Reducing Stack with intra-task Threshold Priorities in Real-Time Systems *

Gang Yao and **Giorgio Buttazzo**
Scuola Superiore Sant'Anna,
Pisa, Italy
{g.yao, g.buttazzo}@sssup.it

Abstract

In the design of hard real-time systems, the feasibility of the task set is one of the primary concerns. However, in embedded systems with scarce resources, optimizing resource usage is equally important. In particular, the RAM is highly expensive in terms of chip space, and it heavily impacts the cost of the final product.

In this paper, we address the problem of reducing the stack usage of a set of sporadic tasks with timing and resource constraints, running on a uni-processor system. With respect to other approaches available in the literature, this work considers each task consisting of a set of functions (or subjobs), each characterized by a maximum stack requirement. This makes it possible to prohibit arbitrary preemptions through a dynamic priority protocol that reduces the overall system stack usage. Resource synchronization is also considered and, an extension of the Stack Resource Policy is presented to arbitrate the access to mutually exclusive resources while reducing the overall stack space. Simulations are performed on randomly generated task sets to evaluate the efficiency of the proposed method with respect to existing approaches.

1 Introduction

In a real-time kernel, it is well known that different scheduling policies lead to different system performance and different memory usage. For example, executing all the tasks in non-preemptive fashion can significantly reduce the overall stack usage, however it degrades task responsiveness and the total processor utilization. For this reason, most real-time operating systems (RTOSs) adopt fully preemptive schedulers, which reduce the latency of high priority tasks, but introduce higher runtime overhead and require a larger amount of memory. Many RTOSs (e.g., uC/OS-II, FreeRTOS, AvrX) allocate a dedicated stack space for each task, whereas others (such as Erika [1], Fusion [2], RTA-OSEK RTOS [3] etc.) allow several tasks to share a single stack space.

*This work has been partially supported by the European Community's Seventh Framework Programme FP7/2007-2013 under grant agreement no. 216008.

In classical schedulability analysis, the structure of the task code is typically neglected, so that any preemption pattern is assumed to be possible. This fact leads to very pessimistic results in terms of memory usage, since each preemption is supposed to occur when the current task is using its maximum stack space. Therefore, the total stack space that has to be allocated to the application is the sum of the maximum stack spaces required by all the tasks.

Sharing a common stack for a group of tasks is possible when the tasks in the group can execute in non-interleaved fashion, so one stack frame is sufficient for the entire group. For instance, the preemption thresholds methodology [25] allows designers to partition the system into task groups, where all tasks inside the same group are executed non-preemptively. In this case, the stack usage of each group is equal to the stack space needed by the task with the maximum requirement, and the overall stack usage is reduced to the sum of all the group stacks, rather than all the task stacks.

Modern system design methodologies and standards require high portability and reuses of functional components. For example, in AUTOSAR [4], used for the development of automotive systems, the application is developed as a group of software components, whose behavior is represented by a set of *runnables*, each implementing a specific functionality. Each runnable is then mapped to a task with a given priority, which is defined to enforce a desired order of execution, with respect to other runnables.

Recently, limited preemptive scheduling with non-preemptive regions has received increasing attention in the real-time community, for its possibility of combining the advantages of preemptive and non-preemptive scheduling [8, 26]. This framework not only is suitable for modelling tasks structured as a sequence of functions, but it also allows reducing the overall memory usage. For example, each runnable can be mapped to a function within a task, and treated as subjob with given memory and computational requirements. How mapping is performed is out of the scope of this paper (interested readers may refer to [13, 14] for details). The advantage of partitioning a task into subjobs is that at the subjob boundaries the stack usage is relatively low, hence a method can be proposed to schedule tasks so that the overall system stack is reduced.

Contributions of the paper This work provides three main contributions. First, a new task model is proposed to specify the structure of each task as a set of functions with specific timing and memory requirements. Preemption between subjobs is always allowed, whereas preemption within a subjob is allowed only when necessary to guarantee the feasibility of the task set. Second, a dynamic priority protocol is proposed to limit preemptions and reduce the overall stack usage, while meeting all task deadlines. Finally, the method is integrated with the Stack Resource Policy (SRP) [6] to handle shared resources.

Paper Organization The rest of the paper is organized as follows. Section 2 introduces the basic assumptions and methodology used in the paper, and presents a motivation example for the paper. Section 3 presents the proposed approach. Section 4 describes the integration with the SRP and how the scheduling policy can be implemented. Section 5 reports some simulation results. Section 6 presents some related work and the distinctions from prior results. Finally, Section 7 states our conclusions

and future work.

2 Task Model and Assumptions

The system consists of a set $\mathcal{T} = \{\tau_1, \tau_2, \ldots, \tau_n\}$ of n real-time tasks to be executed on a uniprocessor under fixed priority scheduling. Each task τ_i can be periodic or sporadic, where P_i denotes its nominal priority, D_i its relative deadline, C_i its worst-case execution time, and T_i its period or minimum inter-arrival time between two consecutive releases. A constrained deadline model is adopted here, so D_i is assumed to be less than or equal to T_i. Task priorities are assigned according to Rate Monotonic [21], if all relatives deadlines are equal to periods, or according to Deadline Monotonic [20], otherwise. Tasks are indexed by decreasing priority, and a larger value of P_i denotes a higher priority. For convenience, we also use the following notations to represent some task subsets:

$$\begin{cases} hp(i) = \{\tau_j |\ P_j > P_i\} \\ hep(i) = \{\tau_j |\ P_j \geq P_i\} \\ lp(i) = \{\tau_j |\ P_j < P_i\} \end{cases}$$

The system utilization U is defined as the sum of the utilization factors of all the tasks:

$$U = \sum_{i=1}^{n} U_i = \sum_{i=1}^{n} C_i/T_i.$$

In the model considered in this paper, each task τ_i consists of m_i functions (or subjobs), $\{F_{i,1}, \ldots, F_{i,m_i}\}$, where each function $F_{i,j}$ is characterized by a worst-case execution time $q_{i,j}$ and a maximum stack usage $s_{i,j}$. Therefore, the worse-case execution time of a task is the sum of all its subjob durations, that is, $C_i = \sum_{j=1}^{m_i} q_{i,j}$.

