Minimizing Stack and Communication Memory Usage in Real-time Embedded Applications

HAIBO ZENG, McGill University
MARCO DI NATALE, Scuola Superiore S. Anna
QI ZHU, University of California at Riverside

In the development of real-time embedded applications, especially those on systems-on-chip, an efficient use of RAM memory is as important as the efficient scheduling of the computation resources. The protection of communication and state variables accessed by concurrent tasks must provide for real-time schedulability guarantees while using the least amount of memory. Several schemes, including preemption thresholds, have been developed to improve schedulability and save stack space by selectively disabling preemption. However, the design synthesis problem is still open. We target the efficient assignment of the scheduling parameters to minimize memory usage for systems of practical interest, including designs that are compliant with automotive standards. We propose algorithms that are either proven to be optimal, or shown to improve on randomized optimization methods like simulated annealing.

Categories and Subject Descriptors: C.3 [**Computer Systems Organization**]: Special-Purpose and Application-based Systems – Real-Time and Embedded Systems

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Preemption threshold scheduling, stack requirement, data synchronization mechanism, memory usage

ACM Reference Format:
Zeng, H., Di Natale, M., and Zhu, Q. 2012. Minimizing Stack and Communication Memory Usage in Real-time Embedded Applications. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 39 (March 2012), 26 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Many real-time embedded systems, including automotive controls [Kopetz et al. 2007], are today developed as systems-on-chip. Because of the mass production and the cost constraints of such systems, they are typically characterized by tight time constraints, extremely high utilization, limited computation resources, and limited availability of memory. In most systems-on-chip and also, in general, in embedded systems, availability of RAM is a major constraint because of the hardware fabrication technology. Today, the space required to manufacture a RAM cell is 10 to 25 times that of a ROM cell, thus availability of both types of memory is typically inversely proportional with the same ratio.

This work is supported by the Natural Sciences and Engineering Research Council of Canada, under grant RGPIN 418741-12.
Author's addresses: H. Zeng, Department of Electrical and Computer Engineering, McGill University; M. Di Natale, ReTiS Lab, Scuola Superiore S. Anna; Q. Zhu, Department of Electrical Engineering, University of California at Riverside.

DOI 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

In this paper, we target at minimizing the use of RAM memory with the selection of task scheduling policies (impacting the use of stack space), and mechanisms to protect communication and state variables. We discuss the case of fixed-priority scheduling and several mechanisms for limiting preemption, including *preemption thresholds* and *non-preemptive groups*. These choices are driven by practical concerns. In many systems, and especially in the automotive market, where interchange of components and integration across the supply chain is a major requirement, compliance with standards is mandatory. In the automotive domain, the standard that applies to operating systems is AUTOSAR [The AUTOSAR consortium] that 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 allows the definition of non-preemptive groups and, to some degree, of a preemption threshold mechanism. In practice, an application-level implementation only requires an API call to change the task priority at runtime, which is available in most RTOSes.

In this work, we follow the AUTOSAR standard to describe the functional and communication architectures of the system. However, applicability of our results is not limited to AUTOSAR systems, but includes most systems according to a Model-Based design flow, such as those based on popular tools like Simulink [Mathworks] [Di Natale et al. 2010], and also several instances of manual code development.

1.1. AUTOSAR

The AUTOSAR development partnership has been created to develop an open industry standard for automotive software architectures and a common software infrastructure. The current version of the standard includes a reference architecture that supports a design model and process based on components, decoupling functional models from the supporting hardware and software services.

In AUTOSAR, the *functional architecture* is a collection of *Software Components* cooperating through their interfaces. The conceptual framework providing support for components communications is called *Virtual Functional Bus (VFB)*. Software Component interfaces are defined as a set of ports for data-oriented or service-oriented communication. These communications occur over the VFB. The definition of abstract components and the VFB nicely separates functionality from the physical architecture. The two are bound later in a process supported by tools for automatic code generation, where the actual VFB implementation depends on the placement of the components in the physical architecture. These tools take as input the HW platform description and placement constraints defined by the user and produce the task implementation, the placement of tasks on the ECUs, and the communication and synchronization layers. The generated codes include the *Basic Software* (the operating system and the device drivers) and the *Run-Time Environment* (RTE) which realizes communication (both inter- and intra-task as well as intra- and inter-ECU, Electronic Control Unit) and event generation, forwarding, and dispatching.

As shown on the left hand side of Figure 2, the *behavior* of each AUTOSAR component is represented by a set of *runnables*, software functions that can be executed in response to events generated by the RTE, such as timer activations (for periodic runnables), and data writes on ports or other application signals. Runnables are also called *function blocks* such as in Simulink [Mathworks]. Runnables may need to use and update state variables for their computations. This often requires exclusive access to such state variables. In addition, (data) interactions among components occur when runnables write into and read from interface ports. The reading and writing code is automatically generated by the AUTOSAR tools.

A mapping relation is defined between runnables (atomic schedulable units) and tasks, meaning that the runnable code is executed in the context of the task. Runnables from different components may be mapped into the same task, but their ordering relations (for example, resulting from a sequence of calls) must be preserved. Although one of the main objectives of AUTOSAR is to cope with complex distributed architectures and the placement of SW components on the ECUs of a distributed system, in this paper, we only deal with timing issues at the local level, that is, for components mapped into **tasks executing on the same ECU**. In the end, the mapping of runnables into tasks is defined as in the right-hand side of Figure 2. When communicating runnables are mapped into different tasks that can possibly preempt each other, the shared resources (denoted as ε_i in the figure) implementing the communication ports and state variables need to be suitably protected to ensure data consistency. For the example in Figure 2, ε_1 is a state variable communicated between two runnables (runnables 11 and 14) that are mapped to the same task (thus no preemption is possible), while the other resources may require additional mechanism to guarantee an atomic access to them (see Section 6). The figure shows one example in which the system model is a graph (possibly with cycles) of runnables communicating asynchronously over state variables and intercomponent ports (the graph edges). The execution of runnables inside each tasks is necessarily sequential. Models that include order of execution constraints (and flow preservation constraints) can be handled by adding runnables-to-task mapping constraints ([Di Natale and Zeng 2012]).

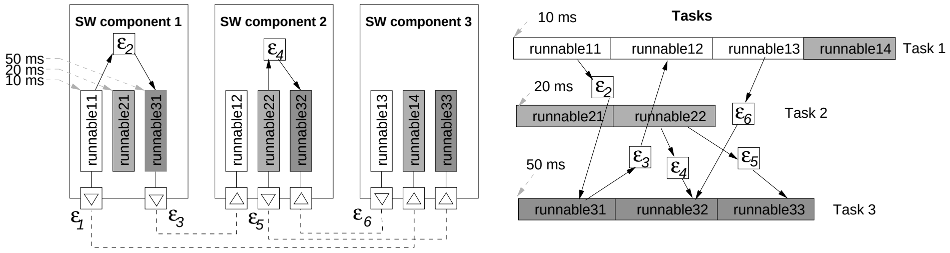


Fig. 1. Mapping of runnables into tasks in AUTOSAR.

The mapping of runnables into tasks, the configuration of the task model, and the selection of the mechanisms for the communication over ports (protecting against data inconsistency and possibly flow semantics violation [Ferrari et al. 2009]) all have a large impact on the performance of the system. In this paper, *we target at* **the design synthesis problem** *to select the assignment of these design variables such that the system is schedulable and the memory usage is minimized*.

The paper is organized as follows. In Section 2, we describe the system model, the related work, and our contributions. In Section 3, we summarize the conditions for system schedulability. The algorithm for minimizing the stack usage for our task model is presented in Section 4. In Section 5, we propose algorithms for the optimal assignment of parameters for the functional model. Section 6 extends the results to the consideration of memory usage for shared resources and, finally, Section 7 concludes the paper.

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 i-th runnable is denoted as ρ_i. In this work, we restrict to runnables that are activated in response to periodic timer events. Therefore, each runnable ρ_i is associated with a period θ_i and characterized by a worst-case

Table I. Notations for runnables and tasks

Name	Notation	WCET	Period	Deadline	Stack	Priority	Threshold
Runnables	ρ_i	γ_i	θ_i	δ_i	σ_i	-	η_i
Tasks	τ_j	c_j	t_j	d_j	s_j	p_j	y_j

execution time (WCET) γ_i, a worst-case stack requirement σ_i (in bytes), and a deadline δ_i. Also, each runnable may be associated with a preemption threshold η_i (its meaning is explained later). Runnables may communicate asynchronously by means of port variables (shared buffers) in a directed (possibly cyclic) communication graph (in agreement with AUTOSAR). In this section, we do not consider the optimization of the communication mechanisms, but only the impact of preemption thresholds on the runnables scheduling. The communication model and the optimization of resource sharing mechanisms are formally defined and evaluated in Section 6.

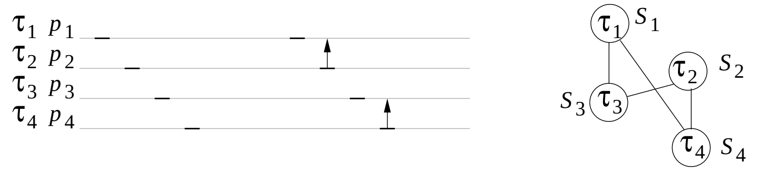
The implementation of runnables into tasks generates the *task* model $\mathcal{T} = \{\tau_1, \ldots, \tau_l\}$. Each task τ_j has a priority p_j (the higher the number, the higher the priority) and a period t_j. τ_j is also characterized by a WCET c_j, a deadline d_j equal to its period, stack space usage s_j, and a preemption threshold y_j.

