Hindawi Publishing Corporation
Advances in Operations Research
Volume 2011, Article ID 561794, 19 pages
doi:10.1155/2011/561794

Research Article

Optimizing Distributed Real-Time Embedded System Handling Dependence and Several Strict Periodicity Constraints

Omar Kermia

AOSTE Team, INRIA Rocquencourt, BP 105, 78153 Le Chesnay Cedex, France

Correspondence should be addressed to Omar Kermia, omar.kermia@gmail.com

Received 16 March 2011; Revised 8 June 2011; Accepted 4 July 2011

Academic Editor: Xiaoqiang Cai

This paper focuses on real-time nonpreemptive multiprocessor scheduling with precedence and strict periodicity constraints. Since this problem is NP-hard, there exist several approaches to resolve it. In addition, because of periodicity constraints our problem stands for a decision problem which consists in determining if, a solution exists or not. Therefore, the first criterion on which the proposed heuristic is evaluated is its schedulability. Then, the second criterion on which the proposed heuristic is evaluated is its execution time. Hence, we performed a schedulability analysis which leads to a necessary and sufficient schedulability condition for determining whether a task satisfies its precedence and periodicity constraints on a processor where others tasks have already been scheduled. We also present two multiperiodic applications.

1. Introduction

Hard Real-Time problematic is to maintain temporal and functional achievement of systems execution. Hard real-time scheduling has been concerned with providing guarantees for temporal feasibility of task execution whatever the situations. A scheduling algorithm is defined as a set of rules defining the execution of tasks at system run-time. It is provided thanks to a schedulability analysis, which determines, whether a set of tasks with parameters describing their temporal behavior will meet their temporal constraints if executed at run-time according to the rules of the scheduling algorithm. The result of such a test is typically a yes or a no answer indicating whether a solution will be found or not. These schemes and tests demand precise assumptions about task properties, which hold for the entire system lifetime.

In order to assist the designers, scientists at INRIA proposed a methodology called AAA (Algorithm Architecture Adequation) and its associated system-level CAD software called SynDEx. They cover the whole development cycle, from the specification of the application functions, to their implementation running in real-time on a distributed architecture composed of processors and specific integrated circuits. AAA/SynDEx provides a formal framework based on graphs and graph transformation. On the one hand, they are used to specify the functions of the applications, the distributed resources in terms of processors, and/or specific integrated circuit and communication media, the nonfunctional requirements such as temporal criteria. On the other hand, they assist the designer in implementing the functions onto the resources while satisfying timing requirements and, as much as possible, minimizing the resources. This is achieved through a graphical environment which allows the user to explore manually and/or automatically, using optimization heuristics, the design space solutions. Exploration is mainly carried out through real-time scheduling analysis and timing functional simulations. Their results predict the real-time behavior of the application functions executed onto the various resources, that is, processors, integrated circuits, and communication media. This approach conforms to the typical hardware/software codesign process. Finally, the code that is automatically generated as a dedicated real-time executive, or as a configuration file for a resident real-time operating system such as Osek and RTlinux, [1, 2], details the AAA methodology and SynDEx.

In practice, periodic tasks are commonly found in applications such as avionics and process control when accurate control requires continual sampling and processing of data. Such applications based on automatic control and/or signal processing algorithms are usually specified with block-diagrams. They are composed of functions producing and consuming data, and each function can start its execution as soon as the data it consumes are available, and the cost of a task scheduling is a constant included in its worst case execution time (WCET). This periodicity constraint is the same as the one we find in the Liu and Layland model [3]. A data transfer between producer and consumer tasks leads to precedence constraints that the scheduling must satisfy. In systems we deal with, besides periodicity and precedence constraints, some tasks must be repeated according to a strict period. These tasks represent sensors and actuators by interacting with the environment surrounding the system and for which no data exchanged with this environment is lost. As data produced by the environment to the system are consumed strict periodically on the one hand and data expected by the environment from the system with a strict periodically way on the other hand, sensors and actuators must be executed at a strict periods [4]. In order to satisfy the strict periodicity of these tasks, we consider that all the system tasks have a strict period.

Strict period means that if the periodic task t_i has period $T(t_i)$ then for all $j \in \mathbb{N}^*$, $(S(t_{i_{j+1}}) - S(t_{i_j})) = T(t_i)$, where $t_{i_{j+1}}$ and t_{i_j} are the jth and the $(j+1)$th instances of the task t_i, and $S(t_{i_j})$ and $S(t_{i_{j+1}})$ are their start times [5]. Notice that t_{i_j} is the jth instance or repetition of the periodic task t_i.

