Minimizing a real-time task set through Task Clustering

Antoine Bertout, Julien Forget, Richard Olejnik

Minimizing a real-time task set through Task Clustering

Antoine Bertout, Julien Forget
Univ. Lille
LIFL - UMR 8022
F-59650 Villeneuve d'Ascq, France
{antoine.bertout,julien.forget}@lifl.fr

Richard Olejnik
CNRS
LIFL - UMR 8022
F-59650 Villeneuve d'Ascq, France
richard.olejnik@lifl.fr

Abstract

In the industry, real-time systems are specified as a set of hundreds of functionalities with timing constraints. Implementing those functionalities as threads in a one-to-one relation is not realistic due to the overhead caused by the large number of threads. In this paper, we present *task clustering*, which aims at minimizing the number of threads while preserving the schedulability. We prove that our clustering problem is *NP-Hard* and describe a heuristic to tackle it. Our approach has been applied to fixed-task or fixed-job priority based scheduling policies as Deadline Monotonic (DM) or Earliest Deadline First (EDF).

1 Introduction

In this paper, we focus on real-time systems programming. Designs of such systems often consist of several hundreds of high-level functionalities (or computational nodes) with timing constraints. For example, the number of nodes ranges from 500 to 1000 in the flight control system of an aircraft or of a space vehicle [9, 17]. When implementing such systems, real-time programmers can not directly implement each of the nodes as a thread (or task) because real-time operating system usually do not support such a high number of threads. This limitation stems from the fact that having a huge number of tasks in a system induces important overheads, such as time overhead due to context switches [33, 21] and a bigger memory footprint (e.g. task control block, size of the stack, etc.) Thus, to cope with this limitation, real-time developers have to group several nodes together into the same thread. This work is generally performed manually and may be tedious and error prone regarding the number of nodes involved. We are concerned here with the automation of this process.

In our work, we address this question from the scheduling point of view. We model a system as a set of tasks with real-time constraints, where each task is characterized by an execution time, an activation period and a deadline, in the same way as Liu and Layland's task model [24]. With respect to this model, nodes can simply be considered as finer grain tasks, while threads (which may consist of several nodes) are just coarser tasks. Thus, mapping nodes to tasks amounts to gathering several tasks into a single one, which we call *task clustering*. Clustering several tasks implies to choose only one deadline for the cluster, which may reduce some task deadlines. As a consequence, we have to check that the system schedulability is preserved after the clustering.

Related Works In the literature, task clustering is most often studied in the context of distributed systems implementation, where it consists in distributing a set of tasks over a set of computing nodes (processors or cores). This is different from our context, because in the distributed systems context a cluster corresponds to the set of tasks allocated to the same computing resource. For instance, works [29, 1] aim at minimizing communications by clustering tasks that communicate a lot. The approaches in [28, 19] cluster tasks based on communications, in order to reduce the system makespan. The number of tasks of the resulting implementation is however not reduced.

Task clustering is known as *runnable-to-task mapping* and is identified as a step of the development process in the augmented real-time specification for AUTomotive Open System ARchitecture (AUTOSAR) [4]. This document and [33] also provide guidelines defining under which conditions runnables can be grouped to the same tasks. In that context, authors of [25] proposed two heuristics for multicores architecture. The first one allocates runnables to cores considering dependencies, locality constraint and core load. The second one constructs the sequencing of the runnables through one *dispatcher* (or sequencer) task per core. They consider implicit deadlines and fixed task-priority scheduling. In the same context, authors of [38] formulate the task clustering problem as an optimization problem. Authors present a first technique based on mixed integer linear programming (MILP) and a second one based on the genetic algorithms (GA) to optimize end-to-end responses and memory consumption. Runnable-to-task mapping is a step of the work of [41], but it is restricted to functionalities that have deadlines equal to their periods.

In a model-driven development context, authors of [26] aims at reducing the number of priority levels by gouping tasks. They propose a technique based on MILP to reach a specific number of priority levels that the target RTOS can handle. In [13, 27], the authors study the multi-task implementation of multi-periodic synchronous programs and must allocate the different elements of the program to tasks. The clustering is out of the scope of [27], while the heuristic proposed in [13] is very specific to the language structure. The necessity of the task clustering is also emphasized by Zheng and Di Natale in [39]. They claim that mapping each functional node of their system design to a task is not realistic regarding the context switch overheads. Santinelli et al. [32] propose to reduce the number of real-time tasks in a uniprocessor system. They consider tasks with precedence constraints and their main objective is to cope with the high complexity of the schedulability analysis with a large set of tasks. They mostly consider functional requirements and temporal constraints are limited to monoperiodic task sets. Their model is based on latency constraints while we work with multi-periodic system on a deadline-based model.

This research In this paper, we propose a technique to minimize the number of tasks in a real-time system while preserving timing constraints and schedulability. This work aims at exploring the task clustering in simple settings so as to better comprehend the problematic before tackling task clustering with precedences constraints. Unlike existing works, we consider both fixed task-priority and fixed job-priority scheduling policies. We also propose conditions under which tasks can be grouped without affecting the schedulability. This research follows the work we made in [7] where we laid the basis for the task clustering and proposed a first heuristic. In this paper, we mainly develop four improvements. First, we emphasize the necessity of the task clustering by examining the impact of having a huge number of threads. To this intent, we achieved experiments on the effect of the clustering on context switches and preemptions. Second, we prove the *NP-Hardness* of the task clustering. Third, we propose conditions under which clustering does not require further schedulability test. Finally, we prove the *sustainable unschedulability* of classic schedulability tests (basically the reverse property of sustainable schedulability introduced in [10]), which enables us to limit the search space of the heuristic. A slightly different variant of the heuristic consists in stopping the minimization to the number of tasks the RTOS can handle.

Organization The rest of the paper is organized as follows. In Section 2, we inspect the overhead caused by a large number of threads in a real-time operating system. We define the way we cluster tasks in Section 3. Section 4 is dedicated to the influence of the task clustering on the schedulability. The *NP-Hardness* of the task clustering is proved and a heuristic to cope with it is described in Section 5. Section 6 contains experimental results on the task

clustering and evaluation of its impact in terms of context switches and preemptions.

2 Overhead of numerous threads

Designs of real-time embedded systems consist of hundreds of threads. The task clustering aims at grouping a large number of nodes into a reduced set of tasks. In this section, we examine the impact of having a high number of threads.