Since it is difficult to determine the exact stack usage at every point for a given task instance, it is assumed that each subjob gradually increases its stack usage up to its maximum level $s_{i,j}$, remains at that level for the entire subjob duration, and then releases the stack space at the end of the subjob code. This is the typical assumption adopted for the task stack usage in related research works [16, 17, 23, 24], which provides safe but pessimistic bounds. Moreover, a fixed stack $s_{i,0}$ is considered to be required by task τ_i between subjobs. Finally, it is assumed that the stack is only used between the start and the finishing time of each task instance, hence, no local data remains on the stack at the end of a task instance. Notice that this assumption complies with the OSEK standard [5] used in the development of automotive applications. The subjob division and mapping is beyond the scope of this paper, instead, the information about subjobs is used to perform the analysis and derive a protocol that reduces system stack.

To limit preemptions, each subjob $F_{i,j}$ is assigned a static priority level $P_{i,j} \geq P_i$, computed off line to limit stack usage while preserving schedulability. The priority $P_{i,j}$ is used during subjob execution as a *threshold priority* to prevent preemption from any task τ_h, with $P_i < P_h \leq P_{i,j}$. Between subjobs, task is executed using its nominal priority P_i. As a consequence, the actual priority of a task τ_i is not fixed, but varies at

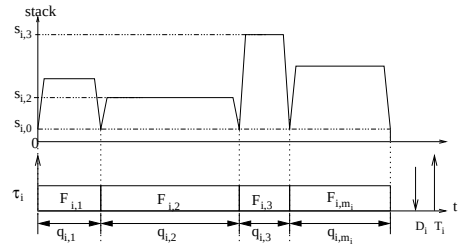

Figure 1: Example of task with different computational and stack requirements.

runtime depending on the portion of code under execution. The variable p_i is used to denote such a dynamic priority for task τ_i.

Figure 1 illustrates a sample task consisting of four subjobs. The maximum stack usage is also shown at each time instant. Notice that a stack equal to $s_{i,0}$ is needed between subjobs, as the previous subjob has released its stack space and the coming subjob has not yet started.

For the sake of convenience, the following parameters are also defined:

s_i^m denotes the maximum stack usage among all subjobs of task τ_i, that is

$$s_i^m = \max_{1 \le j \le m_i} \{s_{i,j}\}.$$

$\mathcal{S}_{i,j}$ denotes the maximum system stack usage when subjob $F_{i,j}$ is feasibility scheduled together with all tasks in $hp(i)$.

$\mathcal{S}_i$ denotes the maximum system stack required to feasibly schedule the subset $hep(i)$. Clearly, it can be computed as:

$$\mathcal{S}_i = \max_{1 \le j \le m_i} \{\mathcal{S}_{i,j}\}. \tag{1}$$

$\mathcal{S}$ denotes the maximum system stack required to feasibly schedule the whole task set:

$$\mathcal{S} = \max_{1 \le i \le n} \{\mathcal{S}_i\}. \tag{2}$$

2.1 Motivation example

Consider a simple task set of three periodic (or sporadic) tasks as reported in Table 1. The task parameters are given in the second column and the execution time and stack requirement of subjobs is in the third and forth column, respectively. Each task is assumed to consist of two subjobs ($m_i = 2, i = 1, 2, 3$) and $s_{i,0}$ is the same value for each task ($s_{i,0} = 1, i = 1, 2, 3$).

Under fully preemptive scheduling, the stack bound is equal to the sum of all task stack spaces ($\mathcal{S}^{\mathcal{FPS}}$= 18) and the system can be easily verified to be schedulable.

	C_i	D_i	T_i	s_i^m	$q_{i,1}$	$q_{i,2}$	$s_{i,1}$	$s_{i,2}$	β_i
τ_1	10	14	20	5	5	5	4	5	4
τ_2	4	30	30	7	2	2	5	7	6
τ_3	9	40	40	6	5	4	4	6	3

Table 1: A sample task set of 3 tasks.

Preemption threshold methodology[1] assigns the first two tasks (τ_1 and τ_2) to a non-preemptive group, thus allowing some stack saving ($\mathcal{S}^{\mathcal{PTS}} = \max\{s_1^m, s_2^m\} + s_3^m = 13$) while still guaranteeing the schedulability. Prohibiting preemptions can save the system stack usage but at the risk of deadline misses. The questions arise that with the additional information on the subjob requirement on timing and stack space, how can system further reduce the overall stack space with system schedulability still preserved and how to implement this method with limited efforts. These two questions will be investigated in the following two sections.

3 Reducing stack usage

This section presents the approach proposed in this paper for limiting the stack usage of the task set and computing the maximum stack requirements. To limit the growth of the stack, tasks are executed using limited preemptive scheduling. In fact, the non-preemptive execution of subjobs prevents a task from being preempted arbitrarily when using the maximum stack space, which would require a higher overall stack usage. However, non-preemptive regions increase the blocking delay on higher priority tasks, possibly jeopardizing the system feasibility. Taking into consideration both the two effects, the proposed method computes the highest priority level $P_{i,j}$ for each subjob $F_{i,j}$ that reduces the overall stack usage while guaranteeing the schedulability of the task set.

3.1 Main results on limited preemptive scheduling

Feasibility analysis under fixed priority scheduling can be performed using the *request bound function* [19] $\text{RBF}(\tau_i, t)$, which is defined as the maximal cumulative execution request that can be generated by jobs of τ_i within an interval of length t from the critical instant:

$$\text{RBF}(\tau_i, t) = \left\lceil \frac{t}{T_i} \right\rceil C_i.$$

The cumulative execution request of a task τ_i and $hp(i)$ over an interval of length t is therefore bounded by:

$$W_i(t) = C_i + \sum_{\tau_j \in hp(i)} \text{RBF}(\tau_j, t).$$

[1] Many valid task grouping with different objective are possible, the `OPT-Partition` algorithm in [24] is adopted here for comparison.

Definition 1. *The blocking tolerance β_i for task τ_i under fixed priority scheduling is defined as:*

$$\beta_i = \max_{t \in (0, D_i]} \{t - W_i(t)\}. \tag{3}$$

Notice that the right hand side of Equation (3) does not need to be evaluated at every instant $t \in (0, D_i]$, but only at those times in which $W_i(t)$ has a discontinuity, that is:

$$\{t \in [C_i, D_i] \mid t = k \cdot T_j,\ k \in \mathbb{N} \text{ and } \forall T_j,\ \tau_j \in hp(i)\}.$$

The β_i calculated in Equation (3) represents the maximal blocking that τ_i can suffer without violating the deadline constraint. Notice this value can also be used to verify the feasibility of the task set *under fully preemptive scheduling*: if all the computed β_i are non-negative, the task set is deemed feasible.

