Optimizing Stack Memory Requirements for Real-time Embedded Applications

Haibo Zeng
McGill University, Canada

Marco Di Natale
Scuola Superiore Sant'Anna, Italy

Qi Zhu
University of California at Riverside, USA

Abstract

In the development of some real-time embedded applications, especially systems-on-chip, an efficient use of RAM memory is as important as the effective scheduling of the computation resources. The design problem is to find a schedulable solution that fits within the memory budget. In a real-time concurrent system, preemption plays an important role in the exploration of these tradeoffs. Several schemes, including preemption thresholds and non-preemption groups, have been developed to improve schedulability and saving stack memory space by selectively disabling preemption. However, the design synthesis problem for such systems and protocols is still an open problem. We target the efficient assignment of the scheduling parameters for systems scheduled according to these policies in several cases of practical interest, including those that are compliant with automotive standards.

1 Introduction

In systems-on-chip and also, in general, many embedded systems, availability of RAM is a major constraint because of issues related to the hardware fabrication technology. Also, in many systems, and especially in the automotive domain where the interchange of components and integration across the supply chain are major requirements, compliance with standards is mandatory.

In this paper, we target task scheduling and stack management policies, as implemented by the operating system. We discuss the case of fixed-priority scheduling and several protocols for limiting preemption, including *preemption thresholds* and *non-preemption groups*. These choices are driven by practical concerns. In the automotive domain, the standard that applies to operating system mechanisms and API is AUTOSAR [1] [17]. In its current specification, AUTOSAR has conformance classes for fixed-priority scheduling and time-triggered scheduling but not for dynamic-priority scheduling (such as Earliest Deadline First). Also, AUTOSAR supports the concept of Internal Resources, which allow the definition of non-preemption groups and, to some degree, the preemption threshold mechanism. Direct support for preemption thresholds is made available by the ThreadX kernel [8]. In practice, an (approximate) application-level implementation only requires an API call for changing the task priority at runtime. In this paper, we only consider single-CPU systems.

The behavior of SW components is represented by a set of *RunnableEntities* (or simply *runnables*): functions executed in response to events, such as periodic timer activations, data writes or explicit call requests. Runnables are atomic schedulable units, communicating with each other over *ports*. A mapping relation is defined between runnables and tasks, meaning that the runnable code is executed in the context of the task. Given the runnables, the designer's problem is to define the task model and map runnables into tasks. Then, priorities must be assigned to tasks to ensure the schedulability of the system. Preemption can be selectively disabled using the mechanisms allowed by the operating system to reuse as much as possible the stack space and possibly improve schedulability.

2 System Model

The system model consists of the functional model and the task model. In the functional model (also denoted as $\mathcal{F}$), the functions (runnables) of component Γ_k are denoted as ρ_i^k, or simply by a single index as ρ_i (the reference to the owner component is not needed for the analysis). The worst-case execution time of ρ_i is γ_i. In this work, we restrict to runnables that are activated in response to periodic timer events. Therefore, we associate to each runnable ρ_i a period θ_i. Each runnable is characterized by a worst-case stack requirement σ_i (in bytes) needed for its execution, and a deadline δ_i. Also, each runnable might be associated with a preemption threshold ψ_i and possibly a non-preemption group η_i (their meaning is explained later).

The implementation of runnables into tasks generates the task model $\mathcal{T}$. $\mathcal{T} = \{\tau_1, \ldots, \tau_l\}$ is the set of *tasks* in the system. Each task τ_j has a priority p_j (the higher the number, the higher the priority) and an activation period t_j. Each task is also characterized by a worst-case execution time c_j, a deadline d_j, a quantity of memory that is required for the stack space s_j, and either a preemption threshold y_j or a non-preemption group g_j.

A mapping relation $m(\rho_i, \tau_j, k)$ may be defined between a runnable ρ_i and a task τ_j, This mapping relation also defines a static scheduling (execution order) of the runnables inside the task, meaning that the code implementing the runnable ρ_i is executed in the context of τ_j in the k-th order ($k-1$ other runnables have an execution order smaller than

ρ_i in τ_j). The runnable with an execution order of k in task τ_j is also labeled as $\rho_{j,k}$. A mapping relation is only possible if the execution rate of ρ_i and τ_j are such that $l \times t_j = \theta_i$ for some integer l. The set of runnables mapped into τ_j is also denoted as $\mathcal{M}_j$. Runnables must be mapped in such a way that ordering relations (resulting from functional constraints) are preserved. Please note that runnables do not have a priority level, but only a preemption threshold level (possibly inherited from their non-preemption group).

Name	**WCET**	**Period**	**Dline**	**Stack reqmt**	**Prio.**	**Thresh.**	**NP group**
Runnables							
ρ_i	γ_i	θ_i	δ_i	σ_i		η_i	ψ_i
Tasks							
τ_j	c_j	t_j	d_j	s_j	p_j	y_j	g_j
Mapping of ρ_i into τ_j and constraints							
$m(\rho_i, \tau_j, k)$	$c_j = c_{j,0} + \sum_i \gamma_i, \theta_i = l \times t_j, s_j = s_{j,0} + \max_i \sigma_i$						

Table 1. Notations for runnables and tasks

If the task set is derived from a functional model, then some of the task parameters may be computed from the parameters of the runnables. For example, $c_j = c_{j,0} + \sum_k \gamma_{j,k}$, where $c_{j,0}$ is the computation time required for the task main function calling the runnables (setting up the calls to the runnable functions, forwarding data and event to the runnables in the task and/or other tasks). Of course, this is only an approximation of the relationship linking c_j to the $\gamma_{j,k}$, whose exact form is very complex due to factors such as cache dependencies. With a much better approximation, for the stack size we have $s_j = s_{j,0} + \max_k \sigma_{j,k}$.