The multiprocessor real-time scheduling problem with precedences constraints, but without periodicity constraints, whatever the number of processors, has always a solution. The distributed schedule length, corresponding to the total execution time (Makespan), determines the solution quality. On the contrary, when periodicity constraints must be satisfied the problem may not have a solution. In other words, an application with precedence and periodicity constraints is either schedulable or not.

Thus, this paper discusses three mains goals:

(i) the precedence and periodicity constraints must be satisfied. This is achieved by repeating every task according to its period onto the same processor, and receiving all the data a task needs before it executes. In comparison with the previous AAA/SynDEx, additional steps are required to assign the tasks to the processors and to add missing precedence before being distributed and scheduled;

(ii) distributed architectures involve interprocessor communications the cost of which must be taken into account accurately, as explained in [6]. These communications must be handled even when communicating tasks have different periods;

(iii) since the target architecture is embedded, it is necessary to minimize the total execution time in the one hand to insure that feedback control is correct, and in the other hand to minimize the resource allocation.

As it is mentioned previously, we are interested in nonpreemptive scheduling. This choice is motivated by a variety of reasons including [7]:

(i) in many practical real-time scheduling problems such as I/O scheduling, properties of device hardware and software either make preemption impossible or prohibitively expensive. The preemption cost is either not taken into account or still not really controlled;

(ii) nonpreemptive scheduling algorithms are easier to implement than preemptive algorithms and can exhibit dramatically lower overhead at run-time;

(iii) the overhead of preemptive algorithms is more difficult to characterize and predict than that of nonpreemptive algorithms. Since scheduling overhead is often ignored in scheduling models, an implementation of a nonpreemptive scheduler will be closer to the formal model than an implementation of a preemptive scheduler.

For these reasons, designers often use nonpreemptive approaches, even though elegant theoretical results on preemptive approaches do not extend easily to them [8].

1.1. Related Work

According to the multiprocessor scheduling scheme (partitioned or global), the use of these algorithms changes:

(i) in partitioned multiprocessor scheduling, it is necessary to have a scheduling algorithm for every processor and an allocation (distribution) algorithm is used to allocate tasks to processors (Bin Packing) which implies heuristics utilization. Schedulability conditions for RM and EDF in multiprocessor were given;

(ii) in global multiprocessor scheduling, there is only one scheduling algorithm for all the processors. A task is put in a queue that is shared by all processors and, since migration is allowed, a preempted task can return to the queue to be allocated to another processor. Tasks are chosen in the queue according to the unique scheduling algorithm, and executed on the processor which is idle. Schedulability conditions for RM and EDF in multiprocessor were given in the case of preemptive tasks.

It is well known that the Liu and Layland results, in terms of optimality and schedulability condition, break down on multiprocessor systems [9]. Dhall and Liu [10] gave examples of task sets for which global RM and EDF scheduling can fail at very low processor utilization, essentially leaving almost one processor idle nearly all of the time. Reasoning from such examples is tempting to conjecture that perhaps RM and EDF are not good or efficient scheduling policies for multiprocessor systems even if, at the moment, these conclusions have not been formally justified [11].

In addition, these scheduling algorithms consider a unique independent tasks which means that tasks do not exchange data, whereas these dependences are specified in our applications by a direct graph of tasks.

In [12] tasks are preemptive but dependent except that only tasks with the same period can communicate. AAA/SynDEx handles dependences between tasks with multiple periods by transforming the initial graph.

1.2. Model

We deal with systems of real-time tasks with precedence and strict periodicity constraints. A task t_i (denoted in this paper as $(t_i : C(t_i), T(t_i))$ is characterized by a period $T(t_i)$, a worst case execution time $C(t_i)$ with $C(t_i) \le T(t_i)$, and a start time $S(t_i)$.

The precedences between tasks are represented by a directed acyclic graph (DAG) denoted G such that $G = (\mathbb{V}, \mathbb{E})$. $\mathbb{V}$ is the set of tasks characterized as above, and $\mathbb{E} \subseteq \mathbb{V} \times \mathbb{V}$ the set of edges which represents the precedence (dependence) constraints between tasks. Therefore, the directed pair of tasks $(t_i, t_j) \in \mathbb{E}$ means that t_j must be scheduled, only if t_i was already scheduled, and thus we have $S(t_i) + C(t_i) \le S(t_j)$.