A mapping relation may be defined between a runnable ρ_i and a task τ_j. The 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 execute before it in τ_j). We denote this mapping relation as $m(\rho_i, \tau_j, k) = 1$. The runnable with execution order k in τ_j is also labeled as $\rho_{j,k}$. A mapping relation is only possible if the execution period of ρ_i and τ_j are such that $\theta_i = z \cdot t_j$ for some integer z. For example, in Figure 2, runnable 14 is mapped to a task with a period (10ms) that is half of its own (20ms), thus it is executed once every two activations of the task. The set of runnables mapped into τ_j is also denoted as $\mathcal{F}_j$. *Runnables do not have a priority level, but only a preemption threshold level*. The linear mapping of runnables into tasks does not mean that the communication dependencies among runnables is restricted to data pipelines (as for example in Figure). Order of execution constraints among runnables (and a synchronous communication model) are not included in the model but could be considered by using an additional set of constraints in our mapping problem and the proposed solutions. For an overview of the constraints that need to be enforced in a runnable to task mapping in order to preserve a partial order of execution among runnables please refer to [Di Natale and Zeng 2012].

Finally, there may be cases in which a runnable is executed with multiple rates and possibly mapped into multiple tasks. Our model can be extended to handle this case (in a similar way as AUTOSAR does) by considering a runnable replica in the communication and scheduling problem. The runnable internal variables need to be considered as state variables and protected from concurrent access as in Section 6.

In *preemption threshold scheduling* [Wang and Saksena 1999], a task has two priority levels: a nominal priority, and a threshold priority that is assumed as soon as the task starts execution and retained until the end of its execution. 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.

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



(a) fully preemptive **(b)** preemption thresholds **(c)** preemption graph of (b)

Fig. 2. An example of preemption thresholds.

the runnables (setting up the calls to the runnable functions, forwarding data and events 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 $\gamma_{j,k}$, which is in reality very complex due to factors such as cache dependencies. Similarly, a stack equal to $s_{j,0}$ is needed to carry the communication data shared between runnables in the same task (which is typically much smaller than the stack space for each runnable). For the task stack size we have $s_j = \max_k \sigma_{j,k}$. Table I summarizes the list of design parameters/variables and their notations associated with tasks and runnables.

Figure 2 shows an example consisting of four tasks in priority order, where the endpoints of the arrow indicate the threshold priority at runtime, using fully preemptive scheduling (case (a)), or a set of preemption thresholds in (b). For each pair of tasks τ_i and τ_j, we denote $\tau_i \preccurlyeq \tau_j$ if τ_i can preempt τ_j, or $p_i > y_j$. This relation is transitive. A *preemption graph* is built where each task is represented by a vertex, with the weight equal to its stack usage. An edge is added from τ_i to τ_j if $\tau_i \preccurlyeq \tau_j$. The maximum system stack usage can then be computed as the highest weight path in the preemption graph [Bohlin et al. 2008] (the maximum system stack results from the chain of preemptions with maximum stack memory cost). The stack usage for the functional model is calculated in a similar way in [Yao and Buttazzo 2010]. The preemption graph for (b) is shown in Figure 2 (c), and the stack required for the execution of the tasks is

$$\text{(a) } \mathcal{S} = s_1 + s_2 + s_3 + s_4, \quad \text{(b) } \mathcal{S} = \max\{s_1 + s_3, s_1 + s_4, s_2 + s_3, s_2 + s_4\}$$

2.1. State of the Art

The two main mechanisms for limiting preemption in a controlled way are the *preemption thresholds* and the *non-preemptive groups*. The definition of *preemption thresholds* is first proposed in [Wang and Saksena 1999] to improve schedulability of real-time tasks. It spans from fully-preemptive to non-preemptive scheduling, subsuming these two extremes. A commercial implementation of the mechanism is provided in the ThreadX kernel [W. Lamie]. *Non-preemptive groups* make use of a similar but not equivalent concept (as shown in [Gai et al. 2001]). Each group is associated with a ceiling priority equal to the highest priority among all tasks belonging to the same group. Tasks belonging to a group execute with a preemption threshold equal to the group ceiling. This mechanism prevents interleaved executions (in an ABAB pattern, as opposed to fully nested) of tasks belonging to different groups and is fully supported by the AUTOSAR/OSEK OS standard [OSEK 2006].

The worst-case response time of tasks with preemption thresholds can be computed using the formulas in [Regehr 2002]. In [Wang and Saksena 1999; Saksena and Wang 2000], several algorithms are proposed to assign priority and preemption thresholds to tasks to improve their schedulability. When a feasible priority assignment is defined for tasks, an algorithm allows to compute the maximum preemption threshold for each task. The algorithm was formally proven correct in [Chen et al. 2005]. With respect to the task priority assignment, two algorithms are proposed. One is a branch-and-bound algorithm, the other is a heuristic with a similar strategy as Audsley's algorithm [Audsley 1991]. Starting from the lowest priority level, it selects the task to be assigned with the current priority level based on an approximate estimate of the blocking time limit (defined in Section 3).

Also, in [Saksena and Wang 2000] the preemption threshold assignment is followed by a tasks to thread mapping algorithm. Two tasks mapped into the same thread cannot preempt each other and, the mechanism partitions tasks into non-preemptive groups (similar to the mapping of a functional model of jobs/runnables into tasks). The stack requirement is obtained as the sum of the maximum stack requirement for each thread/group. The proposed algorithm maps tasks into threads so that the system is schedulable and the number of threads is minimized. Since the objective is the minimization of the number of threads (not the stack usage), this algorithm does not directly compares to ours.

Preemption thresholds and preemption groups are considered in [Gai et al. 2001] in the context of multiprocessor systems, where an analogy is pointed out between the concept of preemption threshold and the ceiling priority of a critical section protected by the Priority Ceiling protocol. This allows an extension of the mechanism to dynamic priority schemes (such as EDF). The analysis of the stack space requirements is today enabled by tools such as AbsInt [AbsInt] that perform the evaluation of the worst case stack space requirement for each function based on the analysis of the object files.

Task clusters and task barriers are introduced in [Regehr 2002] for better robustness. In [Ghattas and Dean 2007] a unified framework for static and dynamic priority scheduling with preemption thresholds is presented. The authors demonstrate that the algorithm in [Wang and Saksena 1999] for the assignment of the highest possible preemption thresholds *after priorities are assigned to tasks* is also optimal with respect to stack usage. When scheduling offsets are known, they can be exploited to further improve the analysis on stack usage [Hanninen et al. 2006; Bohlin et al. 2008].

In [Yao and Buttazzo 2010] a functional model is considered in which runnables are already mapped into tasks, task priorities are given, and the objective is the assignment of preemption thresholds to runnables. The maximum amount of blocking that can be tolerated by each runnable is computed and the stack use is minimized by increasing the runnable preemption thresholds as much as possible, starting from those belonging to the highest priority task. This algorithm is very similar to the one in [Saksena and Wang 2000]. However, feasibility is computed assuming tasks are preemptable, thus is unnecessarily pessimistic.

Other approaches have been proposed to limit preemption among tasks, including Deferred Preemption Scheduling [Baruah 2005; Yao et al. 2009] and Fixed Preemption Points[1] [Burns 1995; Bril et al. 2009; Bertogna et al. 2011]. These approaches, while reducing runtime overhead due to preemption and possibly improving system schedulability [Buttazzo et al. 2013], assume that preemption can be disabled for a predefined time interval or between selected locations inside the task code, therefore not guaranteeing savings in stack space.

The other design variables (and none of the above work considers them) include the mapping of runnables into tasks and the selection of the best mechanisms for the implementation of the communication over ports. The reduction of the context switch overhead is among the main drivers when defining the runnable mapping scheme presented in [Long et al. 2009]. An initial discussion of the possible data synchronization mechanisms to protect state variables is provided in [Ferrari et al. 2009]. These options offer tradeoffs as they have different time and memory overhead, and worst-case blocking time. [Wang et al. 2011] discusses the problem of sharing stacks for a differ-

[1] The terminology is sometimes used in an inconsistent way in different papers. We refer to the terms in [Buttazzo et al. 2013].

ent setting in a component-based OS, where each component uses a dedicated stack for error isolation, but the component may be concurrently invoked in multiple threads.

2.2. Our contributions

We identify two possible design scenarios:

- *Scenario 1:* hand-written code with no explicit identification of runnables. Only the task model is available. The operating system provides support for the definition of task-level preemption thresholds.
- *Scenario 2:* hand-written or model-developed code with mapping of runnables into tasks. The operating system provides support for preemption thresholds associated with runnables.

Our contribution is a rich set of algorithms to help in the solution of the design synthesis problem. More specifically,

- in Section 4, we propose an improved algorithm for the task priority assignment in scenario 1 for the minimization of stack memory.
- in Section 5 we provide rules and algorithms for the optimal runnable threshold assignment and runnable execution order inside a task where the priority assignment and the runnable-to-task mapping are given (scenario 2). Then we develop a heuristic to find the runnable mapping and task priority assignment.
- in Section 6, we extend the results to also consider the memory usage for data synchronization mechanisms based on shared variables. We provide an algorithm for the selection of data synchronization mechanism, and prove the applicability of the results from Section 5.

These algorithms are either proven to be optimal, or shown (by extensive experiments of more than 4000 hours CPU runtime) to be of comparable quality with respect to simulated annealing and provide improved results over existing work.

3. SYSTEM SCHEDULABILITY

We first recall the schedulability analysis results for systems using preemption thresholds or group ceilings.