State of the art

The two main mechanisms for limiting preemption in a controlled way are the *preemption thresholds* and the *non-preemption groups*. The definition of *preemption thresholds* is provided first in [12] to improve schedulability of real-time tasks. This mechanism provides a flexible way of limiting preemption and spans from fully-preemptive to non-preemptive scheduling, subsuming these two extremes. In preemption threshold, a task has two priority levels: a nominal priority, and a threshold priority that is assumed as soon as the task starts its execution and retained until it terminates. At runtime, a task is allowed to preempt another only if its priority is higher than the threshold of the task in execution. When the definition applies to runnable thresholds, the task executes at its priority level, but as soon as it starts executing a runnable, its preemption threshold level matches the one of the runnable, and is restored to the task nominal priority when the runnable ends. The worst-case response time of tasks with preemption threshold can be computed using the formula in [10]. In [12], two algorithms are proposed to assign priority levels and preemption threshold levels to improve system schedulability. One of them is a branch and bound algorithm, which can explore exhaustively (with pruning) all the possible feasible priority assignments and, for each of them, the highest preemption threshold value that can be assigned to each task. This algorithm is optimal in the sense that, if there is a feasible assignment of priority and preemption thresholds, it will find it. Although the optimization of the memory used for the stack space was not one of the algorithm objectives, the authors proposed a branch and bound algorithm to explore all possible priority assignments and the optimal threshold assignment for each of them.

The preemption threshold mechanism was extended in [11] by targeting a specific problem instance, in which the jobs to be scheduled with their preemption thresholds are mapped for execution into threads. Two jobs mapped into the same thread cannot preempt each other and, therefore, this mechanism effectively partitions jobs into non-preemption groups. This model is equivalent to a functional model in which the jobs/runnables need to be mapped into tasks. The stack required is then typically obtained as the sum of the maximum of the stack requirements for each group. The authors also proposed an algorithm for mapping jobs into threads so that the scheduling behavior does not change and the number of threads is minimized.

The *non-preemptive groups* model makes use of a similar but not equivalent concept (as shown in [5]). In this model, tasks are partitioned into non-preemptive groups. Tasks belonging to the same group cannot preempt each other. This can be simply implemented by having all tasks belonging to the same group share a virtual resource that is locked with the first instruction of each task and released with the last line of code. This mechanism practically requires that each group is associated with a runtime priority level (ceiling or preemption threshold) equal to the highest priority among the tasks in the group. Each task in the group assumes the group threshold priority when it starts its execution. This ceiling priority also prevents mutual interleaved executions of tasks belonging to different groups (in an ABAB pattern, as opposed to fully nested). Such an implementation model is supported by the OSEK OS standard [2].

Both scheduling models have been considered in [5] in the context of multiprocessor systems, where the authors pointed out an analogy between the definition of a preemption threshold level and the ceiling priority of a task executing a critical section protected by the Priority Ceiling protocol. This allows an extension of the mechanism to dynamic priority assignment schemes (such as EDF). In the same paper, the authors presented a task grouping algorithm (of exponential complexity) that finds the feasible group configuration with minimum requirements of stack memory, given a priority assignment and a preemption threshold assignment computed according to [12].

Task clusters and task barriers, are introduced in [10] for better robustness. In [6] a unified framework for static and dynamic priority systems with the definition of preemption thresholds is presented. The authors demonstrate that the algorithm in [11] for the assignment of the highest possible preemption thresholds *after priorities are assigned to tasks* is also optimal with respect to stack usage (again, with the assumption of a given priority assignment). When scheduling offsets are known, they can be exploited to further improve the analysis and the definition of the threshold levels,

as discussed in [7] [3]. Also, [15] provides an analysis of the use of non-preemptive regions for the purpose of improving schedulability (considering scheduling overheads).

In [13] a functional model is considered in which runnables are already mapped into tasks and the priority of the tasks is given, and preemption thresholds can be assigned to runnables. The maximum amount of blocking that can be tolerated by each runnable is computed and the stack requirement is minimized by iteratively increasing the preemption threshold of the runnables as much as possible, starting from those belonging to the second-highest priority task, as long as the task set remains schedulable, with an algorithm very similar to the one in [11]. However, the feasibility of the runnables and the tasks is computed with the pessimistic assumption that the task is preemptive. In this paper, we show how a better estimate of the worst case blocking time can be obtained using the approach in [16].

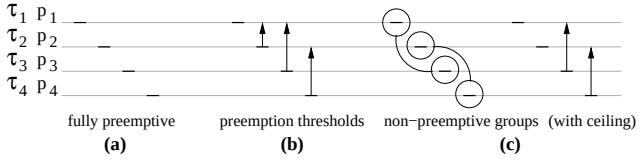

Figure 1. Example of preemption thresholds and non-preemptive groups.

