at the cost of a potentially large spin lock time.

In general, there are situations in which, given an allocation of tasks to processors with resource sharing, MSRP performs better than MPCP, and vice versa. this fact mainly depends on the different blocking time formulas that are used. We are currently investigating a better characterization of the two algorithms. The comparison in fact is made difficult by the fact that there are more than 8 parameters that can vary in an independent way. That comparison is outside the scope of this paper, and will be presented as a future work.

## 8. Optimizing Stack Usage in Multiprocessors

Sections 5 and 7 provide the basis for the implementation of run-time mechanisms for global and local resource sharing in multiprocessor systems. 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 [24], [22], to find the optimal processor binding for real-time tasks to be scheduled according to fixed-priority policies, in [19] to solve the problem of scheduling with minimum jitter in complex distributed systems and in [25] 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 $S$ (all possible assignments of tasks to processors) has dimension $p^n$ where $p$ is the number of processors and $n$ is the number of tasks. 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.

Due to space constraints, we will not explain in detail the SA mechanism and why it works in many combinatorial optimization problems. Please refer to [1] for more details.

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 (see Figure 8).

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:

— Schedulable solutions must always have energy lower than non-schedulable solutions;

— The energy of schedulable solution is proportional to the worst case overall RAM requirements for stack usage;

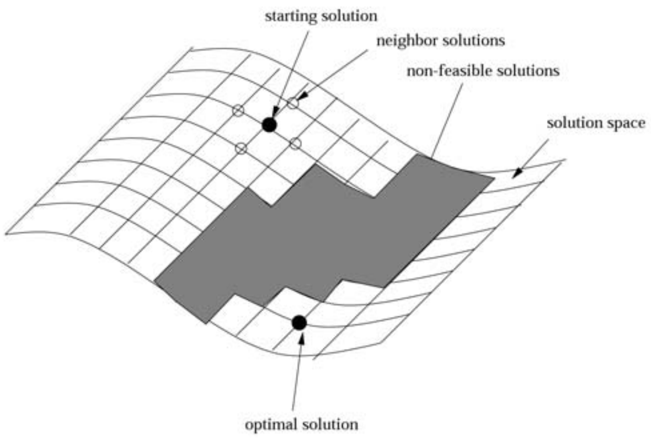


*Figure 8.* Non feasible solutions must be accepted in order to reach the optimal solution.

— The energy of non schedulable solutions is proportional to the maximum excess utilization resulting from the evaluation of formula (4) for non-schedulable tasks.

The purpose of the last requirement is to rank non-schedulable solutions. We feel that solutions that are close to schedulability should be preferred to solutions which are definitely non schedulable. If all non schedulable solutions would be evaluated as equal there is an (albeit small) chance that the algorithm keeps jumping from one non schedulable solution to the other (the transition cost always being 0) endlessly.

If *TotalStack* is the overall stack requirement, obtained by adding up the stack requirements of all tasks, and *OptStack* 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:

$$\begin{cases} \max_{\forall \tau_i}\left(\sum_{k=i}^{n}\frac{C'_k}{\theta_k}+\frac{B_i}{\theta_i}\right) * \mathit{TotalStack} & \text{non schedulable assignment} \\ \mathit{TotalStack} + \Delta * (\mathit{OptStack} - \mathit{TotalStack}) & \text{schedulable assignment.} \end{cases}$$

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 $\Delta$ factor estimates the average ratio between the stack requirements before task grouping and the stack requirements after optimization and is defined as:

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

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 latter value should be chosen to be near to the results obtained in the simulation experiments done for uniprocessors. In our experiments, for example, we chose a value of 4, that is near the double of the mean value obtained from the simulation results (see Section 9).

The $\Delta$ 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 sum 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 except for the experiments of Figures 11 and 12 in which the stack size ranges between 10 and 400 bytes.

In Figure 9 the average number of preemption groups is shown. Figure 10 is a crosscut section of Figure 9.