3.1. Scenario 1: Task model

The worst-case response time of task τ_i is computed as the largest response time in a busy period of level p_i [Wang and Saksena 1999; Regehr 2002]. Inside this busy period, several instances of τ_i may be activated, identified by an index q (with $q = 0 \ldots q^*$). The length of the busy period (and the maximum index q^*) is computed with the formula in [Regehr 2002]. The worst-case response time of τ_i is the maximum among the response times of these instances.

$$r_i = \max_q \{r_i^{(q)}\} \leq d_i$$

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 $f_i^{(q)}$.

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

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

Here $j \in hp(i)$ means the set of tasks with priority higher than p_i and $j \in ht(i)$ means the set of tasks with priority higher than the threshold of τ_i. The response time of the q-th instance is $r_i^{(q)} = f_i^{(q)} - q \cdot t_i$. The blocking term B_i is the WCET of any task τ_k that has priority lower than p_i, but cannot be preempted by τ_i as $p_i \leq y_k$.

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

Overall, the impact is an increase of the response time because of possible blocking time, and a possible reduction because of the limited preemption once the task starts.

3.2. Scenario 2: Runnables executed by tasks, threshold defined on runnables

In this case, preemption thresholds apply to runnables rather than tasks. The worst-case computation time of tasks are computed using Equation (1).

$$c_i = c_{i,0} + \sum_{j:\rho_j \in \mathcal{F}_i} \gamma_j \tag{1}$$

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 it starts executing. The blocking time B_i is therefore computed on the first runnable of the task as the maximum execution time among those 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. The start time of the k-th runnable of task τ_i is

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

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

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

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

4. STACK OPTIMIZATION FOR THE TASK MODEL

In case task priorities are given, the algorithm proposed in [Wang and Saksena 1999] defines the maximum preemption threshold assignment for all tasks. As demonstrated in [Ghattas and Dean 2007], *this assignment minimizes preemption among tasks and has minimum system 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 [Saksena and Wang 2000] (and later also in [Yao and Buttazzo 2010]) to assign priorities to tasks. The blocking time limit of task τ_i, denoted as h_i, is defined as the maximum blocking time that τ_i can tolerate while still meeting its deadline.

Saksena et al. [Saksena and Wang 2000] propose an algorithm similar to Audsley's [Audsley 1991] for preemptive systems. Starting from the lowest priority level, the current priority is assigned 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 limits are negative for all tasks. However, the maximum preemption threshold of a lower priority task depends on the priority order and preemption threshold of higher priority tasks. Since the algorithm in [Saksena and

Wang 2000] (later in the paper referred to as **PA-Preemptive**, **P**riority **A**ssignment algorithm assuming **Preemptive** tasks) assigns task priorities from the lowest level, the task blocking time limit calculation has to be based on an estimate (instead of an exact assignment) of its preemption threshold. PA-Preemptive uses a conservative estimate of the blocking time limit by assuming the task can be preempted by all higher priority tasks. Thus, the blocking time limit is determined by the set of higher priority tasks, regardless of their relative priority order.

We developed an improved heuristic for estimating the blocking time limit, referred to as **PA-DMMPT**, **P**riority **A**ssignment algorithm assuming **D**eadline **M**onotonic and **M**aximum **P**reemption **T**hreshold (for the remaining tasks in the unassigned set) and summarized in Algorithm 1. Given that the computation of the task blocking time limit requires the exact priority order and preemption thresholds of higher priority tasks, we use deadline monotonic to estimate the priority order of higher priority tasks (line 5), and the maximum preemption threshold of these tasks are then found (for the estimated priority assignment) with the algorithm in [Wang and Saksena 1999] (line 6). Based on this improved estimate of the blocking time limit, starting from the lowest priority level, the task with the maximum blocking time limit (or the smallest lateness) among the ones in the unassigned set is selected at each step.

ALGORITHM 1: PA-DMMPT for Task Priority Assignment

```
   Input: Task set $\mathcal{T}$.
   Output: Priority assignment for tasks in $\mathcal{T}$.
1  Unassigned $= \mathcal{T}$;
2  for each priority level $p = 1$ to $|\mathcal{T}|$ do
3      for each task $\tau_i$ in Unassigned do
4          assume $p_i = p$;
5          assume deadline monotonic priorities for the set Unassigned$\setminus\{\tau_i\}$;
6          assign maximum preemption threshold to Unassigned;
7          calculate blocking time limit $h_i$ for $\tau_i$;
8          if $r_i \leq d_i$ then
9              $a_i = h_i$;
10         else
11             $a_i = d_i - r_i$;
12         end
13     end
14     select $\tau_i$ from Unassigned with the largest $a_i$;
15     $p_i = p$;
16     Unassigned = Unassigned $\setminus\{\tau_i\}$;
17 end
```

The complexity of finding the maximum preemption threshold assignment (line 6 of Algorithm 1) is $O(n^2 \cdot F(n))$ [Ghattas and Dean 2007], where n is the number of tasks in the system, and the function $F(n)$ is the complexity to check the schedulability (or compute the blocking time limit) of a task.[2] It is easy to see that Algorithm 1 makes $O(n^2)$ calls to line 6, thus the complexity of this algorithm is $O(n^4 \cdot F(n))$.

The task blocking time limit can be calculated by binary search until a given precision is achieved (as proposed in [Saksena and Wang 2000]). A more elegant way is to use the method based on the formulation of feasibility regions in [Zeng and Di Natale

[2] The complexity of schedulability analysis is only known to be pseudo-polynomial, see e.g. [Chen et al. 2005].

2013]. Task τ_i is feasible if for each instance $q = 0 \cdots q^*$ in the busy period, there exists a pair of points $s, f \in [q \cdot t_i, q \cdot t_i + d_i]$ such that

$$\begin{cases} s \geq B_i + q \cdot c_i + \sum_{j \in hp(i)} (1 + \left\lfloor \frac{s}{t_j} \right\rfloor) c_j \\ f \geq B_i + (q+1)c_i + \sum_{j \in ht(i)} \left\lceil \frac{f}{t_j} \right\rceil c_j + \sum_{j \in hnt(i)} (1 + \left\lfloor \frac{s}{t_j} \right\rfloor) c_j \end{cases} \tag{4}$$

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

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

$$\begin{aligned} \mathcal{I}_i^{(q)} &= \{(s, f) : s \in \mathcal{S}_i^{(q)}, f \in \mathcal{F}_i^{(q)}, f \geq s\} \\ \mathcal{S}_i^{(q)} &= \{mt_j : m \in \mathbb{N}^+, j \in hp(i), m \cdot t_j \in [q \cdot t_i, q \cdot t_i + d_i]\} \bigcup \{q \cdot t_i + d_i\} \\ \mathcal{F}_i^{(q)} &= \{mt_j : m \in \mathbb{N}^+, j \in ht(i), m \cdot t_j \in [q \cdot t_i, q \cdot t_i + d_i]\} \bigcup \{q \cdot t_i + d_i\}. \end{aligned}$$

We define the execution requests from the tasks with higher or equal priority on the right-hand sides of (4) by

$$\begin{cases} \Sigma_i^{(q)}(s) = q \cdot c_i + \sum_{j \in hp(i)} \left\lceil \frac{s}{t_j} \right\rceil c_j \\ \Phi_i^{(q)}(s, f) = (q+1)c_i + \sum_{j \in ht(i)} \left\lceil \frac{f}{t_j} \right\rceil c_j + \sum_{j \in hnt(i)} \left\lceil \frac{s}{t_j} \right\rceil c_j \end{cases}$$

The schedulability condition of τ_i can be rewritten as

$$\begin{aligned} &\forall q = 0 \cdots q^*, \exists (s, f) \in \mathcal{I}_i^{(q)} \text{ such that} \\ &s > B_i + \Sigma_i^{(q)}(s) \quad \text{and} \quad f \geq B_i + \Phi_i^{(q)}(s, f) \end{aligned} \tag{5}$$

Thus, the blocking time limit $h_i^{(q)}$ of the q-th instance of task τ_i in the busy period is

$$h_i^{(q)} = \max_{(s,f) \in \mathcal{I}_i^{(q)}} \{\min(s - \Sigma_i^{(q)}(s) - \epsilon, f - \Phi_i^{(q)}(s, f))\} \tag{6}$$

where ϵ is a small number. The blocking time limit h_i of τ_i itself is

$$h_i = \min_{q = 0 \cdots q^{*ub}} h_i^{(q)} \tag{7}$$

The number of instances q^{*ub} in the busy period in Equation (7) is conservatively computed using the upper bound h_i^{ub} (e.g. the laxity between the deadline and response time assuming no blocking time).

Once the blocking time limits of the higher priority tasks are calculated, the maximum preemption threshold of τ_i is

$$y_i = \max\{p_j : \forall p_k \in (p_i, p_j], c_i \leq h_k\} \tag{8}$$

In the experiments, we compare these heuristics with the optimal results obtained from exhaustive search for application sets with a small number of tasks and the results of simulated annealing for larger application sets. Although these algorithms are developed for schedulability alone, they perform very well for the problem of finding the minimal stack usage among the feasible solutions.

4.1. Experimental Results

For task priority assignment, we implemented the heuristic in [Saksena and Wang 2000] (*PA-Preemptive*), its improvement in Algorithm 1 (*PA-DMMPT*), the deadline monotonic assignment, and a simulated annealing solution. For all algorithms, once the priorities are assigned, the maximum preemption threshold assignment [Wang and Saksena 1999] is computed.

The simulated annealing algorithm requires the definition of a transition function for computing new solutions and an evaluation function to estimate the cost/performance of the solutions. A pair of adjacent 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. Any swap of two tasks with adjacent p_i and p_{i+1} will not change the maximum preemption threshold of tasks with priority higher than p_{i+1}. The initial solution is the Deadline Monotonic priority assignment.