Figure 1 shows an example consisting of four tasks in priority order (case (a)), with a possible definition of preemption thresholds, in (b) where the endpoints of the arrow indicate the threshold priority at runtime, a definition of non-preemptive groups in (c), with the corresponding threshold or ceiling priorities for the tasks. The assignment of a ceiling priority to a group is required if the benefits on the use of the stack space are to be retained. Otherwise, in the case of figure (c), τ_2 may still execute while the context of τ_4 is active, by having τ_3 preempt τ_4 and then τ_2 preempt τ_3. Please note that no definition of groups is equivalent to the scheduling constraints imposed by the thresholds in (b). The stack required for the execution of the tasks is

$$\begin{aligned}
&\text{(a) } \mathcal{S} = s_1 + s_2 + s_3 + s_4 \\
&\text{(b) } \mathcal{S} = \max\{s_4 + s_1, s_3, s_2\} \\
&\text{(c) } \mathcal{S} = \max\{s_1 + s_4, s_3, s_2 + s_1\}
\end{aligned}$$

Please note that many problems are open to solutions that improve on a possibly exhaustive branch and bound algorithm [12], to compute the priorities and preemption thresholds to be assigned to tasks/runnables so that they are feasible and the stack requirements are minimized. We identify three main problems.

- *Problem 1:* hand-written code with no clear identification of runnables/functions, in such a way that only tasks are available. The operating system provides support for the definition of task-level preemption thresholds or non-preemption groups.
- *Problem 2:* hand-written or model-developed code with mapping of functions into tasks. The operating system only provides support for the definition of task-level preemption thresholds.
- *Problem 3:* hand-written or model-developed code with mapping of functions into tasks. The operating system provides support for preemption thresholds associated with runnables.

Our contributions

The approach to the problem that we investigate in this paper consists of the following:

- Explore the feasibility and the quality of a simulated annealing (SA) solution to the problems 1 and 3, comparing the results with the heuristics in [11] and the algorithm in [13]. This result is presented in Section 4.
- Provide rules and algorithms for determining the optimality of the threshold assignment to runnables and the ordering of runnables inside a task (a subset of the problem 3) where the priority assignment and the runnable-to-task mapping are given. This contribution is described in Section 5. With respect to this problem we also provide an improved analysis compared to the one in [13].
- Show how the memory requirements can be further reduced by changing the mapping of the runnables into tasks using a simulated annaling algorithm to synthesize the task set.

3 Systems Schedulability

We first quickly recall the formulas that are used to check feasibility in the case of our three problems.

3.1 Problem 1: Task model

The worst case response time of a task τ_i needs to be computed as the largest response time in a busy period of level p_i [12] [10]. Inside this busy period, several instances of τ_i may be activated, identified by an index q (with $q = 0 \dots q^*$). The worst-case response time of the task is the maximum among the response times of these instances.

$$r_i = \max_q \{r_i^{(q)}\} \leq d_i \quad q = 0 \dots q^*$$

The response time of each instance is obtained by first computing its worst-case start time $s_i^{(q)}$ and then its worst-case finish time $f_i^{(q)}$. In both cases of interest (preemption thresholds assigned to tasks or inherited from a non-preemptive group ceiling), $j \in hp(i)$ means the set of tasks with priority higher than the priority of τ_i and $j \in ht(i)$ means the set of tasks with priority higher than the (group ceiling) threshold of τ_i.

$$s_i^{(q)} = B_i + qc_i + \sum_{j \in hp(i)} (1 + \left\lfloor \frac{s_i^{(q)}}{t_j} \right\rfloor) c_j \leq d_i$$

$$f_i^{(q)} = s_i^{(q)} + c_i + \sum_{j \in ht(i)} (\left\lceil \frac{f_i^{(q)}}{t_j} \right\rceil - 1 - \left\lfloor \frac{s_i^{(q)}}{t_j} \right\rfloor) c_j$$

The response time of the q-th instance is $r_i^{(q)} = f_i^{(q)} - qt_i$. The length of the busy period (and the maximum index q^*) is computed with the formula in [10]. The blocking term B_i is the worst-case execution time of any task with priority lower than p_i that cannot be preempted by τ_i. This is because the task has a (group) threshold higher than p_i.

$$B_i = \max_k\{c_k\} \text{ with } p_k < p_i \leq y_k$$

Overall, the impact is a possible increase of the response time because of the B_i term, and a possible reduction at the end, because of the limited preemption once the task starts.

3.2 Problem 2: Runnables executed by tasks, threshold defined on tasks

When runnables are mapped into tasks, the only modification with respect to the previous formulas is to compute the worst-case execution time of a task as the sum of the worst-case execution times of the runnables mapped into it.

$$c_i = c_{i,0} + \sum_j \gamma_j \text{ where } \rho_j \in \mathcal{M}_i \quad (1)$$

3.3 Problem 3: Runnables executed by tasks, threshold defined on runnables

In this last case, preemption thresholds and non-preemption groups apply to runnables rather than tasks. The worst-case computation time of tasks are computed using Equation (1). The response time can be computed iteratively, for all the runnables mapped into the task.