We assume that periods and WCETs are multiple of a unit of time $U = 1$ which means that they are integers representing, for example, some cycles of the processor clock. If a task t_i with execution time $C(t_i)$ is said to start at time unit $S(t_i)$, it starts at the beginning of time unit $S(t_i)$ and completes before the beginning of time unit $S(t_i) + C(t_i)$. Thus the time interval where task t_i is executed is $[S(t_i), S(t_i) + C(t_i)[$.

2. Real-Time Nonpreemptive Scheduling with Precedence and Strict Periodicity Constraints

In order to satisfy these constraints, the heuristic we propose is divided into three algorithms. These algorithms are called Assignment, Unrolling, and Scheduling.

2.1. Assignment

Among the two approaches mentioned below for solving multiprocessor scheduling problems, we chose the partitioned one. Consequently, the first algorithm of the heuristic consists in partitioning tasks over the architecture processors. This first algorithm is called Assignment referring that each task is assigned to a processor such that all tasks assigned to one processor are schedulable on this processor. If a task is schedulable on several processor, it will be assigned to every processor.

This algorithm determines if the system is schedulable when all tasks were assigned to the processors, or not schedulable if at least one task was not assigned. The assignment algorithm uses the outcome of the following scheduling analysis.

2.1.1. Scheduling Analysis

The schedulability analysis consists in verifying that a task could be scheduled with other tasks already proved schedulable using the same schedulability analysis. “A schedulable task" means that it exists one or several time intervals on which this task can be scheduled, that is, its period and periods of already schedulable tasks are satisfied.

The following theorem gives a necessary and sufficient condition for scheduling two tasks.

Theorem 2.1. *Two tasks* $(t_i : C(t_i), T(t_i))$ *and* $(t_j : C(t_j), T(t_j))$ *are schedulable if and only if*

$$C(t_i) + C(t_j) \leq GCD(T(t_i), T(t_j)). \tag{2.1}$$

Proof. Let $g = \mathrm{GCD}(T(t_i), T(t_j))$ (GCD denotes the greatest common divisor). We start by proving that (2.1) is a sufficient condition. Let us assume that t_i and t_j are schedulable and that $S(t_i) = 0$ and $S(t_j) = C(t_i)$. Thus, each instance of the task t_i is executed within an interval belonging to the set of intervals I_1, such that $I_1 = \{\forall k \in \mathbb{N}, [kT(t_i), kT(t_i) + C(t_i)[\}$ and each instance of the task t_j is executed within an interval belonging to the set of intervals I_2, such that $I_2 = \{\forall k \in \mathbb{N}, [kT(t_j) + C(t_i), kT(t_j) + C(t_i) + C(t_j)[\}$.

We merge the notation g in intervals I_1 and I_2. I_1 may be rewritten in this way: $I_1 = \{\forall k \in \mathbb{N}, [gk(T(t_i)/g), gk(T(t_i)/g) + C(t_i)[\}$, and if n ($n \in \mathbb{N}$) such that $n = k(T(t_j)/g)$, then we obtain $I_1 = \{\forall k \in \mathbb{N}, [ng, ng + C(t_i)[\}$. Similarly, we find that (m is an integer such as $m = k(T(t_j)/g)$) $I_2 = \{\forall k \in \mathbb{N}, [mg + C(t_i), mg + C(t_i) + C(t_j)[\}$. The assumption made at the beginning (t_i and t_j are schedulable) implies that no intervals belonging to I_1 and I_2 overlap. We notice that the starts of the intervals of the set I_1 represent multiples of g. On the other hand, the ends of the intervals of the set I_2 represent a multiple of $(g + C(t_i) + C(t_j))$. Intervals of the sets I_1 and I_2 do not overlap which means that if $(n = m+1)$ then $mg + C(t_i) + C(t_j) \leq ng$, which is equivalent to $C(t_i) + C(t_j) \leq g$. This proves the sufficiency of the condition 1.