We apply these algorithms to 13200 randomly generated cases having a number of tasks between 5 and 70. The task stack usage is uniformly distributed between 128 and 2048 bytes. After finding a solution for the task priority and preemption threshold assignment, the system stack usage is computed with as in the example in Figure 2. As the simulated annealing algorithm is quite slow, we are only able to run it for 3200 systems with no more than 20 tasks.

For all random systems, PA-DMMPT always returns a solution that is feasible, with no larger stack space than PA-Preemptive or the deadline monotonic policy. PA-Preemptive returns a priority assignment which is unfeasible in 28 cases. In 453 other cases, it returns a solution with a larger stack space than PA-DMMPT, where the maximum difference is 86.7%. The average difference amortized over all 13200 cases is only 0.8%. The deadline monotonic policy has similar results: it is unfeasible in 28 cases, and in 321 cases the corresponding solution has a larger stack space requirement. The average and maximum differences are 0.6% and 86.7% respectively.

In all the 3200 systems in which simulated annealing was able to compute a solution in reasonable time, *PA-DMMPT computed the same results as simulated annealing*. As expected, all the heuristic algorithms have a much shorter execution time. For instance, when the number of tasks is 20, each of the heuristics takes less than one second, while the simulated annealing algorithm takes 40 minutes on average. Finally, for 1000 test cases with 5 to 9 tasks, we use exhaustive search to find the optimal solution. In all cases, *both PA-DMMPT and simulated annealing return the optimal solution*. Although this does not provide guarantees for the cases with more than 9 tasks, it shows the (expected) good performance of simulated annealing and PA-DMMPT – and to some degree of the other two heuristics.

Overall, the results demonstrate that for the first scenario (the task model), the three heuristics that are designed for schedulability also perform very well in terms of stack usage. Intuitively, *an algorithm that eases schedulability also allows for higher thresholds, less preemptability, and hence less stack usage*. Also, the heuristic PA-DMMPT always generates the same or better results than the other two algorithms (deadline monotonic and PA-Preemptive). In the following, we leverage this result to develop algorithms for systems developed from a functional model (scenario 2).

5. STACK OPTIMIZATION FOR FUNCTIONAL MODELS

In this section, we consider the problem of stack space minimization for systems developed starting from a functional model and where preemption thresholds can be associated with runnables. As demonstrated in [Yao and Buttazzo 2010], assuming runnables as the atomic scheduable units (instead of tasks) can reduce the system stack usage, because provides a finer granularity for the definition of code sections with limited preemption, thus enabling more opportunities to share stack space; and has the advantage of only allowing preemption at runnable boundaries where the stack usage is relatively low (no functional code is executed between two runnables). However, the analysis in [Yao and Buttazzo 2010] has two important limitations. First, the system schedulability is checked assuming tasks are preemptable (and the runnable execution order inside a task is irrelevant), which introduces unnecessary pessimism. Second, it assumes the runnable to task mapping and task priority assignment are given.

We address the limitations of [Yao and Buttazzo 2010] and tackle the design problem in steps. First, we formally prove that, similar to the task model, a maximum preemption threshold assignment exists and is optimal in terms of minimizing system stack usage. Then, we leverage a more accurate schedulability analysis, and provide rules and optimal algorithms for the assignment of runnable thresholds and the runnable execution order inside a task, assuming the runnable to task mapping and the task priority assignment are given (as in [Yao and Buttazzo 2010]). Finally, we develop an efficient heuristic to find the runnable mapping and task priority assignment, with results comparable to (or slightly better than) simulated annealing.

5.1. Optimality of Maximum Preemption Threshold

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

$$\eta_i^{\max} = \max\{p_j : \forall p_k \in (p_r, p_j], \gamma_i \le h_k\} \tag{9}$$

In [Yao and Buttazzo 2010], the blocking time limit computation is performed at the task level with the conservative assumption that the task is fully preemptive. In this case, the order of execution of the runnables inside a task is irrelevant. Instead, 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 the k-th runnable $\rho_{i,k}$ of task τ_i as $\beta_{i,k}$. $\beta_{i,k}$ can be computed in a similar way as in the case of task model.

$$\beta_{i,k} = \min_{q=0\cdots q^{*ub}} \beta_{i,k}^{(q)}, \text{ where } \beta_{i,k}^{(q)} = \max_{s,f\in\mathcal{I}_{i,k}^{(q)}} \{\min(s - \Sigma_{i,k}^{(q)}(s) - \epsilon, f - \Phi_{i,k}^{(q)}(s,f))\} \tag{10}$$

$$\text{and } \begin{cases} \Sigma_{i,k}^{(q)}(s) = qc_i + c_{i,k-1} + \sum_{j\in hp(i)} \left\lceil \frac{s}{t_j} \right\rceil c_j \\ \Phi_{i,k}^{(q)}(s,f) = qc_i + c_{i,k-1} + \gamma_{i,k} + \sum_{j\in ht(i,k)} \left\lceil \frac{f}{t_j} \right\rceil c_j + \sum_{j\in hnt(i,k)} \left\lceil \frac{s}{t_j} \right\rceil c_j \end{cases}$$

Here $ht(i,k) = \{j : p_j > \eta_{i,k}\}$, $hnt(i,k) = hp(i) \setminus ht(i,k)$. The blocking time limit of task τ_i is the minimum among those of the runnables mapped into it

$$h_i = \min_k \beta_{i,k} \tag{11}$$

A set of properties (with a proof sketch) apply to the blocking time limit and maximum preemption threshold in relation to runnable execution order.

— **Property 1 (Monotonicity of $\beta_{i,k}$ with respect to the execution order of runnable $\rho_{i,k}$ in the task):** Since $\Sigma_{i,k}^{(q)}(s)$ and $\Phi_{i,k}^{(q)}(s,f)$ are monotonically increasing with $c_{i,k-1}$, $\beta_{i,k}$ is monotonically decreasing with $c_{i,k-1}$. Thus the blocking time limit of a runnable decreases if we put more runnables ahead of it in the execution order, and vice versa.

- **Property 2 (Monotonicity of $\beta_{i,k}$ with respect to the preemption threshold $\eta_{i,k}$):** $\Phi^{(q)}_{i,k}(s,f)$ is monotonically increasing with $ht(i,k)$, the set of tasks that can preempt $\rho_{i,k}$. The higher $\eta_{i,k}$, the smaller $ht(i,k)$ and $\Phi^{(q)}_{i,k}(s,f)$ are. $\beta_{i,k}$ is non-increasing with respect to $\Phi^{(q)}_{i,k}(s,f)$, hence the proof.
- **Property 3 (Independency of the maximum $\eta_{i,k}$ from runnables with the same or lower priority)**: by Equation (9), the maximum preemption threshold of a runnable is independent from the execution order and preemption threshold of runnables mapped to the same task or with lower priority.

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 in [Chen et al. 2005] applied to the task model, but proved in a different (and simpler) way by using the concept of blocking time limit.

LEMMA 5.1. *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 true for systems with only one task. Suppose 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. By Property 2, it is also the one that maximizes the blocking time limit for the runnables mapped to the n highest priority tasks. By Equations (11) and (9), the thresholds of the runnables in τ_{n+1} are therefore also maximized, thus achieving maximum preemption threshold assignment for the task set with $(n+1)$ tasks. □

$\eta^{\max}$ should be computed starting from the highest priority task down to the lowest priority one, as proposed in [Saksena and Wang 2000; Yao and Buttazzo 2010]. The existence of $\eta^{\max}$ also demonstrates its optimality of stack usage, as it minimizes the possible preemptions among runnables.

THEOREM 5.2. *Among all the legal preemption threshold, $\eta^{\max}$ has the smallest total stack requirement.*

The theorem can be proved in exactly the way as in [Ghattas and Dean 2007] (for the task model). We omit the proof here.

5.2. Optimal Execution Order Assignment

Next, we include *the assignment of the runnables execution order inside the tasks among the design variables*, but runnable mapping and task priority assignment are still assumed to be given.

We propose an algorithm for the assignment of an execution order to runnables based on the blocking time limit, as in Algorithm 2. 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 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 the selected runnable 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 Threshold Assignment

Input: Task set with predefined priority and runnable mapping.
Output: Assignment of execution order and threshold to each runnable.

```
1  for each $\tau_i$ from highest priority to lowest priority do
2      $Unassigned = \mathcal{F}_i, Assigned = \emptyset$;
3      for $k = |\mathcal{F}_i|$ to $1$ do
4          for each runnable $\rho_j \in$ Unassigned do
5              $\eta_j =$ maximum preemption threshold as in Equation (9);
6              $\beta_j =$ blocking time limit of $\rho_j$ assuming $m(\rho_j, \tau_i, k) = 1$;
7          end
8          select $\rho_j$ from Unassigned with the largest $\beta_j$;
9          map $\rho_j$ as the $k$-th runnable of $\tau_i$ ($m(\rho_j, \tau_i, k) = 1$);
10         if $\beta_j < 0$ then
11             RETURN unschedulable;
12         end
13         $Unassigned = Unassigned \setminus \{\rho_j\}, Assigned = Assigned \bigcup \{\rho_j\}$;
14     end
15     $h_i =$ blocking time limit of $\tau_i$ as in Equation (11);
16 end
```

The complexity of Algorithm 2 is analyzed in a similar way as Algorithm 1. Assuming that the blocking time limit of a runnable (line 6 in the algorithm) is computed with complexity $F(m)$ where m is the number of runnables in the system. For each task τ_i containing m_i runnables, it requires $O(m_i^2)$ calls to line 6 to find a total order of runnable execution. Thus, the complexity of Algorithm 2 is $O(\sum_i m_i^2 \cdot F(m)) = O(m^2 \cdot F(m))$ (note that $m = \sum_i m_i$).