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

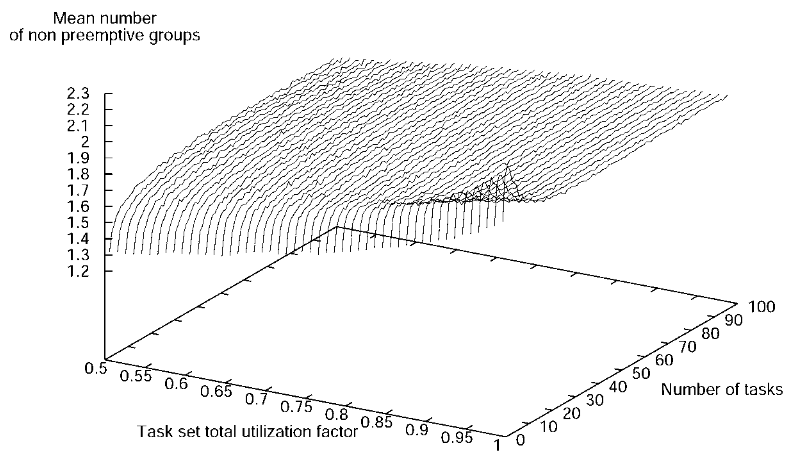


*Figure 9.* Average number of preemption groups.

task is comparable with the period; hence it is more difficult to find a feasible non-preemptive schedule.

— Figure 9 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 11, we compare the optimization algorithm presented in [22] (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 [22] 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. In Figure 12, we plot the average and maximum improvement given by our approach, and its variance varying the spread of the task size.

*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

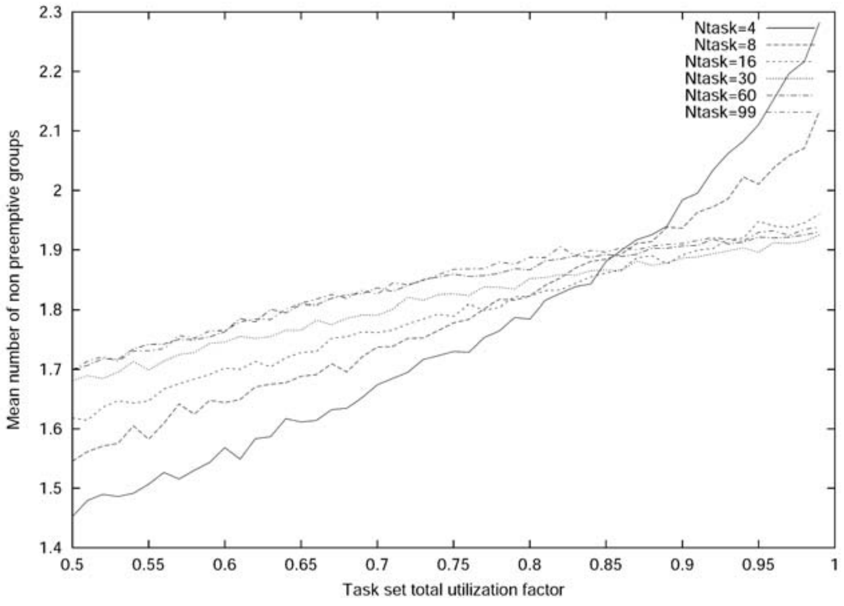


*Figure 10.* Average number of preemption groups for different task set sizes.