In order to prove the necessity of 1, we show that if $C(t_i) + C(t_j) < g$ then tasks t_i and t_j are schedulable. This is equivalent to show that if tasks t_i and t_j are not schedulable then $C(t_i) + C(t_j) \leq g$. Without any loss of generality, we assume that $S(t_i) = 0$.

t_i and t_j are not schedulable means that two integers x and y exist such that

$$[xT(t_i), xT(t_i) + C(t_i)[\cap [S(t_j) + yT(t_j), S(t_j) + yT(t_j) + C(t_j)[\neq \emptyset; \tag{2.2}$$

this is equivalent to

$$[xT(t_i) - yT(t_j), xT(t_i) - yT(t_j) + C(t_i)[\cap [S(t_j), S(t_j) + C(t_j)[\neq \emptyset. \tag{2.3}$$

According to Bezout theorem, two integers p and q exist such that $pT(t_i) + qT(t_j) = g$. By taking $x = zp$ and $y = -zq$, $z \in \mathbb{N}$ we have

$$[zg, zg + C(t_i)[\cap [S(t_j), S(t_j) + C(t_j)[\neq \emptyset. \tag{2.4}$$

This latter is true if the length of the empty intervals between the intervals $[zg, zg + C(t_i)[$, $z \in \mathbb{N}$ (which is equal to $g - C(t_i)$)) is less than the length of the intervals $[S(t_j), S(t_j) + C(t_j)[$(which is equal to $C(t_j)$). That is, $g - C(t_i) < C(t_j)$. This condition is equivalent to $g < C(t_i) + C(t_j)$. This concludes the proof of Theorem 2.1. □

Now we are interested in the schedulability of a set of tasks (more than two). Let us introduce the following property for a given tasks set.

Definition 2.2. If a set of n task $\{\forall i \in \mathbb{N}, i \leq n, (t_i : C(t_i), T(t_i))\}$ such that for each two tasks $(t_a : C(t_a), T(t_a))$ and $(t_b : C(t_b), T(t_b))$, $\mathrm{GCD}(T(t_a), T(t_b))$ is the same, then this set is said to satisfy the SGCD property (SGCD for Same Greater Common Divisor).

The following theorem introduces a necessary and sufficient condition for a set of tasks satisfying the SGCD property.

Theorem 2.3. *Tasks of a set $\{\forall i \in \mathbb{N}, i \leq n, (t_i : C(t_i), T(t_i))\}$, which satisfies the SGCD property, are schedulable if and only if*

$$\sum_{i=0}^{n} C(t_i) \leq g; \tag{2.5}$$

g is the GCD of any pair of tasks from this set.

Proof. In order to prove the sufficiency, we proceed by the same way as in the proof of Theorem 2.3. Let us assume that tasks of the set $\{i \in \mathbb{N}^*, i \leq n, t_i\}$ are schedulable and that $S(t_1) = 0$ and $S(t_i) = \sum_{j=1}^{i-1} C(t_j)$. Thus each instance of the task t_i $(i \in \mathbb{N}^*,\ i \leq n)$ is executed in an interval belonging to the intervals set $I_i = \{\forall k \in \mathbb{N}, [kT(t_i) + S(t_i), kT(t_i) + S(t_i) + C(t_i)[\}$. I_i may be rewritten in $I_i = \{\forall k \in \mathbb{N}, [gk(T(t_i)/g) + S(t_i), k(T(t_i)/g) + S(t_i) + C(t_i)[\}$. If n is an integer such that $n = kT(t_i)/g$, then we obtain $I_i = \{\forall k \in \mathbb{N}, [ng + S(t_i), ng + S(t_i) + C(t_i)[\}$. The assumption made at the beginning (tasks t_i are schedulable) implies that no intervals belonging to the sets I_i $(i \in \mathbb{N}^*,\ i \leq n)$ do not overlap. We notice that the starts of the intervals of the set I_i represent multiples of g. On one hand, the ends of the intervals of the set I_2 represent a multiple of g. On the other hand, the ends of the intervals of the set I_2 represent a multiple of $g + \sum_{i=1}^{n} C(t_i)$. Intervals of the sets I_i do not overlap which means that for $S(t_i)$ maximal then $ng + S(t_i) + C(t_i) \leq (n+1)g + S(t_1)$, which is equivalent to $S(t_i) + C(t_i) \leq g$. As $\max(S(t_i),\ i \in \mathbb{N}^*) = S(t_n)$ we deduce that $\sum_{i=1}^{n} C(t_i) \leq g$. This proves the sufficiency of the condition 2.