We now 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 5.3. *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 $h_i = \beta_j$ of runnable ρ_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 such runnables in O' is denoted as $\mathcal{R}'_j$).

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, $\mathcal{R}_j \setminus \{\rho_k\} \subseteq \mathcal{R}'_k$. In addition, ρ_j has a smaller execution order in O' than ρ_k, therefore $\rho_j \in \mathcal{R}'_k$, and

$$\mathcal{R}_j \setminus \{\rho_k\} \bigcup \{\rho_j\} \subseteq \mathcal{R}'_k \tag{12}$$

Let l denotes the execution order of ρ_j in O. Algorithm 2 (line 8) should guarantee that if we select ρ_k from $\mathcal{R}_j \bigcup \{\rho_j\}$ as the l-th runnable (thus the set of runnables with a smaller execution order than ρ_k is $\mathcal{R}_j \setminus \{\rho_k\} \bigcup \{\rho_j\}$), its blocking time limit should be no larger than β_j. Therefore, by Equation (12) and Property 1, it is $\beta'_k \leq \beta_j$, and $h'_i \leq \beta'_k \leq \beta_j = h_i$. □

Table II. An example of three runnables

			Assignment 1					Assignment 2				
ρ_i	θ_i	γ_i	τ_i	p_i	η_i	δ_i	β_i	τ_i	p_i	η_i	δ_i	β_i
ρ_1	4	10	τ_1	3	3	10	6	τ_1	3	3	10	6
ρ_2	10	25	τ_2	2	2	25	3	τ_2	2	2	25	3
ρ_3	5	50	τ_3	1	1	50	5	τ_2	2	3	25	2

THEOREM 5.4. *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 5.3: if there exists a solution such that $h_i \geq 0$ (the system is schedulable), then Algorithm 2 will find it. From Lemma 5.3, Algorithm 2 maximizes the task blocking time limits. The optimality of the stack space requirement follows from a reasoning similar to Lemma 5.1 and Theorem 5.2. □

5.3. Runnable to Task Mapping and Priority Assignment

Finally, we include *the mapping of runnables to tasks and the task priority assignment in the set of design variables*. As a starting point, we assume that each runnable is executed by a dedicated task, and Algorithm 1 is used to find a close to optimal priority assignment. However, mapping multiple runnables into the same task provides an additional opportunity for avoiding preemption and saving stack space.

Consider an example with three runnables with WCETs and periods $\rho_1 = (3, 10)$, $\rho_2 = (10, 25)$, and $\rho_3 = (5, 50)$, as in Table II. The only feasible solution with a one to one mapping of runnables to tasks is Assignment 1 on the left, in which ρ_3 must be scheduled with preemption as its execution time is larger than the blocking time limit of ρ_2. If we map ρ_2 and ρ_3 to the same task (Assignment 2, on the right), the blocking time limit of ρ_2 is no longer a limiting factor in deciding the maximum preemption threshold of ρ_3. This allows to further disable preemption between ρ_1 and ρ_3.

On the other hand, the task period (and its deadline) must be an integer divisor of the runnable period, therefore, a task implementing multiple runnables must have a period equal to the greatest common divisor of all its runnables. This can result in a tighter deadline for some runnables and possibly make the system unschedulable.

Based on the above observation, we propose an algorithm that works by iterative refinement. A one-to-one runnable to task assignment is the initial solution. Priorities are assigned to tasks using Algorithm 1. Then, the algorithm verifies if there are opportunities for further improvement by merging tasks and reordering the execution of runnables in the merged tasks using Algorithm 2 (optimal for the execution order and preemption threshold assignment).

First, we provide a result guaranteeing that task merging is always beneficial when two tasks have adjacent priority levels and equal period.

THEOREM 5.5. *Consider a task set $\mathcal{T}$ in which two tasks τ_i and τ_{i+1} have adjacent priority $p_i + 1 = p_{i+1}$ and equal periods $t_i = t_{i+1}$. If the runnables in τ_i are moved to τ_{i+1}, the maximum preemption threshold ${\eta'}^{\max}$ of the new task configuration is component wise no smaller than $\eta^{\max}$.*

PROOF. We only provide a proof sketch. By Property 3, the maximum preemption threshold of runnables with priority higher than p_{i+1} and those originally in τ_{i+1} remain the same, so are their blocking time limits.

For the runnables originally in τ_i, suppose the blocking time limit before the merge is $h_i = \beta_{i,k}$. By Equation (9), their maximum preemption threshold does not decrease in the new mapping. In addition, since $\forall s, \forall l$, $\Sigma_{i,k}^{(q)}(s) \geq \Phi_{i+1,l}^{(q)}(s, s) \geq \Sigma_{i,l}^{(q)}(s)$, it must be

$h_i \leq h_{i+1}$. Hence, the blocking time limit of τ'_{i+1} is no smaller than τ_i, and the maximum preemption threshold for runnables with priority lower than p_{i+1} also increases or remains the same. □

ALGORITHM 3: Algorithm for Runnable Mapping and Task Priority Assignment

```
Input: Set of runnables.
Output: Runnable to task mapping and task priority assignment.
1  Create one task for each runnable;
2  Find an initial priority assignment to tasks using Algorithm 1, modified by merging adjacent
   tasks with the same period (Theorem 5.5);
3  for each task $\tau_i$ from lowest priority do
4      $s^{\min}$ = current system stack usage, $j^{\min} = -1$;
5      for each task $\tau_j$ other than $\tau_i$ do
6          temporarily move all runnables in $\tau_i$ to $\tau_j$;
7          temporarily update $p_j$ to $gcd(p_i, p_j)$;
8          assign order and threshold to runnables in $\tau_j$ (Algorithm 2);
9          $s_j$ = system stack usage for new temporary configuration;
10         if $s^{\min} > s_j$ then
11             $s^{\min} = s_j$, $j^{\min} = j$;
12         end
13         undo temporary remapping
14     end
15     if $j^{\min} \neq -1$ then
16         map all runnables in $\tau_i$ to $\tau_{j^{\min}}$;
17         update $p_{j^{\min}}$ to $gcd(p_i, p_{j^{\min}})$;
18         assign order and threshold to runnables in $\tau_j$ (Algorithm 2);
19     end
20 end
```

We propose a greedy algorithm to find the runnable mapping and task priority assignment, shown in Algorithm 3. First, an initial mapping of runnables to tasks and a priority assignment is defined by using Algorithm 1 with a slight change on line 5: a single task implements all runnables with the same period (Theorem 5.5), and Algorithm 2 is used to define the runnable execution order and threshold. Next, further task merging opportunities are explored. Starting from the lowest priority task, the runnables belonging to each task τ_i are tentatively moved to a different task τ_j (with lower or higher priority) to see whether the new mapping can reduce the stack space requirement. When evaluating a merge, the period must be updated to the greatest common divisor of the original tasks and the priority is that of τ_j. Algorithm 2 is applied to find the optimal runnable execution order in the new mapping. This step is a greedy local search. After trying all such possible merge, the runnables in τ_i are remapped to the task (if any) that provides the maximum stack space reduction.

Algorithm 3 makes at most $O(m^2)$ calls to line 8 (Algorithm 2), for which the complexity is $O(m^2 \cdot F(m))$. Hence, the complexity of Algorithm 3 is $O(m^4 \cdot F(m))$.

5.4. Experimental Results

We first evaluate the benefit of *using a more accurate evaluation of the blocking time limit and the optimal runnable execution order* on the required stack space (as compared to [Yao and Buttazzo 2010]). The periods of the runnables are randomly drawn

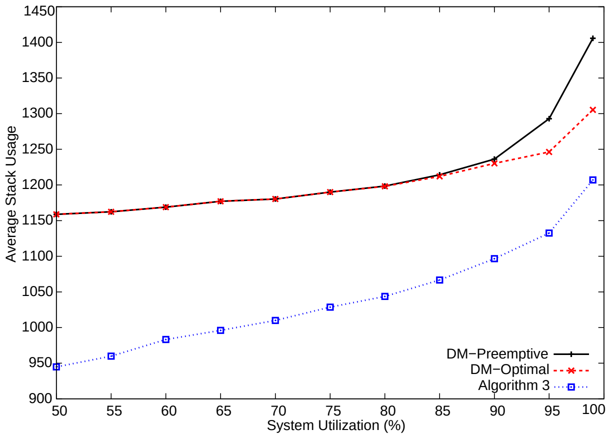


Fig. 3. Average system stack at different system utilization when the number of runnables = 50.

from the set $\{5, 10, 20, 40, 50, 100, 200, 400, 500, 1000\}$. The runnable stack usage is uniformly distributed between 80 and 512 bytes. The stack usage required by each task for its local variables, and active in between the execution of its runnables is 80 bytes.

The *first experiment* is to check the stack usage with respect to the system utilization. As a baseline solution for comparison, we assume a task model for [Yao and Buttazzo 2010] as follows (denoted as *DM-Preemptive*): for each period, one task implements 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 deadline monotonic policy (and the task blocking time limit is estimated assuming it is preemptive, as in [Yao and Buttazzo 2010]). We check the use of Algorithm 2 for the optimal runnable execution order assignment (denoted as *DM-Optimal*), and finally the use of Algorithm 3 for the additional design space of runnable-to-task mapping and task priority assignment. We set the number of runnables to $n = 50$, and vary the system utilization from $U = 50\%$ to 99%. For each U, 1000 schedulable task sets are generated.