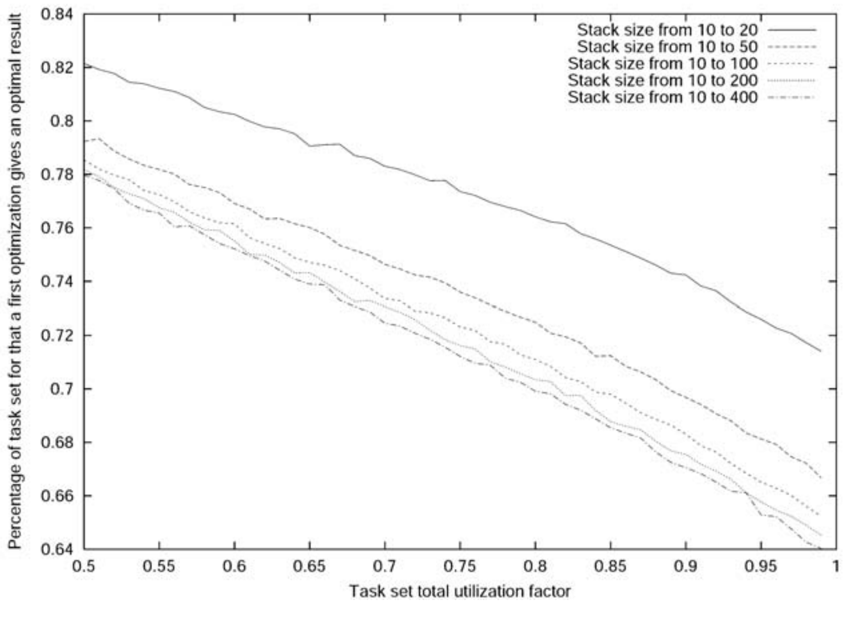


*Figure 11.* Ratio of improvement given by our optimization algorithm.

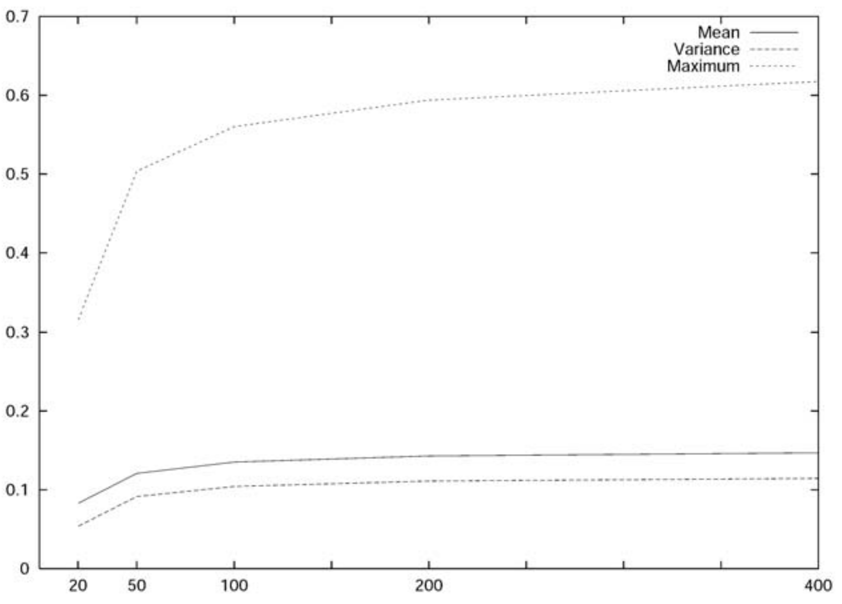


*Figure 12.* Average, maximum and variance of improvement given by our optimization algorithm when randomly varying the tasks stack size from 1 to {20, 50, 100, 200, 400}.