2.1 Scheduling overhead

2.1.1 Scheduler level

In a real-time operating system, the scheduler is responsible for the allocation of processes (or tasks) on the processing unit by applying real-time scheduling algorithms, such as Deadline Monotonic (DM) [22] or Earliest Deadline First (EDF) [24]. Scheduler implementations may vary, but commonly the scheduler handles two queues: one called the *run/dispatch/ready queue* and the other called the *delay queue*. The ready queue holds the tasks ready to be executed, ordered by priority. The delay queue holds the suspended tasks. A task enters the suspended state when the scheduler switches from that task to another, in other words at a context switch. Manipulating a delay queue with a large number of tasks induces an additional cost (cf. [11]), because at each clock interrupt the delay queue is scanned and tasks are inserted into the ready queue sorted according to their release time.

2.1.2 Context switches and preemptions

The context of a task generally consists at least of registers, a program counter and a stack pointer. Each time the scheduler selects a task to be executed, the system has to store the previous task context and to retrieve the next one. This involves several processor instructions.

A context switch occurs each time a job has completed its execution. It also occurs when a task of high priority is released and interrupts the execution of a lower priority task before its completion. This is called a *preemption*. Thus, a preemption necessarily results in a context switch but not all context switches are due to preemptions. Our experiments in the Section 6.2 show that the clustering highly reduces the total number of context switches but that the number of preemptions remains essentially similar.

2.2 Memory

The memory footprint for having a large number of tasks can be considered at two levels. First, each task obviously consumes its proper portion of memory and second, the scheduler also needs some memory to store task queues.

2.2.1 Task level

The memory footprint of a task depends highly on the real-time operating system implementation. Nevertheless, a task generally consists of a structure (sometimes called Process Control Block (PCB)) with a stack, registers, pointers, its state, etc. The global memory allocated to tasks is usually linear with the number of tasks. We experimented the memory consumption of tasks on the FreeRTOS operating system on a TMS570, a Texas Instrument (TI) microcontroller designed for critical applications. This cost is relatively small but may not be negligible with a large number of tasks on a controller with a limited RAM. For instance, considering the constant memory used by a task, clustering 100 tasks to 10 on the TMS570 amounts to reducing the RAM utilization by approximately 13.5 KB on a total of 160 KB available RAM, that is to say a reduction of approximately 150 bytes per task.

2.2.2 Scheduler level

In the worst case, fixed task-priority policies assign as many priorities as tasks (e.g if all deadlines are distinct). Hence, the number of priorities needed to schedule a task set naturally increases with a large number of tasks. This has a cost because usually scheduler implementations require one queue by level of priority. Moreover, having many unique priority levels leads to an additional amount of stack space as the number of preemptions is high. Nevertheless, mechanisms for limiting preemptions such as non-preemption groups [14] or preemption thresholds [37] can reduce the number or priorities.

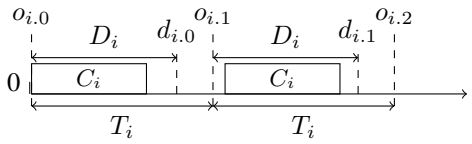


Figure 1: Task Diagram.

3 Task Clustering

In this section, we explain how we cluster tasks. For this purpose, we define our task model and we present the clustering of two tasks.

3.1 Real-time task model

Our model, illustrated in Figure 1, is based on Liu and Layland's model [24]. A system consists of a synchronous (i.e. with offsets equal to zero) set of real-time tasks $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ where C_i is the worst-case execution time (WCET) of τ_i, T_i is the activation period, D_i is the relative deadline with $D_i \leq T_i$. We denote $\tau_{i.k}$ the $(k+1)^{th}$ $(k \geq 0)$ instance, or *job*, of τ_i. The job $\tau_{i.k}$ is released at time $o_{i.k} = kT_i$. Every job $\tau_{i.k}$ must be completed before its absolute deadline $d_{i.k} = o_{i.k} + D_i$

In this paper, we focus on priority-based scheduling policies, either fixed job-priority with EDF or fixed task-priority policies with DM.

Let $\mathcal{J}$ denote the infinite set of job $\mathcal{J} = \{\tau_{i.k}, 1 \leq i \leq n, k \in \mathbb{N}\}$. Given a priority assignment Φ, we define two functions $s_\Phi, e_\Phi : \mathcal{J} \rightarrow \mathbb{N}$, where $s_\Phi(\tau_{i.k})$ is the start time and $e_\Phi(\tau_{i.k})$ is the completion time of $\tau_{i.k}$ in the schedule produced by Φ.

Definition 1 *Let $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$ be a task set and Φ be a priority assignment. S is* schedulable *under Φ if and only if: $\forall \tau_{i.k}, e_\Phi(\tau_{i.k}) \leq d_{i.k} \wedge s_\Phi(\tau_{i.k}) \geq o_{i.k}$*

In the sequel, we will also rely on the notion of *worst-case response time*.

Definition 2 *The worst-case response time R_i indicates the maximum time elapsed between the release of any job of τ_i and its completion.*

3.2 Clustering model

Definition 3 *Clustering tasks τ_i and τ_j, where $D_i \leq D_j$, produces a cluster τ_{ij} with the following parameters:*

$$C_{ij} = C_i + C_j$$
$$T_{ij} = T_i = T_j$$

$$D_{ij} = \begin{cases} D_j & \text{if } ((D_j - C_j \leq D_i) \vee (R_j - C_j \leq D_i)) \quad (1a) \\ D_i & \text{otherwise.} \quad (1b) \end{cases}$$

Notice that, executing τ_{ij} consists in executing sequentially τ_i and then τ_j. In the following, $\tau_{i'}$ and $\tau_{j'}$ denote respectively the parts of τ_i and τ_j in τ_{ij}.

By restriction, we only group tasks with identical periods. In industrial practices, functionalities of different periods are sometimes grouped together, especially when these functionalities interact a lot, to minimize communication as explained in [34]. This possibility makes the clustering more complex because it requires to manage scheduling inside a cluster. For this reason, we do not deal with this option in this paper. Nevertheless, we could relax this assumption via, e.g., hierarchical scheduling [23, 26] or by sequencing tasks offline through a sequencer task similarly to [25].