We prove the necessity of condition 2 by showing that if $\sum_{i=1}^{n} C(t_i) \leq g$ then tasks of the set $\{\forall i \in \mathbb{N}^*$ and $i \leq n, t_i\}$ are schedulable. This is equivalent to show that if tasks of the set $\{\forall i \in \mathbb{N}^*$ and $i \leq n, t_i\}$ are not schedulable, then $g < \sum_{i=1}^{n} C(t_i)$. For this we use the proof by induction.

The Base Case

For a set of two tasks this condition was proved in Theorem 2.1.

The Inductive Step

We show that if the condition 2 is valid for the set $\{\forall i \in \mathbb{N}$ and $i \leq (n-1), t_i\}$ then it is the same for the task $\{\forall i \in \mathbb{N}$ and $i \leq n, t_i\}$.

Tasks of the set $\{\forall i \in \mathbb{N} \text{ and } i \leq n, t_i\}$ are not schedulable which means that integers $x_1, x_2, \dots, x_n$ exist such that

$$\begin{aligned} &([S(t_1) + x_1 T(t_1), S(t_1) + x_1 T(t_1) + C(t_1)[\\ &\quad \cup \cdots \cup [S(t_{n-1}) + x_{n-1} T(t_{n-1}), S(t_{n-1}) + x_{n-1} T(t_{n-1}) + C(t_{n-1})[) \\ &\qquad \cap [S(t_n) + x_n T(t_n), S(t_n) + x_n T(t_n) + C(t_n)[\neq \emptyset, \end{aligned} \tag{2.6}$$

which is equivalent to

$$\begin{aligned} &([S(t_1) + x_1 T(t_1), S(t_1) + x_1 T(t_1) + C(t_1)[\cap [S(t_n) + x_n T(t_n), S(t_n) + x_n T(t_n) + C(t_n)[) \\ &\quad \cup \cdots \cup ([S(t_{n-1}) + x_{n-1} T(t_{n-1}), S(t_{n-1}) + x_{n-1} T(t_{n-1}) + C(t_{n-1})[\\ &\quad \cap [S(t_n) + x_n T(t_n), S(t_n) + x_n T(t_n) + C(t_n)[) \neq \emptyset. \end{aligned} \tag{2.7}$$

We can rewrite this latter in the following way:

$$\begin{aligned} &([S(t_1) + x_1 T(t_1) - x_n T(t_n), S(t_1) + x_1 T(t_1) - x_n T(t_n) + C(t_1)[\cap [S(t_n), S(t_n) + C(t_n)[) \\ &\quad \cup \cdots \cup ([S(t_{n-1}) + x_{n-1} T(t_{n-1}) - x_n T(t_n), S(t_{n-1}) + x_{n-1} T(t_{n-1}) - x_n T(t_n) + C(t_{n-1})[\\ &\quad \cap [S(t_n), S(t_n) + C(t_n)[) \neq \emptyset. \end{aligned} \tag{2.8}$$

In addition to, according to Bezout's theorem, pairs of integers (p_i, q_i) such that $p_i T(t_i) + q_i T(t_n) = g$ $(i = 1, \dots, n-1)$, with $x_i = l_i p_i$ and $x_n = -l_i q_i$ $(l_i, \dots, l_n \in \mathbb{N})$, we obtain

$$\begin{aligned} &\big([S(t_1) + l_1 g, S(t_1) + l_1 g + C(t_1)\big[\cap [S(t_n), S(t_n) + C(t_n)[\big) \\ &\quad \cup \cdots \cup \big(\big[S(t_{n-1}) + l_{n-1} g, S(t_{n-1}) + l_{n-1} g + C(t_{n-1})\big[\cap [S(t_n), S(t_n) + C(t_n)[\big) \neq \emptyset. \end{aligned} \tag{2.9}$$

This may be rewritten in

$$\begin{aligned} &\big([S(t_1) + l_1 g, S(t_1) + l_1 g + C(t_1)\big[\cup \cdots \cup \big[S(t_{n-1}) + l_{n-1} g, S(t_{n-1}) + l_{n-1} g + C(t_{n-1})\big[\big) \\ &\quad \cap [S(t_n), S(t_n) + C(t_n)[\neq \emptyset. \end{aligned} \tag{2.10}$$