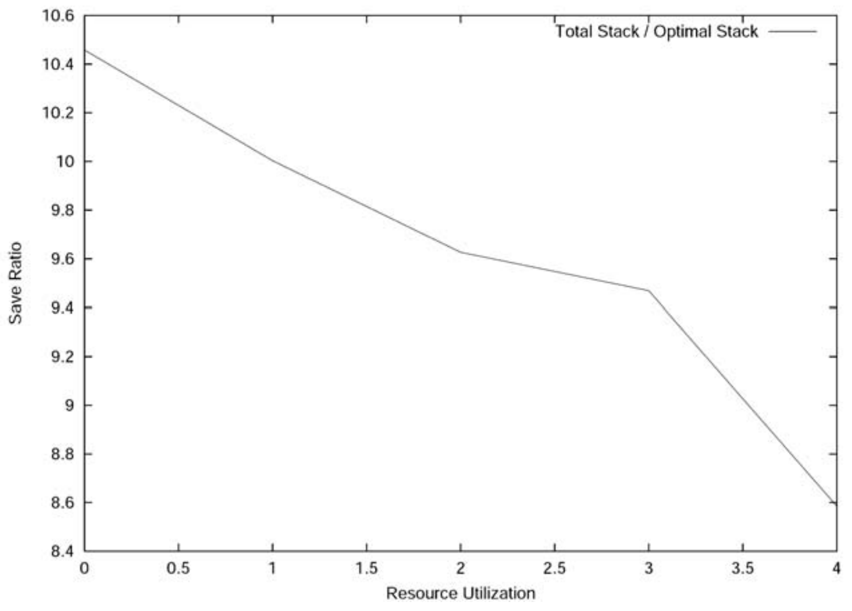


*Figure 13.* Ratio of improvement given by our multiprocessor optimization algorithm when varying the utilization of shared resources.

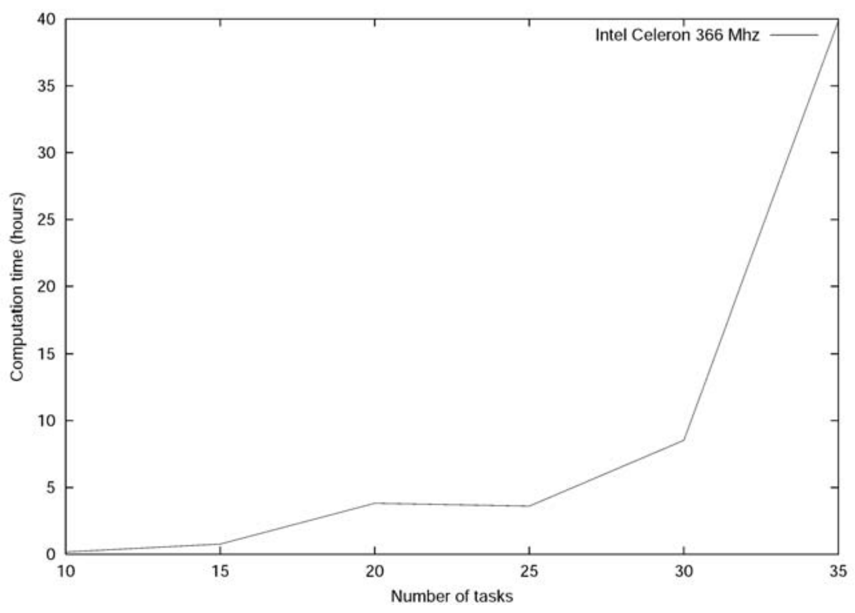


*Figure 14.* Average computation times for the simulated annealing algorithm as a function of the problem size.

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 13).

In Figure 13 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 the 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 between the first schedulable solution found and the final optimized result 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 (Figure 14 shows typical computation times as a function of the task set size). 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

This paper has been strongly motivated by the arrival of the new generation of multiple-processor on-a-chip systems for embedded applications. These platforms not only require real-time executives, but also ask for kernel mechanism that save as much RAM space as possible, RAM memory being one of the most expensive components in those systems.

In this paper, we present a solution for scheduling real-time tasks in single and multiple processor systems with minimal RAM requirements. In uniprocessor 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. The experimental runs show an extremely effective reduction in the occupation of RAM space when compared to conventional algorithms. We plan to implement the algorithms described in this paper in a new version of our ERIKA kernel[5] for the JANUS architecture (2-processors in a single chip).