In the sequel, case (1a) refers to clustering with deadline D_j and case (1b) to clustering with deadline D_i. We choose the cluster deadline D_{ij} (cf. Equations (1a) and (1b)) in such way that both tasks of the cluster still respect their initial deadlines after clustering. This is stated formally as follows:

Theorem 1 *Let $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ be a synchronous task set and two tasks τ_x and τ_y with $D_x \leq D_y$ and $T_x = T_y$. Let Φ be a priority assignment and $\mathcal{S}' = (\{\mathcal{S} \setminus \{\tau_x, \tau_y\}\} \cup \tau_{xy})$.*

$\mathcal{S}'$ is schedulable under $\Phi \Rightarrow \mathcal{S}$ is schedulable under Φ.

Proof 1 *(Case* (1a)*)*
Trivially, $\tau_{j'}$ respects its initial deadline D_j. $\tau_{i'}$ respects

its initial deadline D_i when $D_j - C_j \leq D_i$:

$$\begin{aligned}\mathcal{S}' \text{ schedulable under } \Phi &\Rightarrow \forall \tau_{ij.k}, e_\Phi(\tau_{ij.k}) \leq d_{ij.k} \\ &\Rightarrow e_\Phi(\tau_{i'.k}) + C_j \leq d_{ij.k} \\ \text{As } D_j \leq D_i + C_j,\ & e_\Phi(\tau_{i'.k}) + C_j \leq d_{i.k} + C_j \\ \text{And thus } & e_\Phi(\tau_{i'.k}) \leq d_{i.k}\end{aligned}$$

When $R_j - C_j \leq D_i$:

$$\begin{aligned}&\mathcal{S}' \text{ schedulable under } \Phi \\ &\Rightarrow \forall \tau_{ij.k}, e_\Phi(\tau_{ij.k}) \leq o_{ij.k} + R_{ij} \\ &e_\Phi(\tau_{i'.k}) + C_j \leq o_{ij.k} + R_{ij} \\ \text{As } R_{ij} = R_j,\ & e_\Phi(\tau_{i'.k}) + C_j \leq o_{ij.k} + R_j \\ \text{As } R_j \leq D_i + C_j,\ & e_\Phi(\tau_{i'.k}) + C_j \leq o_{ij.k} + D_i + C_j \\ &e_\Phi(\tau_{i'.k}) + C_j \leq d_{i.k} + C_j \\ \text{And thus } & e_\Phi(\tau_{i'.k}) \leq d_{i.k}\end{aligned}$$

(Case (1b)*)*

Trivial.

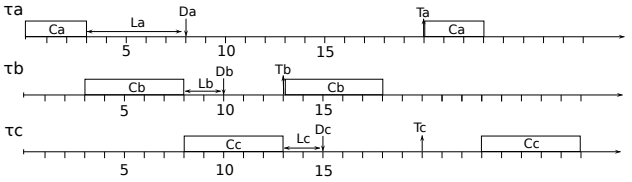
4 Cluster Schedulability

In this section, we first stress that a task set may become unschedulable after a clustering. Then, we present existing schedulability tests to assess whether a task set is still schedulable after a clustering or not. Finally, we determine the conditions under which schedulability does not need to be re-checked after clustering.

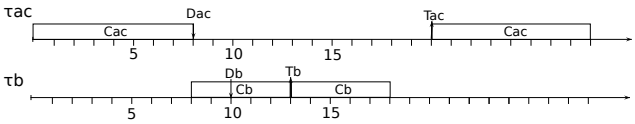
4.1 Impact on schedulability

Remark 1 *In case* (1b)*, $\mathcal{S}$ schedulable under $\Phi \not\Rightarrow \mathcal{S}'$ schedulable under Φ.*

A schedulable system might become non schedulable after clustering under the minimum deadline of the two tasks, as illustrated in Figure 2. Indeed, we notice in Subfigure 2(b) that the task τ_b misses its first deadline after the clustering of tasks τ_a and τ_c. Thus, we must check the resulting task set schedulability after clustering.



(a) Initial schedulable system of tasks τ_a,τ_b and τ_c under DM.



(b) Resulting unschedulable system after clustering of tasks τ_a and τ_c.

Figure 2: Influence of task clustering on system schedulability.

In the following, we consider that a clustering that leads to a unschedulable task set is not valid:

Definition 4 *Let $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$ be a task set and τ_x and τ_y be two tasks of $\mathcal{S}$ such that $D_x \leq D_y$. We say that τ_{xy} is a* valid cluster *if and only if the task set obtained after clustering is schedulable.*

4.2 Schedulability tests

In this section, we review existing schedulability tests that can be used during clustering for DM and EDF scheduling policies. First we highlight their complexity and then, we examine if they may provide a clue of the remaining "schedulabiliy margin" of the task set, i.e. how much more we can reduce deadlines during further clustering before the task set becomes unschedulable.

A schedulability test is called sufficient if all task sets considered schedulable by the test are actually schedulable. In the same manner, a schedulability test is called necessary if all task sets considered unschedulable by the test are in fact unschedulable. Schedulability tests that are both sufficient and necessary are referred to as exact.

We only consider exact or sufficient tests, which ensures that the task sets obtained after clustering are schedulable. Indeed, applying sufficient tests means that we might not get the minimum number of clusters but we are sure to obtain a valid clustering. Remember that we work with synchronous (with offsets equal to zero) task sets that have constrained deadlines (i.e. with $D_i \leq T_i$).

4.2.1 Exact schedulability tests

In the same manner as in [15], we distinguish two types of tests: *Boolean schedulability tests* and *response time tests*. On the one hand, Boolean tests give a Boolean answer, determining only whether a task set is schedulable or not, for instance with Processor Demand Criteria (PDC [6]) or Quick convergence Processor-demand Analysis (QPA) [40]. On the other hand, exact tests based on Response Time Analysis (RTA [20, 2]) provide the worst response time for each task.

RTA of a task τ_i is based on the concept of level-i *busy period* that is presented more fully in Section 5.1.1. RTA for fixed task-priority (FTP) systems as DM can be performed with a pseudo-polynomial time algorithm. On the contrary to FTP systems, the worst response time is not necessarily found on the first processor busy period in a task set scheduled by EDF [36]. Thus, computing RTA for EDF is more complex and has an exponential complexity.