Theorem 1. *For a preemptively feasible task set, subjob $F_{i,k}$ of task τ_i can execute non-preemptively without causing any deadline miss in tasks $\tau_j \in hp(i)$ if*

$$\forall j = 1, \ldots, i-1 \qquad q_{i,k} \leq \beta_j. \tag{4}$$

Proof. By contradiction. Assume Equation (4) is satisfied and $\tau_j (j \in [1, i-1])$ has a deadline miss due to the non-preemptive execution of $F_{i,k}$. Since the maximum blocking time due to $F_{i,k}$ is $q_{i,k}$, for τ_j we have:

$$\forall t \in (0, D_j] \quad q_{i,k} + C_j + \sum_{\tau_l \in hp(j)} \text{RBF}(\tau_l, t) > t$$

which is equivalent to:

$$q_{i,k} > \max_{t \in (0, D_j]} \left\{ t - C_j - \sum_{\tau_l \in hp(j)} \text{RBF}(\tau_l, t) \right\}$$

According to the definition of blocking tolerance in Equation (3), we have:

$$q_{i,k} > \beta_j$$

which contradicts Equation (4) and proves the theorem. □

3.2 Proposed approach

Let us start considering a special case in which the task set is schedulable when all tasks are executed non-preemptively. In this case, there is only one active task at a time, and the maximum stack space required for the task set, denoted as $\mathcal{S}^{\mathcal{NPS}}$, is equal to the largest stack space required by each task, that is:

$$\mathcal{S}^{\mathcal{NPS}} = \max_{1 \leq i \leq n} \{s_i^m\}. \tag{5}$$

When preemption is allowed only between subjobs (that is, when all subjobs are executed in non-preemptive fashion), then only one subjob is active at any time instant.

Hence, the overall stack space, denoted as $\mathcal{S}^{\mathcal{NSJ}}$ (Non-preemptive SubJob), is no greater than:

$$\mathcal{S}^{\mathcal{NSJ}} = \sum_{i=1}^{n} s_{i,0} + \max_{1 \le i \le n} \{s_i^m - s_{i,0}\}. \tag{6}$$

It can easily be verified that $\mathcal{S}^{\mathcal{NPS}}$ is no larger than $\mathcal{S}^{\mathcal{NSJ}}$, since there is no extra $s_{i,0}$ needed for the system under non-preemptive scheduling.

If the non-preemptive execution of one subjob makes the task set infeasible, preemption must take place within that subjob and, consequently, the stack space required by the system may increase. The idea to limit the increase of the overall stack space is to allow preemptions inside a subjob only when strictly necessary, that is, when the non-preemptive execution of that subjob would cause other tasks to miss their deadlines. To implement this idea, each subjob $F_{i,j}$ is assigned a priority level $P_{i,j} \ge P_i$ that prevents preemption as much as possible. Notice that $P_{i,j}$ (referred to as subjob threshold priority) is a static value computed off line, but assigned dynamically at run time when subjob $F_{i,j}$ starts executing.

Given a subjob $F_{i,j}$ of task τ_i, let P_k $(P_i \le P_k \le P_1)$ be the highest priority level such that all tasks τ_h with $P_h \le P_k$ remain schedulable when $F_{i,j}$ is executed non-preemptively. Then, $F_{i,j}$ can be safely executed with priority $P_{i,j} = P_k$ with the system schedulability preserved. Notice that the schedulability of $lp(i)$ and τ_i are not affected by the non-preemptive execution of $F_{i,j}$, and the schedulability of tasks in $hp(i)$ can be verified using Theorem 1. Hence, $P_{i,j}$ can be expressed as follows:

$$P_{i,j} = \max\Big\{\{P_i\} \cup \{P_k \mid k \in [1,i), \forall h = k, \ldots, i-1 : q_{i,j} \le \beta_h\}\Big\}. \tag{7}$$

The following lemma states a property of the $\{\mathcal{S}_i\}$ sequence for the task set.

Lemma 1. *The sequence $\{\mathcal{S}_i\}$ $(1 \le i \le n)$ is non-decreasing.*

Proof. Suppose task subset $hep(i)$ is successfully scheduled with total stack usage $\mathcal{S}_i$. If task τ_i is removed from $hep(i)$, clearly, the remaining tasks are still feasible. Since the system stack usage cannot increase by removing one task, it follows that $\mathcal{S}_i \ge \mathcal{S}_{i-1}$. □

Let $\mathcal{S}_{hp}(P_k)$ denote the maximum stack used by tasks in $hp(k)$. Then, from Lemma 1, it follows that:

$$\mathcal{S}_{hp}(P_k) = \max_{1 \le i \le k-1} \{\mathcal{S}_i\} = \mathcal{S}_{k-1}. \tag{8}$$

And since τ_1 is the highest priority task in the system and it can never be preempted by other tasks, it follows that:

$$\mathcal{S}_{hp}(P_1) = \mathcal{S}_0 = 0. \tag{9}$$

Once the threshold priority $P_{i,j}$ is computed for each subjob $F_{i,j}$, the corresponding stack usage $\mathcal{S}_{i,j}$ can be computed by the following theorem, by considering the tasks that can preempt $F_{i,j}$.

Theorem 2. *Let $P_{i,j} = P_k,\ P_i \leq P_k \leq P_1$. Then, the max stack space needed for feasibly scheduling $F_{i,j}$ together with tasks in $hp(i)$ is:*

$$\mathcal{S}_{i,j} = \max\{s_{i,j} + \mathcal{S}_{k-1},\ s_{i,0} + \mathcal{S}_{i-1}\} \tag{10}$$

where $\mathcal{S}_0 = 0$ as in Equation (9)*.*

Proof. If $P_{i,j} = P_k$, $F_{i,j}$ can be preempted immediately only by tasks with priorities in $[P_1, P_{k-1}]$, while tasks with priorities in $[P_k, P_{i-1}]$ start executing only after the completion of $F_{i,j}$. When $F_{i,j}$ is immediately preempted, the overall stack is equal to the current stack ($s_{i,j}$) plus the increment due to preemptions ($\mathcal{S}_{hp}(P_k)$). According to Equation (8), it can be expressed as: $s_{i,j} + \mathcal{S}_{k-1}$.

When preemptions are deferred to the end of $F_{i,j}$, the stack usage of τ_i is reduced to $s_{i,0}$ and, the stack increment due to preemptions is $\max\{\mathcal{S}_l |\ k \leq l < i\}$. Therefore, according to Lemma 1 the stack usage can be expressed as: $s_{i,0} + \mathcal{S}_{i-1}$.