The results are shown in Figure 3. The memory required by the solutions computed by *DM-Optimal* is almost the same as the memory used by *DM-Preemptive*, for system utilizations lower than 90%. For very high utilization values (95% and 99%), the schedulability analysis that considers the non-preemptability of the last runnable performs significantly better. 492 out of the 2000 cases are incorrectly reported as unschedulable when using the pessimistic blocking time estimate from [Yao and Buttazzo 2010] (*DM-Preemptive*). For 484 of the remaining sets, the use of Algorithm 2 brings an additional stack space improvement with respect to [Yao and Buttazzo 2010]. On average, the exploration of a better runnable execution order, together with the improved analysis (*DM-Optimal*) can find solutions with 5.1% less stack space, compared to *DM-Preemptive* on the 2000 high utilization cases. On the other hand, the improvement that can be obtained from the optimization of the task model in Algorithm 3 does not depend on the system utilization. The optimized task model requires less memory than the solution computed by using *DM-Preemptive*, with an average improvement of 11%-19% for each U. This confirms the benefits of exploring *runnable-to-task mappings and task priority assignments* in the problem space.

In *the second set of experiments*, we keep the system utilization constant (U =70% or 90%) while exploring a different number of runnables $n = 10, 15, \cdots, 100$ in the system (i.e. on average, the number of runnables for each possible period value is from 1 to 10). For each n, 1000 schedulable task sets are generated. We compute the stack

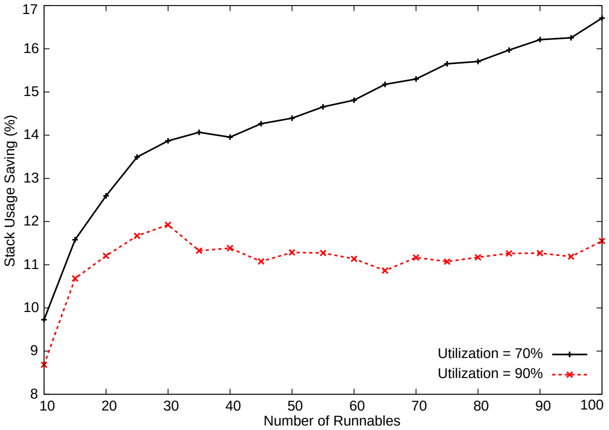


Fig. 4. Average stack savings of Algorithm 3 compared to DM-Preemptive vs. number of runnables.

space savings obtained using Algorithm 3 compared to *DM-Preemptive*, as shown in Figure 4. With $U = 90\%$, the savings are only slightly dependent on the number of runnables. On the other hand, for $U = 70\%$ the stack space savings increase with the number of runnables. The limited improvement at high utilization values can be intuitively explained by the limited number of task mapping configurations that are feasible when the utilization is quite high ($U = 90\%$ and higher, as also shown in Figure 3). Similarly, the greater benefit that our mapping and priority assignment algorithm can provide for a higher number of runnables is intuitively explained by the consideration that the larger is the number of runnables, the higher is the number of feasible task configurations.

To evaluate the quality of the proposed solution for the runnable mapping and task priority assignment, we perform *additional experiments*. The design space is too large to verify the optimality of the computed solutions by exhaustive search, even for a very small number of tasks. Therefore, as a comparison to the greedy Algorithm 3, we developed a simulated annealing solution. Two transition operators are randomly selected (with equal probability): changing the mapping 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 mapping of a runnable, the operator randomly selects a runnable, then it randomly chooses one of the existing tasks or creates 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 with the lowest priority and a period equal to the runnable. After each transition, the execution order and the preemption threshold of the runnables are calculated using the optimal Algorithm 2. The initial solution is set to be DM-Optimal, i.e., to use one task implementing all runnables with the same period and deadline monotonic priority assignment.

We use 11000 random task sets with $n = 10$ to 20 runnables as input to the simulated annealing algorithm. The runnable execution time is generated such that its utilization is uniformly distributed between 0 and 100%. Algorithm 3 generates solutions with comparable quality to simulated annealing: for 55 of the cases the result is better than simulated annealing (with a maximum difference of 50.5%), for 18 of the cases it is worse (maximum difference 35.4%), and for the remaining 10927 cases (by far the majority), the two computed solutions have the same stack usage. The cases in which Algorithm 3 and simulated annealing have different results are almost uniformly distributed with respect to the number of runnables or the processor utilization (besides being in a very limited number), showing no significant correlation with any of these parameters.

6. MEMORY OPTIMIZATION FOR THE FUNCTIONAL MODEL WITH COMMUNICATION VARIABLES

In this section, we extend the design synthesis problem by considering the RAM memory required for the implementation of communication using shared variables.

6.1. System Model

The functional model $\mathcal{F}$ is further enriched for the consideration of shared communication variables by a *Directed Graph* $\{\mathbb{V}, \mathbb{E}\}$, where $\mathbb{V}$ is the set of vertices, representing the runnables, and $\mathbb{E}$ the set of edges or communication links between them. Each link communicates a set of periodic data values, implementing a (discrete time) signal exchanged between runnables. This model can be applied to functional models derived from popular commercial modeling and simulation tools, such as Simulink, in which such a graph has inputs from sampling, source, and constant blocks, representing the signals from the controlled system or plant. At the other end of the graph, the output signals are the result of the controllers' computations.

A runnable ρ_i reads from a set of *input ports* and writes to a set of *output ports*. We denote the set of data ports accessed by ρ_i as $\mathcal{E}_i$. The runnable *period* θ_i is also the sampling period for the signals on the input ports. The signals are processed by the runnable and the result of the computation is a set of signal with the same rate, produced on the output ports.

$\mathcal{E} = \{\varepsilon_1, \ldots, \varepsilon_{|\mathcal{E}|}\}$ is the set of *shared resources* (in a one-to-one correspondence with data ports). We consider the case of one-to-many communication: a shared resource ε_i has a writer runnable, and a set of reader runnables. We denote the set of runnables accessing ε_i as $\rho(\varepsilon_i)$.

The execution time of runnable ρ_i is characterized by the tuple $(\gamma_{i,0}, \gamma_i(\varepsilon_k), \forall \varepsilon_k \in \mathcal{E}_i)$ where $\gamma_{i,0}$ is the total WCET of the normal execution segments, and $\gamma_i(\varepsilon_k)$ is the WCET of the critical section accessing the input/output ports ε_k. The total worst case execution time γ_i of ρ_i is

$$\gamma_i = \gamma_{i,0} + \sum_{\varepsilon_k \in \mathcal{E}_i} \gamma_i(\varepsilon_k)$$

As summarized in [Ferrari et al. 2009], there are three different mechanisms to guarantee data consistency, with different timing and memory overhead, as discussed below.

- **M1:** Disabling preemption among runnables by appropriately setting their preemption thresholds.
- **M2:** Semaphore locks with predictable blocking time, as in the Priority Ceiling Protocol (PCP) [Sha et al. 1990].
- **M3:** Wait free methods, as in Chen and Burns' algorithm [Chen and Burns 1997], or their flow preserving counterparts in [Sofronis et al. 2006] and [Wang et al. 2009].

M1: the implementation has minimum impact on the code, and we assume that *their timing and memory overhead are negligible*. As discussed previously, this protection method results in a worst case blocking time, unless the two runnables are mapped into the same task.

M2: if a shared resource is protected by an immediate priority ceiling semaphore, the resource is assigned a priority ceiling equal to the highest priority of any task which may lock the resource. When a task locks the shared resource, its priority is temporarily raised to the priority ceiling of the shared resource, thus no task that share the resource is able to execute. This allows a low priority task to defer the execution of any task with priority lower than or equal to the priority ceiling of the shared resource.

In this case, we assume *the timing overhead is negligible, and the memory overhead is zero*. However, the use of priority ceiling semaphores may introduce blocking to task τ_i in the measure of the longest critical section from a lower priority task on a resource with priority ceiling $\geq p_i$.

M3: shared resources can also be protected against concurrent access by replicating the communication buffers and by leveraging information on the time instant and order (such as priority and scheduling) of the access to the buffer. For a shared resource ε_i, let n_i^{LR} be the number of reader tasks with priority lower than the writer. As in [Sofronis et al. 2006; Wang et al. 2009], the readers with priority higher than the writer always use only one buffer, each of the n_i^{LR} readers with lower priority than the writer uses one buffer in the worst case, and one buffer is reserved for the writer to store the newest data. Thus, if there is any reader with priority higher than the writer, then the number of additional buffers needed for the wait-free method is $n_i = n_i^{LR} + 2$; otherwise it is $n_i = n_i^{LR} + 1$. We also assume that *the timing overhead associated to the wait-free method is negligible*.

In the following, we first provide an optimal algorithm for the assignment of preemption thresholds to runnables and the selection of the shared variable protection mechanisms when the runnable-to-task mapping, the priority assignment, and the ordering of runnables inside a task are given. Then, we leverage the algorithms for the efficient selection of the other design variables (task priority, runnable to task mapping, and runnable preemption threshold) presented in Section 5, and extend them with the consideration of the protection mechanisms for shared variables and the corresponding memory costs.

6.2. Memory Optimality of Maximum Threshold Assignment

In the absence of timing overhead, the blocking time limit and the maximum preemption threshold assignment for runnables and tasks are the same as in Section 5.1. Properties 1-3 still apply, in addition, we have

— **Property 4 (Independency of the maximum runnable preemption threshold from the mechanism for the protection of the shared resources)**: by (9)–(11), the runnable blocking time limit and maximum preemption threshold are independent from the selection of mechanism to protect the shared resources.

By Property 4, the selection of the mechanisms to protect shared variables does not affect the existence of $\eta^{\max}$, thus Lemma 5.1 and Theorem 5.2 still hold.

After the assignment of the maximum preemption threshold $\eta^{\max}$, a subset of the shared resources is protected because of the threshold configuration (M1), which comes with no memory overhead. For the remaining subset, it is necessary to use either semaphore locks (M2) or wait-free methods (M3). To minimize memory usage, lock-based methods (M2) are always preferable. However, the critical sections protected by semaphore locks may introduce additional blocking time to higher priority tasks and lead to system unfeasibility.