4.2.2 Sufficient schedulability conditions

In order to reduce the complexity of the computations, we also considered linear sufficient schedulability tests. Audsley [3] and Devi [16] propose sufficient but not necessary schedulability tests, respectively for DM and EDF in $\mathcal{O}(n)$ complexity. Those two sufficient tests actually provide an approximate worst response time for each task. They can be considered as an approximate RTA analysis.

4.3 Zero-cost clustering

In this section, we prove that in the case 1a, a clustering preserves the schedulability of the task set. A clustering that can be done without re-running a schedulability test is called a *zero-cost* clustering.

Theorem 2 *Let $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ be a synchronous task set and two tasks τ_x and τ_y with $D_x \leq D_y$, $T_x = T_y$ and $(D_y - C_y \leq D_x) \vee (R_y - C_y \leq D_x)$. Let Φ be a priority assignment and $\mathcal{S}' = (\{\mathcal{S} \setminus \{\tau_x, \tau_y\}\} \cup \tau_{xy})$.*

In the case (1a)*, $\mathcal{S}$ schedulable under $\Phi \Leftrightarrow \mathcal{S}'$ schedulable under Φ.*

Proof 2 *(Only if part)*

By definition, $D_i \leq D_j$.
$\mathcal{S}$ schedulable under $\Phi \Rightarrow \forall \tau_{i.k}, e_\Phi(\tau_{i.k}) \leq d_{i.k}$
$\wedge\, \forall \tau_{j.k}, e_\Phi(\tau_{j.k}) \leq d_{j.k}$
From (1a)*, $d_{i.k} \leq d_{ij.k} \wedge d_{ij.k} = d_{j.k}$*
The rest of the proof follows.

(If part) Proved in Theorem 1.

5 Minimizing the number of tasks

After having defined how to correctly cluster tasks, we now go to the heart of the matter: finding the minimum schedulable task set using task clustering. In this section, we present the theoretical complexity of the problem, a heuristic to tackle it and some results related to the sustainable (un)schedulability tests that serve this heuristic.

5.1 Sustainable unschedulability

In this section, we prove that a task set that is unschedulable remains so after clustering. This allows us to cut the search space when looking for the minimum valid cluster.

Burns and Baruah [10] defined the notion of *sustainable schedulability*, which means that a task set deemed schedulable by a schedulability test remains so with "better" timing constraints (e.g increased deadlines and periods, decreased execution time). If so, this test is considered sustainable. In a similar manner, we study in this section how an unschedulable task set behaves, in "worse" conditions, with decreased deadlines, periods or increased execution times. We examine the sustainable unschedulability in the context of uniprocessor preemptive scheduling, with synchronous tasks under fixed task-priority and fixed job-priority assignment.

5.1.1 Fixed task-priority assignment

In the sequel, we rely on the existing exact test based on RTA. Worst response time R_i of a task τ_i is based on the concept of level-i *busy period*. Intuitively, the level-i busy period is the maximum continuous time interval during

which a processor executes tasks of higher or equal priority to the priority of the considered task τ_i. RTA computes for each task its worst response time denoted by R_i. Equation for finding R_i is based on the fact that, if a task τ_i is released at time t where t is a critical instant, there must be time for τ_i and for every higher priority job to complete in the interval $(t, Ri]$. The equation for R_i is solved by the following recurrence relation:

$$w_i^{n+1} = \sum_{j \in hp(i)} \left\lceil \frac{w_i^n}{T_j} \right\rceil \times C_j \quad (2)$$

For all i, $hp(i)$ denotes the set of higher priority tasks than τ_i. The recurrence holds until $w_i^{n+1} = w_i^n$ and then $R_i = w_i^{n+1}$. Notice that w_i^0 can be initialized to C_i and that the task set is not schedulable if a task τ_i exists such that $R_i > D_i$.

Theorem 3 (Sustainable unschedulability) *A task set deemed unschedulable with RTA remains so with smaller relative deadlines.*

Proof 3 *By contrapositive. Let $\mathcal{S}$ be a task set deemed unschedulable by RTA. Let $\mathcal{S}'$ be a task set identical to $\mathcal{S}$ except that some tasks have shorter deadlines. Assume that $\mathcal{S}'$ is schedulable. As RTA is sustainable, this would mean that $\mathcal{S}$ is schedulable (because $\mathcal{S}$ is obtained by increasing some deadlines of $\mathcal{S}'$). Thus we have a contradiction.*

Lemma 1 *A system that is unschedulable with RTA remains so with smaller periods and longer execution times.*

Proof 4 *Considering the RTA, we can trivially observe in Equation* (2) *that decreasing a period T_i and increasing C_i will increase R_i.*

5.1.2 Fixed job-priority assignment

In this section, we use the method PDC that provides an exact test for a task set scheduled by Earliest Deadline First (EDF) to prove the *sustainable unschedulability* for fixed job-priority assignment. The PDC is based on the processor demand bound function (dbf). For synchronous task set, $dbf(t)$ corresponds to the cumulative execution requirement by the jobs that have their absolute deadlines before or at t. It is given by the following formula:

$$\mathrm{dbf}(t) = \sum_{i=1}^{n} \left\lfloor \frac{t + T_i - D_i}{T_i} \right\rfloor \times C_i \quad (3)$$

A synchronous task set is EDF-schedulable iff the demand bound never exceeds the available time t:

$$\forall t \geq 0, \mathrm{dbf}(t) \leq t \quad (4)$$

Theorem 4 *A system using deemed unschedulable with PDC remains so with smaller deadlines, smaller periods and longer execution times.*

Proof 5 *Simply observe that in Equation* (3) *the function* dbf *is monotonically non-decreasing with respect to decreasing D_i, decreasing T_i and increasing C_i.*

5.2 Task Clustering is NP-Hard

In [7], we gave a practical idea of the high complexity of the task clustering. Our experiments showed that an exhaustive search is not achievable due to the exponential number of partitions to explore. Indeed, the number of partitions to assess to find the minimum task set is equal to $\prod_{i=0}^{m} B_i$ where B_i is the exponential Bell number [30] of the set of tasks with period i and m is the number of different periods of the task set. However, we did not address the problem in terms of computational complexity theory. Thus, we prove here that our problem is *NP-hard*.