Hence, the maximum stack space required for executing subjob $F_{i,j}$ is the maximum between these two values. Hence, the theorem follows. □

Two additional notes regarding the proof of Theorem 2:

1. From the scheduling point of view, our proposed approach provides a compromise between two extreme cases: fully preemptive and non-preemptive subjob scheduling. When $P_k = P_i$, the task priority is always equal to the nominal one and the task instance is executed in fully preemptive mode. In this case, the stack usage $\mathcal{S}_{i,j}$ becomes $s_{i,j} + \mathcal{S}_{i-1}$, hence Equation (10) still holds. Whereas, when $P_k = P_1$ it corresponds to the special case in which subjob $F_{i,j}$ is executed non-preemptively. In this case, Equation (10) still holds as $\mathcal{S}_0$ is defined in Equation (9).

2. One special case is when $F_{i,j}$ is the last subjob of τ_i, then $s_{i,0}$ in Equation (10) can be avoided as this task instance ends. However, Equation (10) is used as the uniform expression and is still correct as an upbound of the stack usage.

It is worth pointing out that the stack increase due to mutual preemptions among tasks within $hp(i)$ is already accounted in the value of $\mathcal{S}_i$, as the computation is performed in decreasing priority order. Moreover, proceeding in this way, $\mathcal{S}_j$ for task $\tau_j \in hp(i)$ is available when considering task τ_i.

Since τ_1 is the highest priority task, all its subjobs will never be preempted, hence $\mathcal{S}_1 = s_1^m$. For the remaining tasks in the system, each value $\mathcal{S}_{i,j}$ $(1 \leq j \leq m_i)$ is calculated using Theorem 2. Then, the stack usage for the subset $hep(i)$ can be computed according to Equation (1).

Corollary 1. *The maximum system stack usage $\mathcal{S}$ is equal to the stack needed to feasibly schedule task τ_n together with the other tasks in $hp(n)$, that is*

$$\mathcal{S} = \mathcal{S}_n. \tag{11}$$

Proof. It directly follows from Equation (2) and Lemma 1. □

3.3 The algorithm implementation

Using the results presented above, this section describes how to compute the maximum stack space required by the system. A procedure `set_threshold(`i, j`)`, shown in Figure 2, is used to compute the threshold priority $P_{i,j}$ for subjob $F_{i,j}$. The main algorithm, `stack_size()`, is presented in Figure 3.

Procedure: set_threshold(i, j**)**
Input: Task set parameters.
Output: $P_{i,j}(1 < i \leq n, 1 \leq j \leq m_i)$.
begin
  $k = i$
  while $k > 1$ **do**
1   $k = k - 1$
2   **if** $q_{i,j} > \beta_k$ **then return** P_{k+1}
  end
  return P_1
end

Figure 2: Procedure for computing $P_{i,j}$.

The $while$ loop in Figure 2 scans all tasks with priority higher than P_i to verify whether they are feasible when $F_{i,j}$ is executed non-preemptively. According to Theorem 1, if $q_{i,j} \leq \beta_k$, task τ_k remains feasible and priority is raised to the next level, otherwise Line 2 returns the priority of the previous level P_{k+1}. If all tasks are found to be schedulable, then $P_{i,j}$ is assigned the highest priority P_1.

The algorithm in Figure 3 considers tasks in decreasing priority order, from τ_1 to τ_n. The values of $\mathcal{S}_0$ and $\mathcal{S}_1$ are initialized in Line 1. For each task $\tau_i(1 < i \leq n)$, subjobs $F_{i,j}(1 \leq j \leq m_i)$ are checked sequentially, by increasing index of j. In particular, $P_{i,j}$ is computed at Line 2 using the algorithm in Figure 2. Then, $\mathcal{S}_{i,j}$ is computed for subjob $F_{i,j}$ at Line 3, according to Theorem 2. The stack usage $\mathcal{S}_i$ is derived using Equation (1). In this way, Line 4 keeps the record of $\mathcal{S}_i$ and updates it when necessary. Finally, Line 5 returns $\mathcal{S}_n$ as the system stack usage, according to Corollary 1.

In terms of complexity, calculating the blocking tolerances requires pseudo-polynomial complexity, as showed in [9, 26]. Hence, the complexity of the `set_threshold` procedure is pseudo-polynomial. In the main algorithm, stack is computed task by task and, within each task, subjob by subjob, hence the complexity of this procedure is $O(n * m_i)$. Therefore, the overall complexity of the algorithm is pseudo-polynomial.

3.4 Motivation example revisit

The motivation example presented in Section 2 is revisited here to illustrate the proposed approach. The blocking tolerances β_i, computed using the algorithm in [26], are reported in the last column of Table 1.

The stack usage of τ_1 can be easily derived as $\mathcal{S}_1 = s_1^m = 5$. For τ_2, as $q_{2,1}$ and $q_{2,2}$ are both less than β_1, subjobs $F_{2,1}$ and $F_{2,2}$ can always execute non-preemptively

Algorithm: stack_size()

Input: Task set parameters.

Output: Maximum system stack usage $\mathcal{S}$.

begin

1 $\quad \mathcal{S}_0 = 0, \quad \mathcal{S}_1 = s_1^m$

$\quad$ **for** $i \leftarrow 2$ **to** n **do**

$\quad\quad \mathcal{S}_i = s_{i,1}$

$\quad\quad$ **for** $j \leftarrow 1$ **to** m_i **do**

2 $\quad\quad\quad P_{i,j}$ = set_threshold (i, j)

3 $\quad\quad\quad \mathcal{S}_{i,j} = \max\{s_{i,j} + \mathcal{S}_{hp}(P_{i,j}), s_{i,0} + \mathcal{S}_{i-1}\}$

4 $\quad\quad\quad \mathcal{S}_i = \max\{\mathcal{S}_i, \mathcal{S}_{i,j}\}$

5 $\quad \mathcal{S} = \mathcal{S}_n$

end

Figure 3: Algorithm for computing $\mathcal{S}$.