This latter is true if the sum of the lengths of empty intervals between the intervals$([S(t_1) + l_1 g, S(t_1) + l_1 g + C(t_1)[\cup \cdots \cup [S(t_{n-1}) + l_{n-1} g, S(t_{n-1}) + l_{n-1} g + C(t_{n-1})[)$ (which is equal to $(g - \sum_{i=1}^{n-1} C(t_i)))$ is less than the length of intervals $[S(t_n), S(t_n) + C(t_n)[$ (which is equal to $C(t_n)$),that is, $(g - \sum_{i=1}^{n-1} C(t_i)) < C(t_n)$ which is equivalent to $g < \sum_{i=1}^{n} C(t_i)$. This concludes the proof of Theorem 2.3. □

Theorem 2.3 gives a schedulability condition for the tasks which satisfy the SGCD property (introduced in Definition 2.2). Nevertheless, we need a condition for all tasks whatever their periods are. Unfortunately, this condition does not exist because of the complexity of the problem [13]. As an alternative, we propose the following reasoning.

We choose to give a condition which allows to assign one task to a processor where a set of tasks has already been assigned to. This task is called candidate task and once it is assigned another task, among tasks which are not assigned yet, becomes the candidate task. To be assigned to a processor, the candidate task and tasks already assigned to this processor must be schedulable on the same processor.

First, we grouped already assigned tasks according to the SGCD property into sets and, second, looked for a condition which takes into account the candidate and each set of already assigned tasks.

Before going further we need to introduce the notion of "identical tasks". Two tasks are said to be identical if they have the same period and the same WCET even though they do not perform the same function.

The following theorem introduce an equation allowing to compute the number of the tasks which are identical to the candidate task and can be assigned to a processor. On this processor, a set of tasks satisfying the SGCD property have already been assigned.

Theorem 2.4. *Let* $(t_{\text{cdt}} : C(t_{\text{cdt}}), T(t_{\text{cdt}}))$ *be the candidate task and* $\{\forall i \in \mathbb{N}^*$ *and* $i \leq n, (t_i : C(t_i), T(t_i))\}$ *the set of already assigned tasks which satisfy the SGCD property. The number of identical tasks to the candidate task which can be assigned to this processor is given by*

$$\Psi(t_{\text{cdt}}) = \frac{T(t_{\text{cdt}})}{\text{GCD}(g, T(t_{\text{cdt}}))} \left\lfloor \frac{\text{GCD}(g, T(t_{\text{cdt}})) - \sum_{i=1}^{n} C(t_i)}{C(t_{\text{cdt}})} \right\rfloor. \tag{2.11}$$

Proof. $\lfloor (\text{GCD}(g, T(t_{\text{cdt}})) - \sum_{i=1}^{n} C(t_i))/C(t_{\text{cdt}}) \rfloor$ that represents the number of identical tasks to the candidate task can be scheduled in one interval of length g. $T(t_{\text{cdt}})/\text{GCD}(g, T(t_{\text{cdt}}))$ represents the number of intervals of length g in one interval of length $T(t_{\text{cdt}})$. □

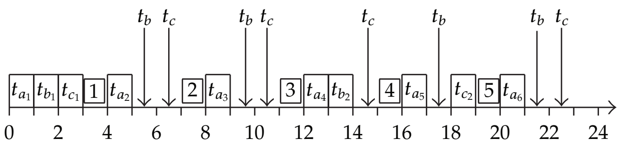
Example 2.5. Let $(t_a : 1, 4)$, $(t_b : 1, 12)$, and $(t_c : 1, 16)$ be three tasks already assigned. We look for assigning a task $(t_d : 1, 20)$. By using Theorem 2.4, we compute the number of tasks identical to t_d which can be assigned.

First we check that tasks t_a, t_b, t_c, and t_d satisfy the SGCD property. $\text{GCD}(T(t_a), T(t_b)) = \text{GCD}(T(t_b), T(t_c)) = \text{GCD}(T(t_c), T(t_a)) = \text{GCD}(T(t_c), T(t_d)) = \text{GCD}(T(t_b), T(t_d)) = \text{GCD}(T(t_a), T(t_d)) = 4$.

Then, $C(t_a) + C(t_b) + C(t_c) + C(t_d) = 4 \leq 4$ proves that the condition of Theorem 2.3 is satisfied.

Finally, $\Psi(t_d) = 20/4 * \lfloor (4-3)/1 \rfloor = 5$, which means that 5 tasks identical to t_d can be scheduled on this processor.