Even when preemption thresholds are associated with runnables and a task dynamically inherits the runnable threshold, the task (and the runnables in it) can only be blocked once, before they start executing (after they start, they can only be preempted by a high priority task/runnable). The blocking time B_i is therefore computed on the first runnable of the task as the maximum execution time among those of lower priority runnables with a higher (group) preemption threshold, and mapped into a different task. This blocking time is inherited by all the other runnables mapped into the same task (for which B_i is computed). The start time of the k-th runnable of task τ_i is

$$s_{i,k}^{(q)} = B_i + qc_i + c_{i,k-1} + \sum_{j \in hp(i)} (1 + \left\lfloor \frac{s_{i,k}^{(q)}}{t_j} \right\rfloor) c_j \quad (2)$$

where $c_{i,k-1}$ is the sum of the worst-case execution times of all runnables mapped into τ_i from position 1 to $k-1$ plus $c_{i,0}$. The finish time is

$$f_{i,k}^{(q)} = s_{i,k}^{(q)} + \gamma_{i,k} + \sum_{j \in ht(i,k)} (\left\lceil \frac{f_{i,k}^{(q)}}{t_j} \right\rceil - 1 - \left\lfloor \frac{s_{i,k}^{(q)}}{t_j} \right\rfloor) c_j \quad (3)$$

where $ht(i,k)$ is the set of tasks that can preempt the k-th runnable of task τ_i.

4 Priority Assignment for the Task Model

In case task priorities are given, the algorithm proposed in [11] defines the maximum preemption threshold assignment for all tasks and (as demonstrated in [6]) the assignment with minimum preemption among tasks and stack usage. The remaining problem is to find an algorithm to assign task priorities such that all tasks are schedulable and the stack usage is minimized. The concept of task **blocking time limit** is proposed in [11] (and later defined in [13]) to assign priorities to tasks under preemption threshold scheduling. The blocking time limit of a task τ_i, denoted as h_i, is defined as the maximum blocking time a task can tolerate while still meeting its deadline. Starting from the lowest priority level, the algorithm assigns the current priority to the task with the largest blocking time limit among the remaining tasks, or the one with the smallest reduction in interference from higher priority tasks if the blocking time limit for all tasks are negative. The blocking time limit is calculated assuming that the preemption threshold is the same as the priority.

Algorithm 1 Heuristic for Priority Assignment to Tasks

```
Unassigned = 𝒯, Assigned = ∅.
for p = 1 to |𝒯| do
  for each task τ_i in Unassigned do
    assume p_i = p and p_j > p ∀τ_j ≠ τ_i in Unassigned
    if r_i ≤ d_i then
      a_i = h_i
    else
      a_i = d_i − r_i
    end if
  end for
  select τ_i from Unassigned with the largest a_i
  p_i = p
  Unassigned - = {τ_i}, Assigned + = {τ_i}
end for
```

The calculation of the task blocking time limit can be performed by trying possible values in a binary search until a given precision is achieved (as proposed in [11]). A more elegant way is to use the method in [13] based on the concept of request bound function. The *request bound function* of a task τ_i in $t \geq 0$, $rbf_i(t)$, is defined as the maximum cumulative execution time required by τ_i in any time interval of length t. For a periodic task, the request bound function is $rbf_i(t) = \lceil \frac{t}{t_i} \rceil c_i$. A preemptive task τ_i (i.e. its preemption threshold is the same as its priority) with deadline $d_i \leq t_i$ is feasible ([9]) if and only if there exists at least one time instant $t \leq d_i$ when the available CPU time in $[0,t]$ is larger than or equal to the time required for execution by τ_i and higher or equal priority tasks (denoted as $he(i) = hp(i) \bigcup \{i\}$), i.e. $\sum_{j \in he(i)} rbf_j(t) \leq t$. Thus, the blocking time limit h_i is the maximum slack between the available CPU time t and the total request from $he(i)$

$$h_i = \max_{t \in \mathcal{I}_i} (t - \sum_{j \in he(i)} \left\lceil \frac{t}{t_j} \right\rceil c_j) \quad (4)$$

The time instants t to be checked belong to a finite set $\mathcal{I}_i$, a subset of the integer multiples of the task periods (for

the definition of $\mathcal{I}_i$ please refer to [14]).

Equation (4) provides a conservative estimate of the blocking time limit, as it assumes the task is preemptive. However, the maximum preemption threshold of a lower priority task depends on the priority and preemption threshold of higher priority tasks. Since the algorithm in [11] assigns task priorities from the lowest level, the task blocking time limit calculation has to be based on an estimation (instead of an exact assignment) of its preemption threshold.

Other heuristic algorithm includes a deadline monotonic priority assignment (sub-optimal even for schedulability alone). In the experiments, we compare them with simulated annealing. Although these algorithms are developed for schedulability alone, they perform very well.

4.1 A Simulated Annealing Solution and Experimental Results

A simulated annealing algorithm is a general type of solution for the synthesis of system configurations when the problem is of exponential complexity. At each iteration, the algorithm computes a new system configuration using a random *mutation or transition operator*. The new solution is evaluated using a *metric function*. If it is better than the previous one, it is retained and becomes the new current solution. If the solution is worse than the previous one, it can be still conditionally accepted with a probability that decreases (exponentially) with the cost difference and a temperature parameter (the lower the temperature, the more difficult to accept higher cost transitions). The temperature must be high enough at the beginning of the algorithm to allow transitions to almost any possible solution (whether higher or lower cost). Then, it is iteratively lowered, until in the final iteration the algorithm performs in essence a local search.

The algorithm requires the definition of a transition function for computing new solutions and an evaluation function to estimate the cost/performance of the solutions. The transition functions for the task case is the following. A pair of tasks with priority p_i and p_{i+1} is randomly selected and their priorities are swapped. After the swap, the maximum preemption thresholds are assigned according to the (optimal) algorithm in [11]. Any swap of p_i and p_{i+1} will not change the preemption threshold of tasks with priority higher than p_{i+1}.