($P_{2,1} = P_{2,2} = P_1$). The maximum stack usage for $F_{2,1}$ is when τ_1 starts after the completion of $F_{2,1}$, hence,

$$\mathcal{S}_{2,1} = \max\{s_{2,1} + \mathcal{S}_0,\ s_{2,0} + \mathcal{S}_1\} = s_{2,0} + \mathcal{S}_1 = 6.$$

When τ_1 starts after $F_{2,2}$ finishes, $F_{2,2}$ does not have the stack usage of $s_{2,0}$ as current instance finishes. Hence,

$$\mathcal{S}_{2,2} = \max\{s_{2,2} + \mathcal{S}_0, \mathcal{S}_1\} \leq \max\{s_{2,2} + \mathcal{S}_0,\ s_{2,0} + \mathcal{S}_1\} = 7.$$

Therefore, the stack usage $\mathcal{S}_2$ is 7 (the maximum between $\mathcal{S}_{2,1}$ and $\mathcal{S}_{2,2}$). For task τ_3, since $q_{3,1} < \beta_2$ and $q_{3,1} > \beta_1$, we have $P_{3,1} = P_2$ and

$$\mathcal{S}_{3,1} = \max\{s_{3,1} + \mathcal{S}_1,\ s_{3,0} + \mathcal{S}_2\} = \max\{4 + 5,\ 1 + 7\} = 9.$$

Finally, being $q_{3,2} \leq \beta_1$ and $q_{3,2} < \beta_2$, $F_{3,2}$ can be executed non-preemptively and $P_{3,2} = P_1$, hence:

$$\mathcal{S}_{3,2} = \max\{s_{3,2} + \mathcal{S}_0, \mathcal{S}_2\} \leq \max\{s_{3,2} + \mathcal{S}_0, s_{3,0} + \mathcal{S}_2\} = 8.$$

and $\mathcal{S}_3 = \max\{\mathcal{S}_{3,1},\ \mathcal{S}_{3,2}\} = 9$. Therefore the maximum system stack usage is $\mathcal{S} = \mathcal{S}_3 = 9$.

A possible scheduling trace for the task set in Table 1 is reported in Figure 4, which also shows the system stack $\mathcal{S}$ and the dynamic priority of τ_3 as a function of time. Each subjob is represented by a different filling pattern. Notice that, since tasks are sporadic, all preemption patterns are possible, depending on the specific arrival times. At time $t = 12$, $p_3 = P_{3,1} = P_2$, hence subjob $F_{3,1}$ resumes after τ_1 finishes. As a consequence, $\mathcal{S}$ remains at $s_{3,1}$. At time $t = 15$, subjob $F_{3,1}$ ends and p_3 decreases to P_3, thus task τ_2 starts executing. Notice that, in this way, the stack increases from $s_{3,0}$, but not from $s_{3,1}$. The priority of τ_3 remains at P_3 for the entire execution of τ_2, then increases to $P_{3,2} = P_1$ at time $t = 19$, as $F_{3,2}$ starts executing.

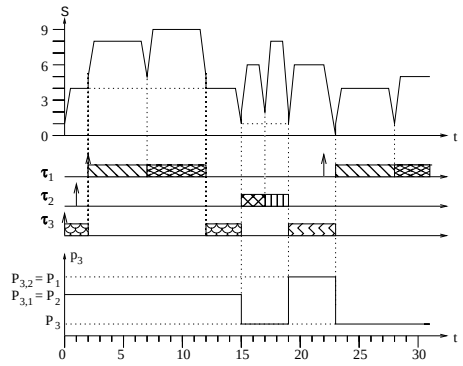

Figure 4: A sample task set with the stack usage and the dynamic priority of task τ_3.

Table 2 summarizes the maximum system stack usage and the result of schedulability test, when the task set is scheduled by different methods. In particular, $\mathcal{S}^{\mathcal{NPS}}$ denotes the stack bound computed under non-preemptive scheduling and, $\mathcal{S}^{\mathcal{NSJ}}$ the stack bound computed when preemption can occur only between subjobs. These two values can be computed by Equation (5) and (6), respectively. Notice that these two methods can achieve higher stack saving, but cannot make the task set schedulable. The maximum stack usage produced by our proposed approach is represented as $\mathcal{S}$. It achieves the lowest system stack usage among the listed methods with the schedulability still preserved.

	$\mathcal{S}^{\mathcal{NPS}}$	$\mathcal{S}^{\mathcal{NSJ}}$	$\mathcal{S}$	$\mathcal{S}^{\mathcal{PTS}}$	$\mathcal{S}^{\mathcal{FPS}}$
System Stack	7	9	9	13	18
Schedulability	N	N	Y	Y	Y

Table 2: Results of different methods.

The performance of these methods is further investigated by extensive simulations reported in Section 5.

4 Integrating with Stack Resource Policy (SRP)

The proposed approach can be easily extended to work under shared resources, provided that each critical section is confined within one subjob. In fact, if SRP is used to access critical sections, the limited preemptive method used to reduce stack can be easily implemented by introducing a set of *pseudo-resources*.

Tasks may share a set of r (real) mutually exclusive resources $\rho^1, \rho^2, \ldots, \rho^r$, which are accessed and released within subjobs, thus no resources are locked between subjobs. Let $\omega_{i,j}^k$ be the length of the longest critical section of resource ρ^k in subjob $F_{i,j}$. No knowledge is assumed on the position of critical sections within subjobs, so the start time of critical sections is unknown to the designer.

Under SRP, each task τ_i is assigned a static preemption level π_i and, under fixed priority scheduling, preemption levels can be set to the nominal priorities ($\pi_i = P_i$). SRP guarantees that each job is blocked at most once, at the time it attempts to preempt. The maximum blocking time for task τ_i under SRP can be calculated as the longest critical section $\omega_{l,j}^k$ accessed by tasks with lower preemption levels and with ceiling higher than or equal to the preemption level of τ_i:

$$B_i = \max_{\forall \tau_l \in \mathcal{T}, \forall j,k} \{\omega_{l,j}^k |\ \pi_i > \pi_l \cap \pi_i \leq ceil(\rho^k)\}. \tag{12}$$

In the approach proposed in this paper, $P_{i,j}$ has a strong similarity with the ceiling of a resource. When $F_{i,j}$ starts, the tasks having priorities lower than or equal to $P_{i,j}$ cannot start execution *as if* a resource were locked and the system ceiling were raised. After $F_{i,j}$ ends, tasks blocked by $F_{i,j}$ can start executing *as if* the resource were released and the system ceiling were reduced to the previous value.

This is equivalent to assuming that each subjob $F_{i,j}$ accesses a pseudo-resource $\rho_{i,j}$ with length equal to $q_{i,j}$ and static ceiling equal to $P_{i,j}$. Such a pseudo-resource is locked when $F_{i,j}$ starts executing and unlocked when $F_{i,j}$ ends. The system ceiling is also updated accordingly, considering the other resources. Notice that, the real local resources ρ^r are shared among the tasks in the system and the resource ceiling is decided accordingly, depending on which tasks access this resource, whereas, the pseudo-resource $\rho_{i,j}$ is added only to one subjob, with the ceiling computed off line according to Equation (7).