## Notes

1. http://www.madess.cnr.it/Summary.htm
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.
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.
4. Since EDF is optimal, there is no need to find an initial priority assignment for the task set.
5. http://erika.sssup.it

## References

1. Aarts, E. and J. Korst. *Simulated Annealing and Boltzmann Machines.* John Wiley & Sons, 1989.
2. Baker, T. Stack-Based Scheduling of Real-Time Processes. *Journal of Real-Time Systems*, vol. 3, 1991.
3. Baruah, S., A. Mok, and L. Rosier. Preemptively Scheduling Hard-Real-Time Sporadic Tasks on One Processor. In *Proceedings of the 11th IEEE Real-Time Systems Symposium*, 1990a, pp. 182–190.
4. Baruah, S., L. Rosier, and R. Howell. Algorithms and Complexity Concerning the Preemptive Scheduling of Periodic Real-Time Tasks on One Processor. *The Journal of Real-Time Systems*, vol. 2, 1990b.

5. Burchard, A., J. Liebeherr, Y. Oh, and S. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Transactions on Computers*, 1995.
6. Craig, T. S. Queuing Spin Lock Algorithms to Support Timing Predictability. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1993.
7. Davis, R., N. Merriam, and N. 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*, 2000.
8. Dertouzos, M. L. and A. K.-L. Mok. Multiprocessor On-Line Scheduling of Hard-Real-Time Tasks. *IEEE Transactions on Software Engineering*, vol. 15, no. 12, 1989.
9. Express Logic, I., http: / / www.threadx.com. available on Internet.
10. Ferrari, A., 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, 2000.
11. Gai, P., G. Lipari, L. Abeni, M. di Natale, and E. 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*, 2000.
12. Gai, P., G. Lipari, and M. di Natale. An Algorithm for Stack Size Minimization in Embedded SoC, RETIS TR2001-04. Technical Report, ReTiS Lab, Scuola Superiore S. Anna, 2001.
13. Graham, R. *Bounds on the Performance of Scheduling Algorithms*. Chap. 5, E. G. Coffman Jr. (ed.) *Computer and JobShop Scheduling Theory*, Wiley, New York, 1976.
14. Jeffay, K., D. F. Stanat, C. U. Martel. On Non-Preemptive Scheduling of Periodic and Sporadic Tasks. In *Proceedings of the IEEE Real-Time Systems Symposium*, 1991, pp. 129–139.
15. Khemka, A. and R. K. Shyamasunda. Multiprocessor Scheduling of Periodic Tasks in a Hard Real-Time Environment. Technical report, Tata Institute of Fundamental Research, 1990.
16. Leung, J. Y. T. and J. Whitehead. On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks. *Performance Evaluation*, vol. 2, pp. 237–250, 1982.
17. Lipari, G. and G. Buttazzo. Schedulability Analysis of Periodic and Aperiodic Tasks with Resource Constraints. *Journal of Systems Architecture*, vol. 46, pp. 327–338, 2000.
18. Liu, C. and J. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the Association for Computing Machinery*, vol. 20, no. 1, 1973.
19. Natale, M. D. and J. Stankovic. Scheduling Distributed Real-Time Tasks with Minimum Jitter. *Transactions on Computer*, vol. 49, no. 4, 2000.
20. Oh, Y. and S. H. Son. Allocating Fixed-Priority Periodic Tasks on Multiprocessor Systems. *Journal on Real Time Systems*, vol. 9, 1995.
21. Rajkumar, R. Synchronization in Multiple Processor Systems. In *Synchronization in Real-Time Systems: A Priority Inheritance Approach*, Kluwer Academic Publishers, 1991.
22. Saksena, M. and Y. Wang. Scalable Real-Time System Design using Preemption Thresholds. In *Proceedings of the Real Time Systems Symposium*, 2000.
23. Sha, L., R. Rajkumar, and J. P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, vol. 39, no. 9, 1990.
24. Tindell, K., A. Burns, and A. Wellings. Allocating Real-Time Tasks (an NP-Hard Problem Made Easy). *Real-Time Systems Journal*, 1992.
25. Wang, Y. and M. Saksena. Fixed Priority Scheduling with Preemption Threshold. In *Proceedings of the IEEE International Conference on Real-Time Computing Systems and Applications*, 1999.