For the assignment of priorities to tasks, we implemented the heuristic based on the blocking time limit proposed in [11], a deadline monotonic assignment, and a simulated annealing algorithm. For all algorithms, once the priorities are assigned, we use the maximum preemption threshold assignment [11] to define the task thresholds.

We apply these algorithms to 2500 randomly generated cases having a number of tasks between 5 and 17. On average, the stack usage using the deadline monotonic approach is only 0.3% larger than the result from the simulated annealing algorithm (although in the worst case it can be 87% more). The stack usage of the blocking time-based algorithm is 1.1% more than the simulated annealing (worst case 93% more). Of course, both the deadline monotonic and the blocking time heuristics are much faster than the simulated annealing. For instance, when the number of tasks is 15, both heuristics take less than one second, while the simulated annealing algorithm takes 12 minutes on average. We further test the two heuristics with 5200 random cases that consist of up to 30 tasks. Both algorithms still finish within one second, and the average difference of the stack usage is within 0.8%. We also tried two other heuristics based on the stack usage of each task and the combination of stack usage and task deadlines. The results do not show improvements over the deadline monotonic approach.

Finally, for 1000 test cases with 5 to 9 tasks, we compare the solution obtained with the simulated annealing algorithm with the optimal solution (in terms of stack usage, obtained by enumerating all the possible priority assignments). In all cases, the simulated annealing algorithm finds the optimal solution. Although this does not provide guarantees for the cases with more than 9 tasks, it shows the (expected) good performance of the simulated annealing algorithm – and consequently of the two heuristics – when the number of tasks is relatively small.

Overall, the results show that for Problem 1 (task model), the two heuristics that are designed for schedulability also perform very well in terms of stack usage. Intuitively, when the utilization is small, and feasibility is not a problem, the threshold assignment algorithm finds a configuration with very little preemptability, so that the result is very often optimal. When the utilization grows, an initial configuration that eases schedulability also allows for higher thresholds, less preemptability, and hence less stack usage.

5 Stack Optimality for Functional Models

In this case, the operating system provides support for the definition of preemption thresholds or non-preemption groups associated with runnables. We first consider the case in which the function to task mapping and the task priority assignment are given, as in [13], where the task blocking time limit is used to find the maximum preemption threshold that can be assigned to a runnable.

The maximum preemption threshold $\eta_i^{\max}$ of ρ_i mapped into τ_r, is the highest priority level p_j such that all the tasks with priority between p_r and p_j have a blocking time limit no smaller than γ_i (its worst case execution time).

$$\eta_i^{\max} = \max\{p_j : \forall p_k \in (p_r, p_j], \gamma_i \leq h_k\} \qquad (5)$$

where $p^{\max}$ is the highest priority level in the system.

In [13], the blocking time limit is computed using Equation (4), which operates at the task level with the conservative assumption that the task is fully preemptive. In this case, the order of the runnables inside a task is irrelevant.

Additional information on runnables and their preemption threshold can be used to improve the computation of the blocking time limit. We denote the blocking time limit of runnable ρ_i as β_i. β_i can be computed by binary search (using (2) and (3)). An upper bound β_i^{ub} of the blocking time limit of a runnable is the difference between its deadline and its response time assuming its blocking time is zero.

The blocking time limit can also be calculated based on the formulation of feasibility regions in [14]. $\rho_{i,k}$, the k-th runnable in task τ_i is schedulable if, for each instance $q = 0 \cdots q^*$ in the busy period, there exists a pair of points $s, f \in [qt_{i,k}, qt_{i,k} + \delta_{i,k}]$ such that

$$\begin{cases} s \geq B_i + qc_i + c_{i,k-1} + \sum\limits_{j \in hp(i)} (1 + \left\lfloor \frac{s}{t_j} \right\rfloor) c_j \\ f \geq B_i + qc_i + c_{i,k-1} + \gamma_{i,k} + \sum\limits_{j \in ht(i,k)} \left\lceil \frac{f}{t_j} \right\rceil c_j \\ \qquad + \sum\limits_{j \in hnt(i,k)} (1 + \left\lfloor \frac{s}{t_j} \right\rfloor) c_j \end{cases} \quad (6)$$

where $hnt(i,k) = \{j : p_i < p_j \leq \eta_{i,k}\} = hp(i) \backslash ht(i,k)$ is the complement of $ht(i,k)$ with respect to $hp(i)$.

The set of candidate point pairs, start and finish times for the q-th instance of $\rho_{i,k}$ in the busy period can be found as $\mathcal{I}_i^{(q)}$, $\mathcal{S}_i^{(q)}$ and $\mathcal{F}_i^{(q)}$,

$$\begin{aligned} \mathcal{I}_{i,k}^{(q)} &= \{(s,f) : s \in \mathcal{S}_{i,k}^{(q)}, f \in \mathcal{F}_{i,k}^{(q)}, s \leq f + \gamma_{i,k}\} \\ \mathcal{S}_{i,k}^{(q)} &= \{mt_j : j \in hp(i), mt_j \in [q\theta_{i,k}, q\theta_{i,k} + \delta_{i,k}]\} \\ &\quad \bigcup \{qt_{i,k} + \delta_{i,k}\} (m \in \mathbb{N}^+) \\ \mathcal{F}_{i,k}^{(q)} &= \{mt_j : j \in ht(i,k), mt_j \in [q\theta_{i,k}, q\theta_{i,k} + \delta_{i,k}]\} \\ &\quad \bigcup \{qt_{i,k} + \delta_{i,k}\} (m \in \mathbb{N}^+). \end{aligned}$$