Figure 1 shows the 5 intervals where 5 identical tasks to t_d can be scheduled (these intervals are numbered from 1 to 5). We show also intervals where t_d cannot be scheduled whereas they are empty, and for each interval the tasks (among t_a, t_b, and t_c) cause the nonschedulability, for example, if t_b is mentioned then it means that t_d cannot be scheduled on this interval because one of the instances of t_b and one of the instances of t_d will be scheduled on the same interval which is not allowed.



t_b t_c t_b t_c t_c t_b t_b t_c
t_{a_1} t_{b_1} t_{c_1} 1 t_{a_2} 2 t_{a_3} 3 t_{a_4} t_{b_2} 4 t_{a_5} t_{c_2} 5 t_{a_6}
0 2 4 6 8 10 12 14 16 18 20 22 24

Figure 1: Several scheduling possibilities of task t_d (Example 2.5).

Let us check, on Figure 1, the result obtained by the previous calculation. We notice that if we divide the time axe in intervals of length equal to GCD = 4 and divide each interval in subintervals of length 1, then it is always the fourth subinterval, which is used to schedule t_d. This means that the order established in the first interval of length GCD = 4 is repeated and observed.

Definition 2.6. We denote by Ω the set of tasks which have already been assigned to a processor. We divide the set Ω into several subsets $\{\Omega_1, \Omega_2, \dots, \Omega_v\}$ such that the tasks of each subset and the task t_{cdt} satisfy the SGCD property (introduced in Definition 2.2). Each subset Ω_j is characterized by a greater common divisor (noticed g_{Ω_j}), and we consider that Ω_1 is the subset with the smallest greater common divisor (which is denoted **BG**). The sum off all execution time of a tasks set Ω_j is denoted by $C(\Omega_j) = \sum_{t_i \in \Omega_j} C(t_i)$.

Now in order to know if t_{cdt} is schedulable or not on a processor where other tasks have already been scheduled, we apply the following algorithm:

(1) choose a processor;

(2) set up sets $(\forall j)$, Ω_j;

(3) compute $\Psi_{\Omega_1}(t_{\text{cdt}})$. This represent the identical tasks to t_{cdt} which can be scheduled by taking into account only Ω_1;

(4) for each subset $(j > 1)$, Ω_j remove to the result obtained in the previous step the number of identical tasks to t_{cdt} which cannot be scheduled because of nonschedulability with tasks belonging to Ω_j (the next theorem gives a way to compute this number);

(5) following the reached result, we decide to assign t_{cdt} to this processor or try to assign it to another processor (go to 1).

Let $\Gamma_{\Omega_j}(t_{\text{cdt}})$ $(j \neq 1)$ represent the number of identical tasks to t_{cdt} that cannot be scheduled because of a non-schedulability with tasks belonging to Ω_j $(j \neq 1)$.

The following theorem gives an equation to compute Γ.

Theorem 2.7. *Let t_{cdt} be the candidate task, $\Omega = \{\Omega_1, \Omega_2, \dots, \Omega_v\}$ the set of already scheduled tasks, and g_{Ω_j} the GCD of all Ω_j. For $(j > 1)$, $\Gamma_{\Omega_j}(t_{\text{cdt}})$ is given by*

$$\Gamma_{\Omega_j}(t_{\text{cdt}}) = \frac{T(t_{\text{cdt}})}{g_{\Omega_j}} \left\lceil \frac{\sum_{t_i \in \Omega_j}}{C(t_{\text{cdt}})} \right\rceil \alpha \tag{2.12}$$

such that

$$\alpha = \begin{cases} 1 & \text{if } \dfrac{T(t_{\text{cdt}})}{g_{\Omega_j}} > 1, \\ 0 & \text{if } \dfrac{T(t_{\text{cdt}})}{g\Omega_j} = 1 \text{ and } \displaystyle\sum_{t_i \in \Omega} C(t_i) < g_{\Omega_{j-1}} - C(t_{\text{cdt}}). \end{cases} \tag{2.13}$$

Proof. $T(t_{\text{cdt}})/g_{\Omega_j}$ is the number of intervals of length g_{Ω_j} in an interval of length $T(t_{\text{cdt}})$. This number is multiplied by $\lceil \sum_{t_i \in \Omega_j} / C(t_{\text{cdt}}) \rceil$ which represents the number of tasks identical to t_{cdt} that cannot be scheduled inside an interval the length of which is equal to g_{Ω_j}. And also, in order to take into account the tasks of the other sets, we introduce the value of α. □

Now we bring together the different results obtained previously to find out an equation which allows to compute the number of schedulable tasks identical to t_{cdt}, and hence deduce if t_{cdt} is schedulable or not.