Such an extension is referred to as SRP with fixed subjob division (*SRP-F*). Under SPR-F, the *system ceiling* is computed as

$$\Pi_s = \max_{j \in [1,n]} \left\{ \left\{ \max_{i \in [1,r])} \{ceil(\rho^i)\} \right\} \cup \{p_j | \tau_j \text{ is active}\} \right\}$$

where the first part represents the ceiling of the real shared resources while second part the pseudo-resource, depending on which subjobs are active. The pseudo-resources added in SRP-F are compatible with the traditional SRP resources, hence, all properties under SRP are automatically retained under SPR-F.

Since we introduce another set of pseudo-resources, each task has more chance to be blocked. Indeed, now blocking may occur for two reasons: i) the real local resources and ii) the pseudo-resources. In the first case, Equation (12) still holds and the feasibility can be verified using the methods introduced in [6]. In the second case, the blocking can be calculated in a similar way, and the ceiling for each pseudo-resource is computed in such a way that the system feasibility is always guaranteed.

5 Simulation Results

This section presents some experimental results performed on synthetic task sets to evaluate the effectiveness of our algorithm with respect the ones listed in Table 2. The proposed method is referred to as SRP-F.

5.1 Simulation setup

In order to simplify the simulation, each task τ_i is assumed to have the same number of subjobs and the same minimum/maximum stack usage, that is, $\forall i \in [1, n]$, $m_i = m$, $s_i^m = s^m$ and $s_{i,0} = s_0$. Moreover, another parameter, $\alpha = s^m/s_0$, has been introduced to represent the level of fluctuation on task stack usage. In all simulations, s^m is set to 1024 and s_0 is computed according to the value of α.

Subjob computation times $q_{i,j}$ are generated as a random numbers uniformly distributed in [100,500], then tasks computation times are derived by summing the subjob durations. Each subjob stack $s_{i,j}$ is randomly distributed within $(s_0, s^m]$ and then the largest one is set to s^m so that the maximum stack usage is guaranteed to be used.

Individual task utilization factors U_i are generated using the UUniFast algorithm [10] given a desired total utilization U. Task periods are computed as C_i/U_i and then deadlines are generated as random integers within $(C_i, T_i]$. Finally, the unfeasible task sets under fully preemptive mode are discarded. In all graphs, each point represents the average value over 1000 runs.

5.2 Exp 1: Overall performance of SRP-F

In a first experiment, the average system stack usage of the proposed algorithm has been monitored as a function of α and U. α was varied from 2 to 20 in step of 2, and U from 0.4 to 0.85 in step of 0.05. The task set includes ten tasks ($n = 10$), and each task has four subjobs ($m = 4$).

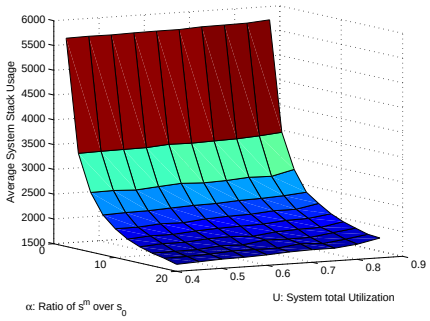

Figure 5: Stack of SRP-F when $n = 10$ and $m = 4$.

As clearly showed in Figure 5, the proposed algorithm can significantly reduce the system stack usage when α increases. In fact, higher values of α denote higher stack fluctuations due to lower stack usage between subjobs (s_0). The stack usage slightly increases with the system utilization, as tasks have more chances to be preempted, but the increment is not significant.

The following experiments compare `SRP-F` against `PTS` and `NSJ` for different parameters values. Notice that `FPS` and `NPS` are not reported in the graphs, because the corresponding required stacks can easily be computed as $\mathcal{S}^{\mathcal{FPS}} = n * s^m$ and $\mathcal{S}^{\mathcal{NPS}} = s^m$, respectively.

5.3 Exp 2: Comparing stack usage

In a second experiment, SRP-F was compared against PTS and NSJ as a function of the total system utilization U, when $n = 8$, $m = 5$ and $\alpha = 10$. Results are shown in Figure 6.

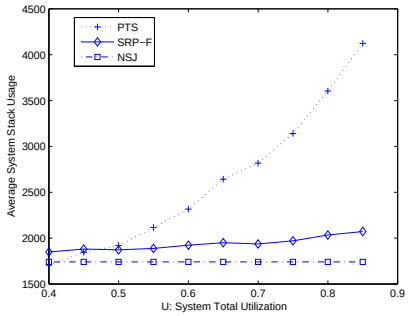

Figure 6: Average system stack at different U when $n = 8$, $m = 5$ and $\alpha = 10$.

As clearly shown in the graph, stack required by PTS increases with the system utilization at a much faster speed with respect to SRP-F. Also notice that, for small utilizations ($U < 0.5$) PTS requires less stack than SRP-F, because there is more chance to group tasks into a small number of non-preemptive groups, thus saving some unnecessary s_0. For high workloads, however, to keep the task set schedulable, PTS must increase the number of groups (each group requiring s^m), while SRP-F can allow preemptions at a lower cost s_0. Notice that NSJ is not affected by U, since preemptions are always allowed between subjobs. Remember, however, that NSJ may not be able maintain the task set schedulable.

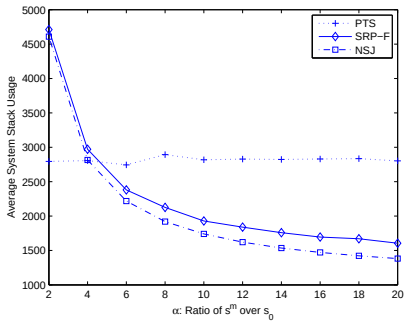

Figure 7: Average system stack at different α when $n = 8$, $m = 5$ and $U = 0.7$.

Figure 7 shows the stack usage as a function of α when $U = 0.7$, $n = 8$, and $m = 5$. Notice that PTS is not affected by α, whereas both SRP-F and NSJ significantly reduce the stack space for large values of α, taking advantage of the lower value of

s_0 with respect to s^m. Although NSJ is able to save more stack space, the task set may not be always schedulable, while SRP-F can guarantee the system feasibility at a reasonable cost of extra stack usage.

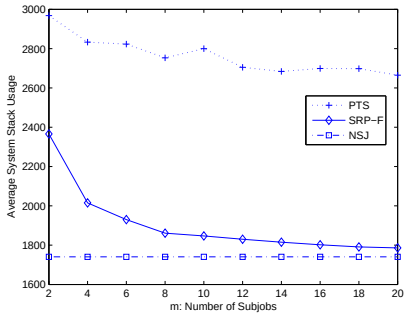

Figure 8: Average system stack at different m when $n = 8$, $\alpha = 10$ and $U = 0.7$.