Let us first define the clustering problem more formally. Let $\mathcal{S} = (\{\tau_i(C_i, D_i, T_i)\}_{1 \leq i \leq n})$ be a synchronous task set. Let $\mathcal{C}(\mathcal{S})$ denote the set of schedulable task sets obtained by recursive valid task clusterings of $\mathcal{S}$. The problem we want to address is the following:

Given $\mathcal{S}$, find the task set with minimal cardinality in $\mathcal{C}(\mathcal{S})$.

Notice that this problem is defined with respect to a given schedulability test. In the following we call *RTA-clustering* the problem defined with respect to RTA (fixed-task priorities) and *DBF-clustering* the problem defined with respect to DBF (fixed job-priorities).

We first recall the bin-packing problem, and its variant, the *bin-packing with fragile objects* (BPFO) introduced in [5], which we will reduce to our problem.

Bin-packing In the bin-packing problem, objects of different sizes must be packed into a set of bins. Let a finite set of objects $\{p_i\}_{1 \leq i \leq n}$ of size a_i and a finite set of bins $\{b_j\}_{1 \leq j \leq m}$ of size v_j, with $a_i, v_j \in \mathbb{Z}+$. Let B_k denote the set of objects assigned to bin k. Each bin content must not exceed its capacity: $\forall k, 1 \leq k \leq m, \forall p_i \in B_k, \sum a_i \leq v_k$. The bin-packing problem (decision version) is stated as follows:

Is it possible to place all the n objects into m bins respecting the necessary condition cited above for each bin ?

The optimization version of this problem consists in computing the minimum number of bins in which we can fit the objects. Bin-packing is known to be *NP-Complete* in the decision version and *NP-Hard* in its optimization version. BPFO is a more general version of the Bin-packing where each object has a fragility f_i and the validity condition is replaced by: $\forall k, 1 \leq k \leq m, \forall p_i \in B_k, \sum a_i \leq min\{f_i\}$.

Theorem 5 *RTA-clustering and DBF-clustering are* NP-Hard.

Proof 6 *To prove the NP-hardness, we reduce BPFO to our problem by showing that any instance of BPFO corresponds to an instance of our problem. Given an instance of BPFO, we make the following correspondences: each object p_i corresponds to a task τ_i, f_i corresponds to D_i, a_i corresponds to C_i and a bin B_k corresponds to a cluster τ_k.*

Now, checking the fragility condition does not directly correspond to checking clusters validity. Indeed, this would correspond to checking that $\forall k, 1 \leq k \leq m, \forall \tau_i \in \tau_k, \sum C_i \leq min\{D_i\}$, which is not a sufficient validity (schedulability) condition. However, both RTA and DBF tests require to check at some point properties of the form $f(\{C_i\}, \ldots) \leq g(\{D_i\}, \ldots)$ where f and g are functions with higher complexity than in the validity condition of BPFO.

5.3 Heuristic

In this section, we detail our approach for minimizing the size of the initial task set by successive clustering. Due to the *NP-Hardness* of the problem, we rely on a heuristic instead of an exact algorithm.

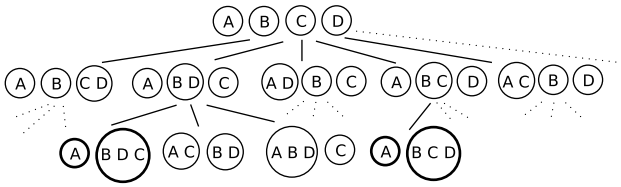


Figure 3: Recursive generation of partitions.

5.3.1 Principle

We start from an initial task set where each task is considered a cluster with one element, we gradually try to group more and more clusters together to minimize the cardinality of the task set, as depicted in the example of Figure 3. At each step, we try to group each cluster with each other and we have several candidates that fulfill conditions of a valid cluster. We must select the best candidate among them. This is done using a heuristic cost (or evaluation) function that estimates which candidate will most likely lead to the best clustering. The sustainable unschedulability allows us to not consider descendants of unschedulable task sets. We propose to achieve task clustering using classic optimization heuristics based on cost functions.

5.3.2 Cost function

We need a schedulability test to determine valid clustering because grouping tasks may make the resulting task set more and more difficult to schedule due to reduced deadlines. Moreover, we need a relevant heuristic cost function to determine the best candidate for the clustering. Thus, we want a schedulability test that exhibits some features that might allow us to compare the potential of two task sets. Therefore, in this section, we explore the compatibility of the tests presented in Section 4.2 with a heuristic based on a cost function.

Boolean exact tests only give a Boolean answer on the schedulability of a task set. Thus, they do not exhibit any clear feature that could be considered a heuristic cost function. Exact tests based on RTA gives worst response times for each task while sufficient tests for DM and EDF presented in Section 4.2.2 are based on a pessimistic approximation of the RTA. Considering a task τ_k with its worst response time denoted R_k, the closer to one $\frac{R_k}{D_k}$ is, the less we have margin to group the task τ_k with another.

Thus, we can use the sum of each task response time divided by its respective deadline as heuristic cost function. Then, we have a heuristic cost function $h(\mathcal{S})$, such that

$$h(\mathcal{S}) = \sum_{k=0}^{|\mathcal{S}|} \frac{R_k}{D_k}$$

5.3.3 Algorithm

Several heuristics based on a cost function exist such as greedy best-first search (greedy BFS), A* algorithm, simulated annealing, etc. We do not aim in this paper at comparing their different performances. We chose a simple heuristic based on greedy BFS [31] detailed in Algorithm 1. The choice of the heuristic (BFS here) is not central in this work. The main idea is the heuristic cost function that may also be applied with other heuristics, as those cited above.

As described in Algorithm 1, we recursively enumerate partitions. At each recursive call, we first try to apply a zero-cost clustering on each generated child. If the zero-cost condition is respected we make a recursive call with the new cluster, if not, we accumulate a 3-tuple containing the task set and indices of the two tasks we want to group in a buffer. Finally, if no zero-cost clustering has been made during this step of partitions enumeration, we choose the most promising task set with non zero-cost clustering. The most promising child is selected according to the heuristic cost function of Section 5.3.2. Notice that, as we cluster one more task by recursive call, we can easily stop the algorithm if a target number of tasks is reached instead of searching for the minimum task set. Practically, it consists in introducing a counter of the number of recursive calls and to stop the recursion as soon as the desired number of tasks is reached.