We define the interferences from the runnables with higher or equal priority on the right-hand sides of the inequalities in (7) by

$$\begin{aligned} \Sigma_i^{(q)}(s) &= qc_i + c_{i,k-1} + \sum_{j \in hp(i)} \left\lceil \frac{s}{t_j} \right\rceil c_j \\ \Phi_i^{(q)}(s,f) &= qc_i + c_{i,k-1} + \gamma_{i,k} + \sum_{j \in ht(i,k)} \left\lceil \frac{f}{t_j} \right\rceil c_j + \sum_{j \in hnt(i,k)} \left\lceil \frac{s}{t_j} \right\rceil c_j \end{aligned}$$

The schedulability condition of $\rho_{i,k}$ can be rewritten as

$$\begin{aligned} &\forall q = 0 \cdots q^*, \exists (s,f) \in \mathcal{I}_{i,k}^{(q)} \text{ such that} \\ &s > B_i + \Sigma_i^{(q)}(s) \quad \text{and} \quad f \geq B_i + \Phi_i^{(q)}(s,f) \end{aligned} \quad (7)$$

The blocking time limit $\beta_{i,k}$ for $\rho_{i,k}$ can be computed as

$$\beta_{i,k} = \min_{q=0 \cdots q^{*ub}} \max_{s,f \in \mathcal{I}_{i,k}^{(q)}} \min(s - \Sigma_i^{(q)}(s) - \epsilon, f - \Phi_i^{(q)}(s,f)) \quad (8)$$

where ϵ is a small number. The number of instances q^{*ub} in the busy period in (8) is conservatively computed using the upper bound $\beta_{i,k}^{ub}$. The blocking time limit of task τ_i is the minimum among those of the runnables mapped into it

$$h_i = \min_k \beta_{i,k} \quad (9)$$

A set of properties applies to the blocking time limit and maximum preemption threshold in relation to task mapping, priority assignment, and runnable execution order.

- **Property 1 (Monotonicity of $\beta_{i,k}$ with respect to the execution order of a runnable in a task):** as both $\Sigma_i^{(q)}(s)$ and $\Phi_i^{(q)}(s,f)$ are monotonically increasing with $c_{i,k-1}$, $\beta_{i,k}$ is monotonically decreasing with $c_{i,k-1}$. Its blocking time limit decreases with an increasing index k, and vice versa.
- **Property 2 (Monotonicity of $\beta_{i,k}$ with respect to the preemption threshold $\eta_{i,k}$):** $\Phi_i^{(q)}(s,f)$ is monotonically increasing with $ht(i,k)$, the set of tasks that can preempt $\rho_{i,k}$. The higher $\eta_{i,k}$, the smallest is $ht(i,k)$ and $\Phi_i^{(q)}(s,f)$. $\beta_{i,k}$ is non-increasing for increasing $\Phi_i^{(q)}(s,f)$, hence the proof.
- **Property 3 (Independency of the maximum $\eta_{i,k}$ from runnables with the same or lower priority)**: by Equation (5), the maximum preemption threshold for a runnable is independent from the execution order and the preemption threshold of runnables mapped to the same task or with lower priority. In particular, the maximum preemption threshold of a runnable is independent from its execution order within the task.

We now prove the existence of a feasible preemption threshold assignment for all runnables that is larger than (dominates) any other feasible assignment. The lemma is an extension of the theorem presented in [4] (where it is applied to the task model), but demonstrated in a different way, using the concept of blocking time limit.

Lemma 1. *Given the runnable to task mapping, the runnable execution order, and the task priority assignment, there exists a valid preemption threshold assignment $\eta^{\max}$ that is component-wise greater than any other valid preemption threshold assignment η: $\forall \rho_i, \eta_i \leq \eta_i^{\max}$.*

Proof. By induction. The theorem is trivially demonstrated for a system with only two tasks.

Suppose that it is possible to find such a maximum preemption assignment for a system with n tasks. Consider a system with $(n+1)$ tasks, where τ_{n+1} is the lowest priority task. Because of Property 3, the preemption thresholds of the runnables belonging to the n higher priority tasks are independent from the thresholds assigned to the runnables of τ_{n+1}. According to our induction hypothesis, such maximum threshold assignment for the n highest priority tasks exists and it is also, by Property 2, the one that maximizes the blocking time limit for the runnables mapped into the n highest priority tasks. According to Equation (5), the thresholds that can be assigned to the runnables in τ_{n+1} are therefore also maximized, thereby achieving the maximum preemption threshold assignment for the task set with $(n+1)$ tasks. □

$\eta^{\max}$ can be computed starting from the highest priority task down to the lowest priority one, as proposed in [13]. The existence of $\eta^{\max}$ also demonstrates the optimality of such preemption threshold assignment, as it minimizes the possible preemptions among runnables.

Theorem 2. *Among all the legal preemption threshold, $\eta^{\max}$ has the smallest total stack requirement.*

The theorem can be proved in exactly the way as it is done in [6] for the task model. We omit the proof here.

Next, we release one of the assumptions in [13] and include the selection of the runnables execution order inside the tasks among the design variables. We propose an algorithm for the assignment of the execution order based on the blocking time limit. The assignment starts from the highest priority task down to the lowest priority one. Within each task τ_i, the set **Assigned** (**Unassigned**) contains the set of runnables $\mathcal{F}_i$ mapped to τ_i that has (has not) been assigned with an execution order. Starting from the highest execution order $|\mathcal{F}_i|$ (the last runnable in the task), the algorithm assign the current execution order to the runnable with the largest blocking time limit among those in **Unassigned**. If the blocking time limit of one of the runnables is negative, then the task set is unschedulable. Otherwise, the algorithm returns a valid execution order.