Figure 8 shows the results when the number of subjobs (m) was changed from 2 to 20, keeping n, α and U at fixed values. As shown in the graph, SRP-F reduces the stack significantly when m increases. This is because larger values of m imply shorter subjob durations, with respect to task computation times, thus higher chances to be executed non-preemptively. Since preemptions between subjobs only require small stack overhead, SRP-F achieves a low overall stack. As usual, NSJ is able to achieve better stack saving, but at the cost of missing deadlines. Notice that PTS slightly reduces the system stack for larger m, as tasks tend to have similar computation times.

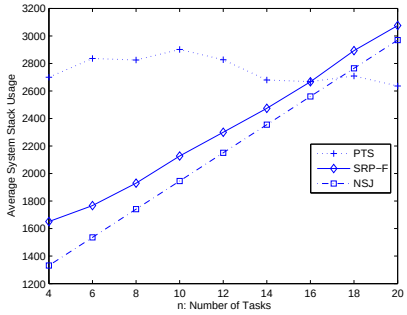

Figure 9: Average system stack at different n when $m = 6$, $\alpha = 10$ and $U = 0.7$.

In the last simulation, n was varied from 4 to 20, whereas m, α and U were kept fixed. The results plotted in Figure 9 show that, under SRP-F and NSJ, the stack usages almost increase linearly with n, since both algorithms suffer the s_0 stack overhead for each task. PTS is less sensitive to n, since the number of non-preemptive groups increases very slowly with the task set size, especially for low utilizations.

Taking into account all the presented results above, our proposed method SRP-F

shows its superiority especially when at large value of α and m. Indeed, in this case, subjobs have relatively short computation requirement and hence, the preemptions are deferred to the end of subjobs, where stack usage is quite low due to large value of α. Another interesting result is that system stack usage under SRP-F increases relatively slowly with utilization. Finally, when the system has a large number of tasks, PTS has good performance since the number of non-preemptive groups grows very slowly. However, in this case, SRP-F can provide high responsiveness as the blocking for a task is the maximum subjob length from lower priority tasks, instead of the task length under PTS. Actually, now the problem becomes a design choice, should more function blocks be mapped into one task or not?

As illustrated by the motivation example, NSJ (and NPS) cannot always guarantee the schedulability of the task set. The exact feasible ratio under NSJ is not the main focus of this paper, and it heavily depends on how the task set is generated, and especially on the deadlines. Also, NSJ can be considered as a very special case of SRP-F, occurring when all subjob threshold priorities are equal to the highest possible level. Therefore, the unschedulable ratio under NSJ can be qualitatively illustrated by the gap between the NSJ and SRP-F curves. For example, in Figure 8 and 9, when the value of m or n increases, NSJ has a larger chance to successfully schedule the task set and the two curves become closer.

6 Related Work

Preemption threshold scheduling (PTS) was first introduced by Express Logic Inc. in their real-time kernel ThreadX and then investigated academically by Wang and Saksena [25]. According to this method, each task has two priority levels: a nominal priority and a preemption threshold, used during task execution. Then, preemption is allowed to take place only when the nominal priority of an arriving task is higher than the threshold of the running task. In this way, a task has the possibility of disabling preemption up to a given priority level, equal to its threshold. The authors also proposed an algorithm to assign the optimal preemption thresholds and priorities for a given task set.

Based on the idea of a dual-priority scheme, Saksena and Wang [24] and Davis et al. [12] independently addressed the problem of reducing the stack space required by the task set. Tasks are partitioned into groups, within which tasks are not allowed to preempt each other, so that the stack required by each group is equal to the maximum stack requirement of the tasks within the group. In this way, the overall stack required by the task set is given by the sum of the stacks required by the groups. In particular, Saksena and Wang [24] also proposed an algorithm that minimizes the number of groups, showing that, for task sets with random attributes, the number of groups increases much more slowly than the number of tasks.

Unfortunately, Gai et al. [15, 16] showed that minimizing the number of groups does not minimize the overall stack usage. The authors also presented an algorithm, having exponential complexity in the worst case, to minimize the total amount of required stack.

Regehr [23] introduced two novel scheduling abstractions – task clusters and task

barriers – that provide better timing robustness. In his work, the execution time variance was also investigated and an algorithm to find fault-tolerant preemption thresholds was presented. Ghattas and Dean [17] proposed a unified framework using preemption thresholds under both static and dynamic scheduling. Their method allows optimizing the selection of preemption thresholds and reducing the average worst-case response times, for systems with limited memory and reduced stack.

While most of the approaches are built upon the dual-priority scheme, there exist other methods that focus on reducing the stack usage. Middha et al. [22] proposed the multi-task stack sharing technique (MTSS) that grows the stack of a particular task into other tasks in the system. Hänninen et al. [18] presented an approximate stack analysis method to derive a safe upper bound on the shared stack usage, in offset-based hybrid (interrupt- and time-driven) fixed priority preemptive systems. Finally, Bohlin et al. [11] introduced some techniques to exploit precedence and offset relations between tasks, thus limiting preemption patterns and bounding the amount of shared-stack usage.

In all the papers considered above, the internal structure of the tasks is never considered in the analysis, and the stack level is always assumed to be equal to its maximum size when a task is preempted. Conversely, to comply with the common industrial practice for modular software development, this work assumes that each task consists of a sequence of functions, with given computational and stack requirements. Then, exploiting the results achieved on limited preemptive scheduling, this work presents a scheduling protocol for limiting the preemption among tasks and reducing the overall system stack usage.

The scheduling protocol proposed in this paper is integrated with the Stack Resource Policy (SRP) [6], proposed by Baker to bound priority inversion among real-time tasks that share mutually exclusive resources. Gai et al. [16] showed how SRP can be extended to support preemption thresholds, with all properties of SRP still preserved. Baruah [7] and Bertogna et al. [9] considered the problem of reducing the resource holding times under EDF+SRP, by raising the system ceiling either statically or dynamically during runtime. In this paper, SRP is extended to limit preemption within subjobs as much as possible, to reduce stack usage while preserving schedulability.

6.1 Distinctions from prior results

The idea of *blocking tolerance* was first used in [9] and [26] with the purpose of bounding the maximal length of the non-preemptive regions for each task. In this paper, however, the blocking tolerance is used to compute the highest priority at which each subjob can run, to limit preemptions and stack usage. As showed by the example in Table 1, subjob $F_{3,1}$ can only be preempted immediately by task τ_1, but not by τ_2. The algorithm in this paper is more similar to [9], however, in that work the main objective was to compute and minimize the resource holding times, rather than stack size. Moreover, that work considered critical sections inside the task code, rather than subjobs.