Lemma 2 *The complexity of Algorithm 1 with linear tests is $\mathcal{O}(n^4)$ and pseudo-polynomial with pseudo-polynomial tests (RTA for DM).*

Proof 7 *The number of children (or direct successors) generated from a partition of i elements is equal to $i \times (i-1)/2$. We only explore one among all visited children at each step with our greedy heuristic. Thus, the maximum number of visited partitions is equal to $\sum_{i=0}^{n} \frac{i\times(i-1)}{2}$. This sum corresponds to the sum of the first n triangular numbers (also called tetrahedral numbers) and its closed-form expression is $f(n) = \frac{n(n+1)(n+2)}{6}$ [35]. Hence, this sequence complexity is $\mathcal{O}(n^3)$. We apply a sufficient schedulability test in $\mathcal{O}(n)$ complexity (whether with DM or EDF) on each visited partition, so the heuristic complexity is $\mathcal{O}(n^3) \times \mathcal{O}(n) = \mathcal{O}(n^4)$. In a similar way, applying schedulability tests with a pseudo-polynomial complexity gives a pseudo-polynomial complexity to the whole algorithm.*

6 Experiments

6.1 Task set generation

We chose the following model to generate random task sets:

- U_i: each task utilization ($\frac{C_i}{T_i}$) is computed following the classic UUnifast [8] method. We denote as u the overall utilization factor of the processor.
- T_i: each task period is uniformly distributed between a set of a maximum of 10 different periods by task set using method [18], ensuring that the simulation can be limited to a reasonable hyper-period
- $C_i = T_i \times U_i$
- $D_i = round((T_i - C_i) \times rand(d1, d2)) + C_i$ with $0 \leq d1 \leq d2$. This computation comes from [18] and use the following functions: $rand(d1, d2)$ which returns a pseudo-random real number uniformly distributed in the interval $[d1, d2]$ and $round(x)$ which returns the closest integer to x. We notice that $d1 = d2 = 1$ corresponds to implicit deadlines and $d1 \leq d2 = 1$ to constrained deadlines.

6.2 Context switches and preemptions

We simulate the schedule to obtain the total number of context switches using the scheduling simulator SimSo [12] and our task clustering heuristic to evaluate the number of context switches and preemptions before and after task clustering under DM. Our experiments did

Algorithm 1 Task clustering algorithm
Function clustering(S)

Require: $\mathcal{S} = (\{\tau_i\}_{1 \leq i \leq n})$: initial set of tasks in non-decreasing deadline order

```
minSumTests ← n + 1
minSet ← null
childrenBuffer ← null

//Try zero-cost clustering.
for i = n − 1 to 0 do
  for j = i − 1 to 0 do
    if T_i == T_j then
      if (D_j − C_j ≤ D_i) ∨ (R_j − C_j ≤ D_i) then
        S' ← {S \ {τ_i, τ_j}} ∪ τ_ij
        return clustering(S')
      else
        childrenBuffer ←
        {childrenBuffer ∪ (S, i, j)}
      end if
    end if
  end for
end for

// If no zero-cost clustering found, find most promising child.
for all (M, x, y) ∈ childrenBuffer do
  if C_x + C_y ≤ min(D_x, D_y) then //laxity
    M' ← {M \ {τ_x, τ_y}} ∪ τ_xy
    if schedulable(M') then
      if h(M') < minSumTests then
        minSumTests ← h(M')
        minSet ← M'
      end if
    end if
  end if
end for

if minSet ≠ null then
  return clustering(minSet) //continue with best child
else
  return S
end if
```

	Nb of Tasks	Total Ctx Switches	Preemptions
After clustering	-93%	-92.5%	+0.2%

Table 1: Evolution of the total number of context switches and preemptions after task clustering under DM.

not exhibit the clear main factors (utilization factor, deadlines, number of tasks, etc) that impact the number of context switches and preemptions. Thus, we use the following settings. The number of tasks is fixed to 200. Results are averaged on 1000 executions. Deadlines are uniformly distributed between bounds $d1 = 0$ and $d2 = 1$ and the utilization factor of the processor is randomly generated between 0.20 and 0.80 for each generated task set. We observe in Table 1 that the total number of context switches is reduced on average by 90% and that the number of preemptions remains similar after clustering.

6.3 Number of tasks

In this section we present the number of tasks obtained after clustering. We cannot compare our heuristic with an optimal solution because the task clustering is not achievable with an exhaustive search among all partitions. Instead we compare our results with our previous work [7]. We also present the part of zero-cost clustering performed in the clustering.

Task sets range from 50 to 300 tasks for DM or 200 tasks for EDF by step of 50 tasks. Maximum utilization factor is fixed at 0.80 and deadlines are uniformly distributed between bounds $d1 = 0$ and $d2 = 1$. We only take into account task sets that are initially schedulable. We compute average results by executing several times the heuristic on randomly generated task sets with the same parameters. We observe in Figure 4(a) that we are able to cluster many tasks under DM. Results are slightly better than in our previous work and most of the clustering is performed by zero-cost clustering as showed in Figure 5(a). Consequently, this results in an important gain of execution time. For example, experiments conducted on a 2.3GHz Intel Core i7 quad-core with 4GByte memory show that we are able to cluster a set of 1000 tasks in a few seconds while clustering an initial set of 500 tasks took more than one hour of computation in our previous work. Nonetheless, compared to our previous work, results un-