For the critical section in ρ_i accessing ε_k, its *maximum priority ceiling* $\lambda_i(\varepsilon_k)$ is defined as the highest priority ceiling it can get without causing system unfeasibility. $\lambda_i(\varepsilon_k)$ is the highest priority level p_j such that all the tasks with priority between p_i and p_j have a blocking time limit no shorter than $\gamma_i(\varepsilon_k)$ (the duration of the critical section).

$$\lambda_i(\varepsilon_k) = \max\{p_j \in [p_i, p^{\max}(\varepsilon_k)] : \forall p_r \in (p_i, p_j], \gamma_i(\varepsilon_k) \leq h_r\} \tag{13}$$

where $p^{\max}(\varepsilon_k)$ denotes the highest priority among the runnables accessing ε_k. The concept of maximum priority ceiling is similar to maximum preemption threshold, as it defines the highest priority level at runtime during the execution of the code. However, it is defined over the critical section code instead of the entire runnable.

If shared resource ε_k can be protected by semaphore locks, it is necessary that all the critical sections accessing it have a maximum priority ceiling equal to $p^{\max}(\varepsilon_k)$.

$$\varepsilon_k \text{ is lock-protected} \Rightarrow \forall \rho_i \in \rho(\varepsilon_k), \lambda_i(\varepsilon_k) = p^{\max}(\varepsilon_k) \tag{14}$$

We now prove that $\eta^{\max}$ also provides the largest maximum priority ceiling of all critical sections, thus maximizing the opportunity to use lock-based methods.

THEOREM 6.1. *Among all the legal preemption threshold assignments, $\eta^{\max}$ allows for the highest priority ceiling to any critical section.*

PROOF. It directly follows the facts that $\eta^{\max}$ maximizes the blocking time limits of all the tasks at the same time, and by (13) the maximum priority ceiling of a critical section is monotonically increasing with respect to the blocking time limit of higher priority tasks. □

ALGORITHM 4: Optimal Algorithm for Preemption Threshold Assignment and Selection of Data Synchronization Mechanisms

Input: Task set with predefined runnable mapping and priority assignment.
Output: Runnable preemption threshold assignment and selection of data synchronization mechanisms.

```
1  for each $\tau_i$ from highest priority to lowest priority do
2      for each $\rho_{i,k}$ do
3          $\eta_{i,k}$ = maximum preemption threshold as in (9);
4      end
5      $h_i$ = maximum blocking time limit as in (11);
6  end
7  for each resource $\varepsilon_k$ do
8      if $\exists \rho_i \in \rho(\varepsilon_k)$ with $\eta_i < p^{\max}(\varepsilon_k)$ then
9          for each runnable $\rho_i \in \rho(\varepsilon_k)$ do
10             $\lambda_i(\varepsilon_k)$ = maximum priority ceiling as in (13);
11         end
12         if $\forall \rho_i \in \rho(\varepsilon_k)$, $\lambda_i(\varepsilon_k) = p^{\max}(\varepsilon_k)$ then
13             use M2 to protect $\varepsilon_k$;
14         else
15             use M3 to protect $\varepsilon_k$;
16         end
17     end
18 end
```

Theorems 5.2 and 6.1 together provide the memory optimality of the maximum preemption threshold assignment, considering both the stack space requirement and communication buffer usage.

Algorithm 4 is *an optimal procedure to minimize the memory usage given the runnable mapping, priority assignment, and runnable execution order inside tasks*. First, the maximum preemption threshold assignment is calculated starting from the highest priority task to the lowest priority one (lines 1-6). Then, for each resource not protected by preemption thresholds, it checks the possibility of using locks by calculating the maximum priority ceiling for each critical section, and comparing it with the highest priority level among the runnables accessing the resource. If there is a critical section with a priority ceiling lower than the computed bound, a wait-free method (M3) must be used with the additional memory cost. Otherwise, a lock-based protection (M2) is selected with no additional memory cost.

6.3. Extension of Algorithm 3

The optimization algorithm that considers the selection of the protection mechanisms is a simple merger of Algorithm 4 with the previous algorithm for the optimal assignment of priorities and runnable mappings (Algorithm 3). This is because the consideration of data synchronization mechanisms does not affect the runnable and task blocking time limit calculation and the maximum preemption threshold (retaining the optimality of Algorithm 2). The result of the merger is a procedure in which Algorithm 4 is called inside the main body of Algorithm 3 when the initial task model is defined (after line 2): between lines 8 and 9 to set the communication protection mechanisms (using Algorithm 4) before the memory usage of the new candidate solutions are evaluated, and finally after line 18 to update the definition of the protection mechanisms for each new generated solution. As demonstrated in the experimental section, this algorithm produces results similar to those computed by simulated annealing, but in a much shorter time.

Algorithm 4 contains two loops, the first (lines 1-6) is the calculation of the maximum runnable preemption thresholds with a complexity of $O(m^2 \cdot F(m))$; the second is to find the protection mechanism for each resource, with complexity $O(r \cdot m^2)$, where r is the number of shared resources in the system, as line 10 requires at most m checks after the runnable preemption threshold is computed. Hence, the complexity of Algorithm 4 is $O(m^2 \cdot (F(m) + r))$, and the extension of Algorithm 3 to consider the selection of the protection mechanisms has complexity $O(m^4 \cdot (F(m) + r))$.

6.4. Experimental Results

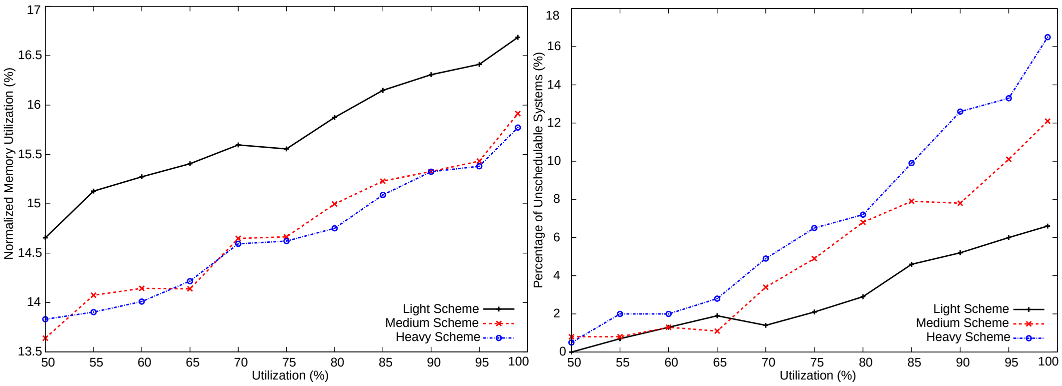
6.4.1. Random Systems. We first generate random systems in which the periods of the runnables are generated in the same way as in Section 5. For communication among runnables, we consider three schemes: light, medium, and heavy. The runnable stack usage is randomly distributed between one to three times of the total size of its communication variables.

In the **light** communication scheme, the output of a runnable is shared with 1 to 3 other runnables, with a probability p of 50%, 40%, and 10% respectively. The size of the output is randomly selected from 1 (with probability $p = 30\%$), 4 ($p = 30\%$), 24 ($p = 20\%$) and 128 ($p = 20\%$) bytes.

In the **medium** communication scheme, the output of a runnable is shared with 1 to 4 readers, with a probably p of 20%, 30%, 30%, and 20% respectively. The size of the output is randomly selected from 1 (with probability $p = 10\%$), 4 ($p = 30\%$), 24 ($p = 30\%$), 128 ($p = 20\%$), and 256 ($p = 10\%$) bytes.

In the **heavy** communication scheme, the output of a runnable is shared by 1-5 other runnables, with a probably p of 10%, 20%, 30%, 30%, and 10% respectively. The size of the output is randomly selected from 1 (with probability $p = 10\%$), 4 ($p = 20\%$), 24 ($p = 20\%$), 48 ($p = 10\%$), 128 ($p = 20\%$), 256 ($p = 10\%$), and 512 ($p = 10\%$) bytes.

The *first experiment* is performed to analyze the possible benefit of finding the optimal combination of data synchronization mechanisms using Algorithm 4 (that is, without optimizing the runnable-to-task mapping and the priority assignment). We use two metrics for comparison: one is the optimal memory usage, normalized with respect to a baseline configuration in which all resources are protected by wait-free methods (M3); the other is on system schedulability, measured by the percentage of systems for which our method find a feasible task and communication model config-



(a) (b)

Fig. 5. (a) Memory usage normalized to the requirement that all resources are protected by wait-free methods; (b) Percentage of unschedulable systems if all resources are protected by semaphore locks.

uration, but would not be schedulable if all resources were protected by semaphore locks (M2). The results for systems with 50 runnables are shown in Figure 5. On average, Algorithm 4 saves about 85% of the memory compared to the one with wait-free methods only, with a slight decrease at higher utilization values (when there are less opportunities for using lock-based methods). Similarly, when utilization increases, our optimization method can leverage the selection of wait-free mechanisms to avoid blocking times and improve schedulability. This results in a significant fraction of systems (up to 18% for very high utilization values) that would not be schedulable if only lock-based methods are used but are feasible using Algorithm 4.

The *second experiment* is to evaluate the quality of the proposed solution for the runnable mapping and task priority assignment when considering communication buffers. For each of the communication scheme, we randomly generate 5500 systems with 10 to 20 runnables, and use the modified Algorithm 3 (Section 6.3) to find the best possible runnable mapping, priority and execution order assignment. As a comparison, we develop a simulated annealing algorithm similar to the one described in Section 5.4 but extended for the consideration of communication variables: after each transition (changing the mapping of a runnable, or changing the priority of a task), the execution order and the preemption threshold of the runnables are computed using the optimal Algorithm 2, then Algorithm 4 is applied to optimize the maximum preemption threshold and selection of data synchronization mechanisms.