Algorithm 2 Optimal Algorithm for Runnable Execution Order and Preemption Threshold Assignment

```
for each τ_i from highest priority to lowest priority do
    Unassigned = F_i, Assigned = ∅.
    for k = |F_i| to 1 do
        for each runnable ρ_j ∈ Unassigned do
            η_j = maximum preemption threshold as in (5)
            β_j = blocking time limit if m(ρ_j, τ_i, k) = 1
        end for
        select ρ_j from Unassigned with the largest β_j
        m(ρ_j, τ_i, k) = 1
        if β_j < 0 then
            return unschedulable
        end if
        Unassigned - = {ρ_j}, Assigned + = {ρ_j}
    end for
    h_j = blocking time limit of τ_j as in (9)
end for
```

Next, we demonstrate that the runnable order generated by Algorithm 2 is optimal with respect to system schedulability and stack space usage. We first provide a lemma.

Lemma 3. *Algorithm 2 maximizes the task blocking time limit h_i among all the possible runnable execution orders.*

Proof. We assume that in the execution order assignment O returned from Algorithm 2, the task blocking time $h_i = \beta_j$. We prove that any other execution order O' has a task blocking time $h'_i \leq h_i$. We denote the set of runnables with a smaller execution order than ρ_j in O as $\mathcal{R}_j$ (the set of runnables executed before ρ_j in O' is $\mathcal{R}'_j$).

Case 1: $\mathcal{R}_j \subseteq \mathcal{R}'_j$. In this case, by Property 1, $\beta'_j \leq \beta_j$, and $h'_i \leq \beta'_j \leq \beta_j = h_i$.

Case 2: $\mathcal{R}_j \nsubseteq \mathcal{R}'_j$. In this case, there exists at least one runnable in $\mathcal{R}_j$ that does not belong to $\mathcal{R}'_j$. Let ρ_k be the runnable in $\mathcal{R}_j$ with the **largest execution order** in O'. Thus, $\mathcal{R}_j - \{\rho_k\} \subseteq \mathcal{R}'_k$. In addition, ρ_j has a smaller execution order in O' than ρ_k, therefore $\rho_j \in \mathcal{R}'_k$, and $\mathcal{R}_j - \{\rho_k\} + \{\rho_j\} \subseteq \mathcal{R}'_k$. Let l be the execution order of ρ_j in O. The order of ρ_k in O' must be larger than l. Also, $\mathcal{R}'_k$ includes all runnables in $\mathcal{R}_j$ and all runnables with higher order than ρ_k in O' are also higher order than ρ_j in O. Hence, from Property 1, it is $\beta'_k \leq \beta_j$ (otherwise ρ_k should have higher order than ρ_j in O). Thus, $h'_i \leq \beta'_k \leq \beta_j = h_i$. □

Theorem 4. *Algorithm 2 is optimal for system schedulability. In addition, it returns the execution order with the smallest total stack requirement.*

Proof. The optimality of schedulability follows directly from Lemma 3: if there exists any solution that $h_i \geq 0$ (thus the system is schedulable), Algorithm 2 will find it.

From Lemma 3, Algorithm 2 maximize the task blocking time limits. The optimality of stack space requirement follows a similar reasoning to Lemma 1 and Theorem 2. □

5.1 Simulated Annealing solution and Experiments

For our third problem, the set of the design variable also includes the mapping of runnables to tasks and the task priority assignment. We develop a simulated annealing algorithm to solve this problem. Two transition operators are randomly selected (with equal probability): changing the allocation of a runnable, or changing the priority of a task. The task priority change is done by swapping the priority of two randomly selected tasks (and the runnables mapped inside them). When we change the allocation of a runnable, the operator randomly selects a runnable and then randomly selects one of the existing tasks or the creation of a new task as the new execution context for the runnable. If an existing task is selected, its period must be an integer divisor of the period of the runnable. In case a new task is created, it is assigned the lowest priority level and a period equal to the period of the runnable. After each transition, the execution order and the preemption threshold of the runnables are calculated using the optimal Algorithm 2.

We first evaluate the benefit of using a more accurate evaluation of the blocking time limit on the stack space that is estimated as required. We randomly generate systems consisting of $n = 10$ to 150 runnables. For each n, 1000 schedulable task sets are generated. The periods of the runnables are randomly drawn from the set $\{5, 10, 20, 40, 50, 100, 200, 400, 500, 1000\}$. For each period, one task is generated to implement all the runnables with the same period, and its deadline is assumed to be equal to its period. Priorities are assigned to tasks according to the rate-monotonic policy. The runnable execution time is generated such that its utilization is uniformly distributed between 0 and 100%. The runnable stack usage is uniformly distributed between 80 bytes and 512 byes. Among these 141000 random task sets, 19391 are incorrectly reported as unschedulable when using the pessimistic blocking time estimate from [13]. For 26972 of the remaining sets, the use of Algorithm 2 brings an additional stack space improvement with respect to [13]. On average, by exploring