A similar approach is also used in [8], which takes into account both the preemption related delays and system schedulability. The strategy adopted in that paper is to make each task chunk totally non-preemptive, by statically inserting potential preemption points inside the code. Our paper differs from that work in several ways. In our task

model, the subjob division comes from the system functional design, as input to our algorithm. Hence, we do not manually add preemption points inside the code, instead, we compute the maximum priority each subjob can raise.

The subjob threshold priority shares similarities with the work on preemption thresholds [24, 25], however, these two differ both in the valid range and the values. With subjob threshold priority, the task instance has a varying priority during its execution, depending on which subjob is executing, and the priority will return to the nominal one between subjobs. On the contrary, preemption threshold is valid for the entire task instance, from its start time to its completion. The group-based task set partition strategy, adopted in [16, 23, 24], cannot be applied here in a straightforward way. In fact, the longest subjob imposes the strongest constraint on the timing requirement and the maximum subjob stack directly affects the final optimization result. Since these two subjobs may not be the same, the problem is clearly combinatorial, thus requiring high computational complexity.

7 Conclusions

This paper presented an approach for reducing the stack usage of a set of real-time sporadic tasks running on a uniprocessor system under fixed priority scheduling. With respect to other approaches available in the literature, this work considered each task consisting of a set of functions (or subjobs), each characterized by a maximum stack requirement. This made it possible to prohibit arbitrary preemptions through a dynamic priority protocol, which rises the priority of each subjob to the maximum possible level to reduce the the system stack, while still guarantee feasibility. Resource synchronization was also considered and an extension of the Stack Resource Policy, called SRP-F, was presented to arbitrate the access to mutually exclusive resources while reducing the overall stack space.

The idea of splitting each task into a sequence of subjobs with different stack requirements allows taking advantage of internal stack fluctuations, trying to prohibit preemptions within a subjob, unless this is necessary for guaranteeing the schedulability of the task set. Simulation results show that, for most task set parameters, SRP-F achieves much smaller stack usage with respect to preemption threshold scheduling (PTS). Non-preemptive scheduling (NPS) and non-preemptive subjob scheduling (NSJ) have also been considered in the experiments, but only as references, since they cannot always guarantee the feasibility of the task set.

There are also particular situations in which PTS achieves more stack saving with respect to SRP-F. For example, when the stack fluctuation is not high (small values of α), the stack s_0 required between subjobs has a large influence on SRP-F, thus degrades the performance, whereas PTS is not affected by α. Also, when the task set utilization is low ($U < 0.5$) or the number of tasks is high ($n > 15$), task computation times become small and PTS has more chances to reduce the number of groups, so saving more stack than SRP-F.

In conclusion, simulation results suggest that, by analyzing the characteristics of the task set, it is possible to select the algorithm that achieves the smallest stack usage.

As a future work, we plan to extend the proposed methodology to multi-core sys-

tems and apply it to industrial applications to precisely measure how much memory space can be saved in real embedded systems.

References

[1] Evidence Srl. Web page: `http://www.evidence.eu.com`.

[2] Unicoi Systems. Web page: `http://www.unicoi.com`.

[3] ETAS Group. Web page: `http://www.etas.com`.

[4] AUTOSAR Consortium. Web page: `http://www.autosar.org`.

[5] OSEK Group. Web page: `http://www.osek-vdx.org`.

[6] T. P. Baker. Stack-based scheduling of real-time processes. *Real-Time Systems*, 3(1):67–100, March 1991.

[7] S. K. Baruah. Resource sharing in edf-scheduled systems: A closer look. In *RTSS '06: Proc. of 27th Real-Time Systems Symposium*, pages 379–387, 2006.

[8] M. Bertogna, G. Buttazzo, M. Marinoni, G. Yao, F. Esposito, and M. Caccamo. Preemption points placement for sporadic task sets. In *ECRTS '10: Proc. of Euromicro Conf. on Real-Time Systems*, July 2010. to appear.

[9] M. Bertogna, N. Fisher, and S. Baruah. Resource holding times: computation and optimization. *Real-Time System*, 41(2):87–117, 2009.

[10] E. Bini and G. C. Buttazzo. Measuring the performance of schedulability tests. *Real-Time System*, 30(1-2):129–154, 2005.

[11] M. Bohlin, K. Hanninen, J. Maki-Turja, J. Carlson, and M. Nolin. Bounding shared-stack usage in systems with offsets and precedences. In *ECRTS '08: Proc. of Euromicro Conf. on Real-Time Systems*, pages 276–285, July 2008.

[12] R. Davis, N. Merriam, and N. Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proc. of Industrial Experience Session, Euromicro Conf. on RealTime Systems*, pages 43–50, 2000.

[13] M. Di Natale. Optimizing the multitask implementation of multirate simulink models. In *RTAS '06: Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium*, pages 335–346, 2006.

[14] A. Ferrari, M. D. Natale, G. Gentile, G. Reggiani, and P. Gai. Time and memory tradeoffs in the implementation of autosar components. In *DATE '09: Proc. of Design, Automation and Test in Europe*, pages 864–869, 2009.

[15] P. Gai. *Real-Time Operating System design for Multiprocessor system-on-a-chip*. PhD thesis, SSSA, Pisa, Italy, 2004.

[16] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *RTSS '01: Proc. of Real-Time Systems Symposium*, pages 73–83, 2001.

[17] R. Ghattas and A. G. Dean. Preemption threshold scheduling: Stack optimality, enhancements and analysis. In *RTAS '07: Proc. of the 13th IEEE Real Time and Embedded Technology and Applications Symposium*, pages 147–157, 2007.

[18] K. Hanninen, J. Maki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Determining maximum stack usage in preemptive shared stack systems. In *RTSS '06: Proc. of the 27th Real-Time Systems Symposium*, pages 445–453, 2006.

[19] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *Proc. of the Real-Time Systems Symposium*, pages 166 – 171, CA, USA, Dec 1989.

[20] J. Leung and J. Whitehead. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation*, 2(4):237–250, 1982.

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

[22] B. Middha, M. Simpson, and R. Barua. Mtss: Multitask stack sharing for embedded systems. *ACM Trans. Embedded Computing System*, 7(4):1–37, 2008.

[23] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. of the 23rd IEEE Real-Time Systems Symposium*, pages 315–326, 2002.

[24] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *RTSS '00: Proc. of Real-Time Systems Symposium*, pages 25–34, 2000.

[25] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 328–335, 1999.

[26] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proc. of Conf. on Embedded and Real-Time Computing Systems and Applications*, pages 351–360, China, 2009.