In general, Algorithm 3 has comparable quality to simulated annealing. In the **light** communication scheme, for 21 (or 0.38%) of the cases the result is better than simulated annealing, for 19 (or 0.35%) of the cases it is worse, and for the remaining the two have the same memory usage. In the **medium** communication scheme, the result is better than simulated annealing in 22 of the cases, and worse in 17 cases. In the **heavy** communication scheme, it is better in 21 cases and worse in 10 cases. Of course, Algorithm 3 runs much faster than simulated annealing. As shown in Figure 6, the runtime of Algorithm 3 is typically four magnitudes smaller than simulated annealing. Simulated annealing becomes impractical for large systems (expected to be more than 12 hours on average for $n = 40$), while Algorithm 3 is still viable for a number of runnables larger than 100 (about 126 seconds for $n = 100$).

6.4.2. An Automotive Case Study. The last experimental case study consists of an automotive system (as described in [Di Natale et al. 2010]). The system is a fuel injection embedded controller with 90 runnables, executed with 7 different periods (in ms): 4, 5,

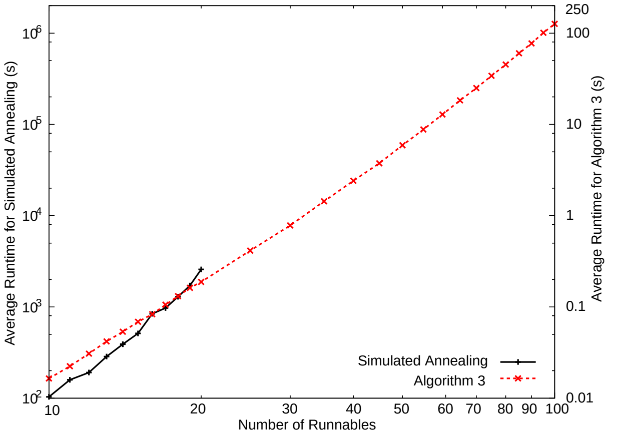


Fig. 6. Runtimes of Simulated Annealing (axis on left) and Algorithm 3 (axis on right).

8, 12, 50, 100, and 1000. The worst-case execution times of some functions (but not all of them) are available from the car electronics supplier as part of the case study. The others are assigned to achieve a system utilization of 94%, which is close to the values found in real systems of this type. The function blocks are communicating through 106 links. The details of the communication graph (including the size of the communication links) and the time parameters of the runnables (including their periods and WCETs) can be found in [Di Natale et al. 2010]. The runnable stack usage (not included in the specification provided by the component supplier) is set to be 8 bytes (a constant value representing the minimum information pushed in the stack as part of the runnable call) plus one to three times the total size of the runnables communication data.

The case study confirms the benefit of finding the optimal combination of data synchronization mechanisms using Algorithm 4. For example, when the duration of the critical section is assumed to be between 1% and 10% of the runnable WCET, the memory usage of the optimal solution drops to 26.5% of the baseline solution (all resources use wait-free methods). At the opposite end of the implementation solutions range, if the critical section is increased to 1%–18% of the runnables WCET, the system becomes unschedulable when resources are all protected by semaphore locks.

7. CONCLUSIONS

In this paper, we discuss the open problems of design synthesis to minimize stack usage for systems with preemption threshold scheduling. We target the optimal 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 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 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

AbsInt. The AiT code analyzer. http://www.absint.com.

AUDSLEY, N. 1991. Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Tech. Rep. YCS 164, Department of Computer Science, University of York, England.

BARUAH, S. 2005. The Limited-Preemption Uniprocessor Scheduling of Sporadic Task Systems. In *Proc. 17th Euromicro Conference on Real-Time Systems*.

BERTOGNA, M., BUTTAZZO, G., AND YAO, G. 2011. Improving Feasibility of Fixed Priority Tasks Using Non-Preemptive Regions. In *Proc. 32nd IEEE Real-Time Systems Symposium*.

BOHLIN, M., HÄNNINEN, K., MÄKI-TURJA, J., CARLSON, J., AND NOLIN, M. 2008. Bounding Shared-Stack Usage in Systems with Offsets and Precedences. In *Proc. 20th Euromicro Conference on Real-Time Systems*.

BRIL, R. J., LUKKIEN, J. J., AND VERHAEGH, W. F. 2009. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Syst. 42,* 1-3, 63–119.

BURNS, A. 1995. Advances in real-time systems. Chapter Preemptive priority-based scheduling: an appropriate engineering approach, 225–248.

BUTTAZZO, G., BERTOGNA, M., AND YAO, G. 2013. Limited Preemptive Scheduling for Real-Time Systems. A Survey. *IEEE Transactions on Industrial Informatics 9,* 1, 3–15.

CHEN, J. AND BURNS, A. 1997. A fully asynchronous reader/write mechanism for multiprocessor real-time systems. Tech. Rep. YCS 288, Department of Computer Science, University of York, England.

CHEN, J., HARJI, A., AND BUHR, P. 2005. Solution Space for Fixed-Priority with Preemption Threshold. In *Proc. 11th IEEE Real Time on Embedded Technology and Applications Symposium*.

DI NATALE, M., GUO, L., ZENG, H., AND SANGIOVANNI-VINCENTELLI, A. 2010. Synthesis of Multitask Implementations of Simulink Models With Minimum Delays. *IEEE Transactions on Industrial Informatics 6,* 4, 637–651.

DI NATALE, M. AND ZENG, H. 2012. Efficient Implementation of AUTOSAR Components with Minimal Memory Usage. *IEEE SIES Conference*.

FERRARI, A., DI NATALE, M., GENTILE, G., REGGIANI, G., AND GAI, P. 2009. Time and memory tradeoffs in the implementation of AUTOSAR components. In *Proc. Conference on Design, Automation and Test in Europe*.

GAI, P., LIPARI, G., AND NATALE, M. D. 2001. Minimizing Memory Utilization of Real-Time Task Sets in Single and Multi-Processor Systems-on-a-Chip. In *Proc. 22nd IEEE Real-Time Systems Symposium*.

GHATTAS, R. AND DEAN, A. G. 2007. Preemption Threshold Scheduling: Stack Optimality, Enhancements and Analysis. In *Proc. 13th IEEE Real Time and Embedded Technology and Applications Symposium*.

HANNINEN, K., MAKI-TURJA, J., BOHLIN, M., CARLSON, J., AND NOLIN, M. 2006. Determining Maximum Stack Usage in Preemptive Shared Stack Systems. In *Proc. 27th IEEE International Real-Time Systems Symposium*.

KOPETZ, H., OBERMAISSER, R., EL SALLOUM, C., AND HUBER, B. 2007. Automotive Software Development for a Multi-Core System-on-a-Chip. In *Proc. 4th International Workshop on Software Engineering for Automotive Systems*.

LONG, R., LI, H., PENG, W., ZHANG, Y., AND ZHAO, M. 2009. An Approach to Optimize Intra-ECU Communication Based on Mapping of AUTOSAR Runnable Entities. In *Proc. International Conference on Embedded Software and Systems*.

Mathworks. The Mathworks Simulink and StateFlow User's Manuals. http://www.mathworks.com.

OSEK 2006. OSEK/VDX operating systems specification, version 2.2.3. http://www.osek-vdx.org.

REGEHR, J. 2002. Scheduling Tasks with Mixed Preemption Relations for Robustness to Timing Faults. In *Proc. 23rd IEEE Real-Time Systems Symposium*.

SAKSENA, M. AND WANG, Y. 2000. Scalable real-time system design using preemption thresholds. In *Proc. 21st IEEE Real-time Systems Symposium*. RTSS'00.

SHA, L., RAJKUMAR, R., AND LEHOCZKY, J. P. 1990. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Trans. Comput. 39,* 9, 1175–1185.

SOFRONIS, C., TRIPAKIS, S., AND CASPI, P. 2006. A memory-optimal buffering protocol for preservation of synchronous semantics under preemptive scheduling. In *Proc. 6th ACM & IEEE International conference on Embedded software*.

The AUTOSAR consortium. The AUTOSAR standard, specification version 4.0. http://www.autosar.org.

W. Lamie. Preemption-threshold. White Paper, Express Logic Inc, http://rtos.com/articles/18833.

WANG, G., DI NATALE, M., AND SANGIOVANNI-VINCENTELLI, A. 2009. Improving the Size of Communication Buffers in Synchronous Models With Time Constraints. *IEEE Transactions on Industrial Informatics 5,* 3, 229–240.

WANG, Q., SONG, J., AND PARMER, G. 2011. Execution Stack Management for Hard Real-Time Computation in a Component-Based OS. In *Proc. 32nd IEEE Real-Time Systems Symposium*.

WANG, Y. AND SAKSENA, M. 1999. Scheduling Fixed-Priority Tasks with Preemption Threshold. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*.

YAO, G. AND BUTTAZZO, G. 2010. Reducing stack with intra-task threshold priorities in real-time systems. In *Proc. 10th ACM International Conference on Embedded Software*.

YAO, G., BUTTAZZO, G., AND BERTOGNA, M. 2009. Bounding the Maximum Length of Non-preemptive Regions under Fixed Priority Scheduling. In *Proc. 15th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*.

ZENG, H. AND DI NATALE, M. 2013. An Efficient Formulation of the Real-Time Feasibility Region for Design Optimization. *IEEE Transactions on Computers 62,* 4, 644–661.

Received October 2012; revised XXX 201X; accepted XXX 201X