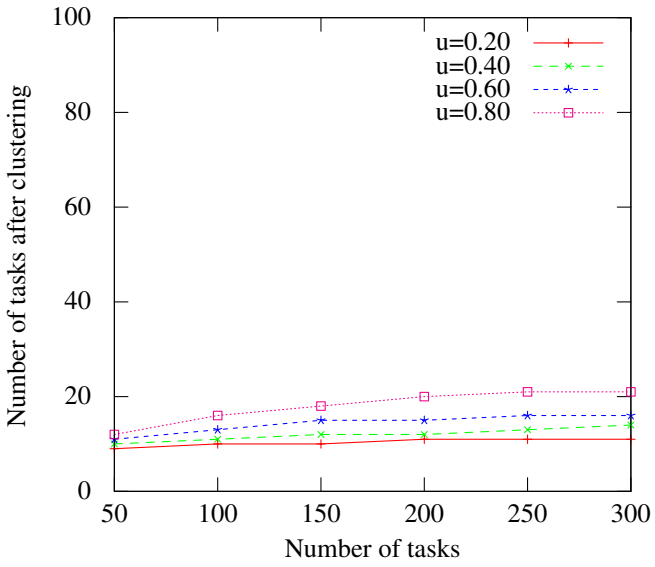
der EDF observed in Figure 4(b) are more encouraging. This difference probably comes from the pessimism of the sufficient test under EDF used in the previous work, especially when the utilization factor was high. It also comes from the efficiency of the zero-cost clustering that highly reduces the number of tasks. Thus, the RTA for EDF that has a high complexity is more able to handle this restricted number of tasks. Nevertheless, experiments shows that we can not get results in a reasonable time from 200 tasks with the RTA for EDF. In the case of larger task set, the sufficient test may be preferred. The results of clustering under DM and EDF are quite similar. Finally notice that globally, the higher the utilization factor is, the less the tasks are clustered.

7 Conclusion and Future work

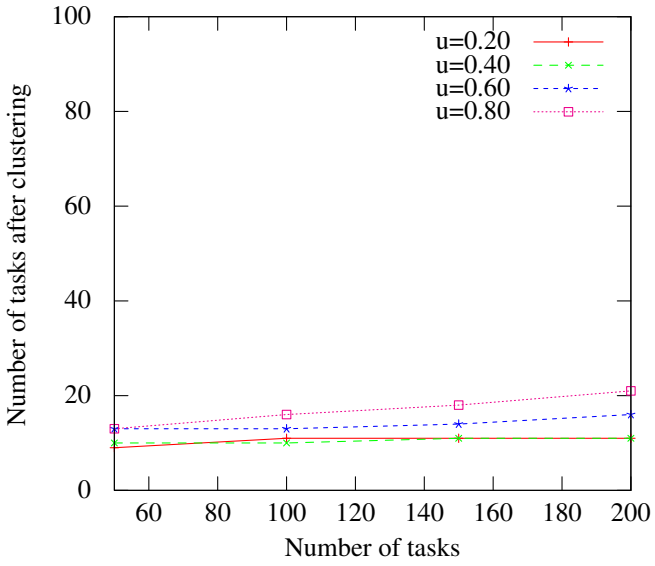
In this paper, we studied the impact of implementing a real-time system as numerous threads. We emphasized that clustering several functionalities in the same thread highly reduces the number of context switches and, to a lesser extent, also reduces memory consumption. We proposed a heuristic to minimize the number of threads of the implementation, while preserving timing constraints and schedulability. This heuristic improves over our previous work, thanks to the zero-cost clustering and sustainable unschedulability that enables us to limit the search space. We presented experimental results of this heuristic under DM and EDF scheduling policies. Concerning future works, we would like to propose solutions based on classic optimization techniques such as mixed integer linear programming (MILP). We are currently working on task clustering applied to tasks with precedence constraints and plan to extend that work to a multi-processor setting.

8 Acknowledgments

We would like to thank Mélanie Lelaure for her experiments on the TMS570 microcontroller carried out during her internship. We also thank Sanjoy Baruah and Robert I. Davis for their valuable advice during the last RTNS conference.

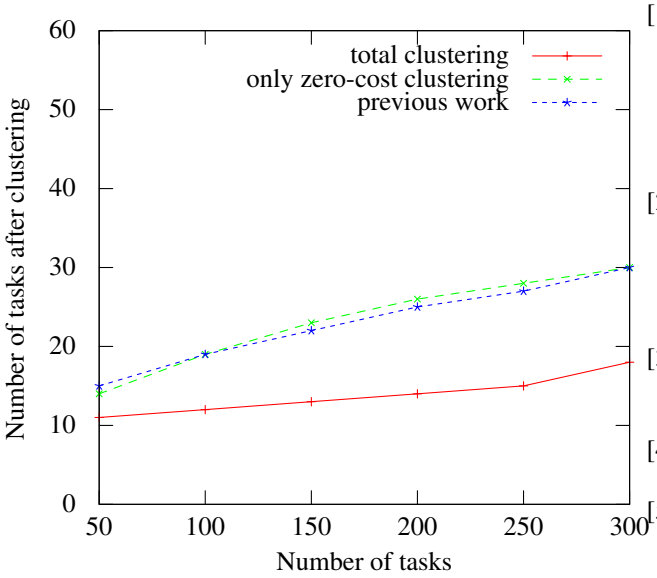


(a) Task clustering under DM.

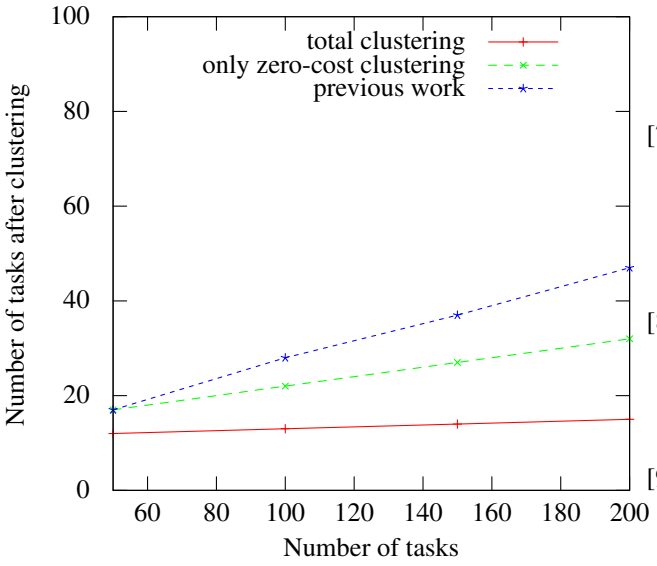


(b) Task clustering under EDF.

Figure 4: Results of task clustering.



(a) Task clustering under DM.



(b) Task clustering under EDF.

Figure 5: Average results on task clustering.

References

[1] A. Ahmadinia, C. Bobda, and J. Teich. Temporal task clustering for online placement on reconfigurable hardware. In *Field-Programmable Technology (FPT), 2003. Proceedings. 2003 IEEE International Conference on*, pages 359 – 362, Dec. 2003.

[2] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A. J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[3] N. C. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings. *Deadline monotonic scheduling*. 1990.