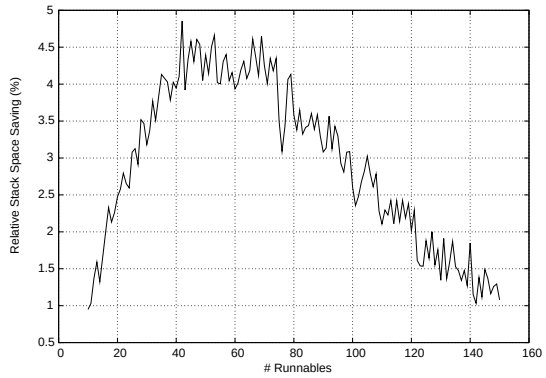

Figure 2. Reducing Stack Usage by Assigning an Optimal Runnable Execution Order.

the runnable execution order and an accurate schedulability analysis, Algorithm 2 can find solutions with 3.0% less stack space, compared to [13] on the total 141000 cases. Figure 2 shows the average relative stack space reduction from Algorithm 2 with respect to the number of runnables n in the set. However, if we consider only the sets on which there is an improvement, the average saving is 15.3%, with a maximum of 62.8%.

If the problem space also includes finding the optimal runnable-to-task mapping and task priority assignment, additional improvements are possible. We use a subset of the random task sets as input to the simulated annealing algorithm, consisting of 1000 applications with $n = 10$ to 20 runnables. The additional degrees of freedom in the design of the task set can be leveraged to further improve on the stack space requirements with respect to [13] for 846 of the 1000 sets. The maximum stack space improvement is 37.6%, and the average is 9.1%. This is significantly larger than the case of task model only. Considering the optimality of the algorithms that define the runnable execution order and the threshold assignment, and the limited gain in the stack space that can be obtained when using heuristics for the priority assignment in the task model (see Section 4.1), it seems that the runnable to task mapping may have a significant impact on the stack space requirements.

6 Conclusions

We discuss the design synthesis to minimize stack usage for systems with preemption threshold and non-preemption groups. We target the optimal assignment of these scheduling parameters in several cases of practical interest, including automotive modeling and coding standards. In particular, we evaluate the heuristics of priority assignment for systems with task information only. For systems that the list of runnables are available and the definition of preemption thresholds or non-preemption groups are supported at the runnable level, we provide rules for determining the optimality of the threshold assignment to runnables and the ordering of runnables inside a task where the priority assignment and the runnable-to-task mapping are given.

References

[1] *The AUTOSAR Standard, specification version 4.0*, the AUTOSAR consortium, web page: http://www.autosar.org.

[2] OSEK. OSEK/VDX Operating Systems specification, version 2.2.3. available at http://www.osek-vdx.org, 2006.

[3] M. Bohlin, K. Hanninen, J. Maki-Turja, J. Carlson and M. Nolin. "Bounding shared stack usage in systems with offsets and precedences." in *Proc. the Euromicro Conference on Real-Time Systems*, 2008.

[4] J. Chen, A. Harji, and P. Buhr, "Solution space for fixed-priority with preemption threshold." in *Proc. 11th IEEE Real-Time and Embedded Technology and Application Symposium*, 2005.

[5] P. Gai, G. Lipari, and M. Di Natale, "Minimizing memory utilization of real-time task sets in single and multiprocessor systems-on-a-chip," in *Proc. the 22nd IEEE Real-Time Systems Symposium*, 2001.

[6] R. Ghattas and A. G. Dean, "Preemption threshold scheduling: Stack optimality, enhancements and analysis." in *Proc. 13th IEEE Real-Time and Embedded Technology and Application Symposium*, 2007.

[7] K. Hanninen, J. Maki-Turja, M. Bohlin, J. Carlson, and M. Nolin;, "Determining Maximum Stack Usage in Preemptive Shared Stack Systems," in *Proc. the 27th IEEE Real-Time Systems Symposium*, 2006.

[8] W. Lamie. "Preemption-threshold." White Paper, Express Logic Inc. *Available at http://rtos.com/articles/18833*.

[9] J.P. Lehoczky, L. Sha, and Y. Ding. "The rate monotonic scheduling algorithm: Exact characterization and average case behavior." in *Proc. the 10th IEEE Real-Time Systems Symposium*, 1989.

[10] J. Regehr, "Scheduling tasks with mixed-preemption relations for robustness to timing faults," in *Proc. the 23rd IEEE Real-Time Systems Symposium*, 2002.

[11] M. Saksena and Y. Wang, "Scalable real-time system design using preemption thresholds," in *Proc. the 21st IEEE Real-Time Systems Symposium*, 2000.

[12] Y. Wang and M. Saksena, "Scheduling fixed-priority tasks with preemption threshold," in *Proc. the 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[13] G. Yao and G. Buttazzo. "Reducing Stack with Intra-Task Threshold Priorities in Real-Time Systems." in *Proc. the 10th International Conference on Embedded software*, 2010.

[14] H. Zeng and M. Di Natale. "An Efficient Formulation of the Real-time Feasibility Region for Design Optimization." *IEEE Transactions on Computers*, 2012.

[15] R. Bril, J. Lukkien, and W. Verhaegh, "Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption", *RTSJ*, 42:63-119, 2009.

[16] M. Bertogna, G. Buttazzo, G. Yao, "Improving feasibility of fixed priority tasks using non-preemptive regions", *RTSS*, pp. 251-260, 2011.

[17] M. Di Natale, A. Sangiovanni-Vincentelli, "Moving from federated to integrated architectures in automotive: The role of standards, methods and tools", Proceedings of the IEEE 98 (4), 603-620, 2010