[4] AUTOSAR. *RTE Standard Specifications*.

[5] N. Bansal, Z. Liu, and A. Sankar. Bin-packing with fragile objects. In *IFIP TCS*, pages 38–46, 2002.

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

[7] A. Bertout, J. Forget, and R. Olejnik. A heuristic to minimize the cardinality of a real-time task set by automated task clustering. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing (SAC 2014)*, Gyeongju, Korea, Mar. 2014.

[8] E. Bini and G. Buttazzo. Biasing effects in schedulability measures. In *16th Euromicro Conference on Real-Time Systems, 2004. ECRTS 2004. Proceedings*, pages 196 – 203, July 2004.

[9] F. Boniol, P.-E. Hladik, C. Pagetti, F. Aspro, and V. Jégu. A framework for distributing real-time functions. In *Proceedings of the 6th international conference on Formal Modeling and Analysis of Timed Systems*, FORMATS '08, pages 155–169. Springer-Verlag, 2008.

[10] A. Burns and S. K. Baruah. Sustainability in real-time scheduling. *JCSE*, 2(1):74–97, 2008.

[11] A. Burns and A. Wellings. *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*, chapter 11. Addison-Wesley Educational Publishers Inc, 2009.

[12] M. Chéramy, P.-E. Hladik, and A.-M. Déplanche. SimSo: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In *Proceedings of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, 2014.

[13] A. Curic. *Implementing Lustre Programs on Distributed Platforms with Real-time Constrains*. PhD thesis, University Joseph Fourier, Grenoble, 2005.

[14] R. I. Davis, N. Merriam, and N. J. Tracey. How embedded applications using an rtos can stay within on-chip memory limits. In *Proceedings Work in Progress and Industrial Experience Sessions, Euromicro Conference on Real-Time Systems (ECRTS)*, 2000.

[15] R. I. Davis, A. Zabos, and A. Burns. Efficient exact schedulability tests for fixed priority real-time systems. *Computers, IEEE Transactions on*, 57(9):1261–1276, 2008.

[16] U. Devi. An improved schedulability test for uniprocessor periodic task systems. In *Real-Time Systems, 2003. Proceedings. 15th Euromicro Conference on*, pages 23 – 30, july 2003.

[17] J. Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. PhD thesis, Université de Toulouse, 2009.

[18] J. Goossens and C. Macq. Limitation of the hyper-period in real-time periodic task set generation. In *In Proceedings of the RTS Embedded System (RTS'01*, pages 13–147, 2001.

[19] L. Guodong, C. Daoxu, W. Daming, and Z. Defu. Task clustering and scheduling to multiprocessors with duplication. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 8 pp., Apr. 2003.

[20] M. Joseph and P. Pandya. Finding response times in a real-time system. *The Computer Journal*, 29(5):390–395, 1986.

[21] E. Lee. The problem with threads. *Computer*, 39(5):33–42, 2006.

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

[23] G. Lipari and E. Bini. A methodology for designing hierarchical scheduling systems. *Journal of Embedded Computing*, 1(2):257–269, 2005.

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

[25] A. Monot, N. Navet, B. Bavoux, and F. Simonot-Lion. Multisource software on multicore automotive ecuscombining runnable sequencing with task scheduling. *Industrial Electronics, IEEE Transactions on*, 59(10):3934–3942, 2012.

[26] R. Mzid, C. Mraidha, A. Mehiaoui, S. Tucci-Piergiovanni, J.-P. Babau, and M. Abid. Dpmp: a software pattern for real-time tasks merge. In *Modelling Foundations and Applications*, pages 101–117. Springer, 2013.

[27] C. Pagetti, J. Forget, F. Boniol, M. Cordovilla, and D. Lesens. Multi-task implementation of multi-periodic synchronous programs. *Discrete Event Dynamic Systems*, 21(3):307–338, 2011.

[28] M. Palis, J.-C. Liou, and D. Wei. Task clustering and scheduling for distributed memory parallel architectures. *Parallel and Distributed Systems, IEEE Transactions on*, 7(1):46 –55, Jan. 1996.

[29] K. Ramamritham. Allocation and scheduling of precedence-related periodic tasks. *IEEE Trans. Parallel Distrib. Syst.*, 6(4):412–420, Apr. 1995.

[30] G.-C. Rota. The number of partitions of a set. *The American Mathematical Monthly*, 71(5):498–504, 1964.

[31] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, pages 94–95. Prentice Hall, 2 edition, 2003.

[32] L. Santinelli, W. Puffitsch, A. Dumerat, F. Boniol, C. Pagetti, and J. Victor. A grouping approach to task scheduling with functional and non-functional requirements. In *International Conference on Embedded Real Time Software and Systems (ERTS)*, 2014.

[33] O. Scheickl and M. Rudorfer. Automotive real time development using a timing-augmented AUTOSAR specification. *Proceedings of ERTS2008*, 4, 2008.

[34] S. Schliecker, J. Rox, M. Negrean, K. Richter, M. Jersak, and R. Ernst. System level performance analysis for real-time automotive multicore and network architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):979 –992, July 2009.

[35] N. J. A. Sloane. The On-Line Encyclopedia of Integer Sequences. A000292.

[36] M. Spuri. Analysis of deadline scheduled real-time systems. 1996.

[37] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Real-Time Computing Systems and Applications, 1999. RTCSA'99. Sixth International Conference on*, pages 328–335. IEEE, 1999.

[38] E. Wozniak, A. Mehiaoui, C. Mraidha, S. Tucci-Piergiovanni, and S. Gerard. An optimization approach for the synthesis of autosar architectures. In *Emerging Technologies & Factory Automation (ETFA), 2013 IEEE 18th Conference on*, pages 1–10. IEEE, 2013.

[39] H. Zeng and M. Di Natale. Schedulability analysis of periodic tasks implementing synchronous finite state machines. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 353–362. IEEE, 2012.

[40] F. Zhang and A. Burns. Schedulability analysis for real-time systems with edf scheduling. *Computers, IEEE Transactions on*, 58(9):1250–1258, 2009.

[41] M. Zhang and Z. Gu. Optimization issues in mapping AUTOSAR components to distributed multi-threaded implementations. In *2011 22nd IEEE International Symposium on Rapid System Prototyping (RSP)*, pages 23 –29, May 2011.