Corollary 2.8. *Let t_{cdt} be the candidate task and $\Omega = \{\Omega_1, \Omega_2, \ldots, \Omega_v\}$ the set of already scheduled tasks. The number of identical tasks to t_{cdt} which can be scheduled is given by*

$$\Psi_{\Omega_1}(t_{\text{cdt}}) - \sum_{j=2}^{v} \Gamma_{\Omega_j}(t_{\text{cdt}}) \tag{2.14}$$

such that

$$\Psi_{\Omega_1}(t_{\text{cdt}}) = \frac{T(t_{\text{cdt}})}{\mathbf{BG}} \left\lfloor \frac{\mathbf{BG} - C(\Omega_1)}{C(t_{\text{cdt}})} \right\rfloor. \tag{2.15}$$

Example 2.9. In order to illustrate the proposed method, we propose the following example: let $(t_{\text{cdt}} : 2, 30)$ to the candidate task to the schedulability analysis. Let $\Omega = \{(t_a : 2, 5), (t_b : 1, 10), (t_c : 1, 20), (t_d : 1, 30), (t_e : 1, 60)\}$ the set of tasks already proved schedulable. In this case, $\mathbf{BG} = \text{GCD}(T(t_a), T(t_b), T(t_c), T(t_d), T(t_e), T(t_{\text{cdt}})) = 5$

(i) the three subsets that we can set up from the set Ω and the task t_{cdt} according to the SGCD property are

(1) $\Omega_1 = \{t_{\text{cdt}}, t_a\}$ with the GCD of periods equal to 5,
(2) $\Omega_2 = \{t_{\text{cdt}}, t_b, t_c\}$ with the GCD of periods equal to 10,
(3) $\Omega_3 = \{t_{\text{cdt}}, t_d, t_e\}$ with the GCD of periods equal to 30.

(ii) From Corollary 2.8,

$$\Psi_{\Omega_1} = \frac{30}{5} \left\lfloor \frac{5-2}{2} \right\rfloor = 6, \tag{2.16}$$

(iii) $\Gamma = \Gamma_{\Omega_2} + \Gamma_{\Omega_3}$,

(1) $\Gamma_{\Omega_2} = ((30/10)\lceil 3/2 \rceil) = 3$,
(2) by the same way $\Gamma_{\Omega_3} = 0$, so $\Gamma = 3 + 0 = 3$,

(iv) then $\Psi_{\Omega_1} - \Gamma = 6 - 3 = 3$, from this, we deduce that t_{cdt} is schedulable.

Remark 2.10. The previous result allows us to assign several tasks (identical tasks to t_{cdt}) at the same times, and if other identical tasks to t_{cdt} are not assigned then it means that these tasks are not schedulable on this processor and they must be assigned to another processor.

Remark 2.11. Notice that, throughout this schedulability analysis, in the given schedulability condition nothing is mentioned about precedence constraints, whereas we allow it in the tasks model. Indeed, a system with precedence constraints but without any periodicity constraint is always schedulable. The next theorem demonstrates that for a set of proved schedulable tasks, that is, satisfying periodicity constraints, it always exists a scheduling of these tasks which satisfies precedences between them, whatever the precedences are.

Theorem 2.12. *Let Ω be a set of proved schedulable tasks. Whatever precedence constraints between Ω's tasks are, it exists, at least, one scheduling which satisfies these precedence constraints.*

Proof. Once a set of n tasks is proved schedulable, these tasks can be scheduled in $n!$ different ways or orders. From these $n!$ orders, at least, one order satisfies the precedence constraints (we remind that tasks are not allowed to be preempted). □

2.1.2. Proposed Approaches

As a result of the previous study, we are able to yield an algorithm allowing the assignment of tasks to processors while satisfying precedence and periodicity constraints. Since the corollary condition is monoprocessor and it is applied for the assignment of each task (the test may be done several time until finding the right processor), the execution time of the assignment algorithm can have be long. Hence, we propose three assignment algorithms as follows:

(1) greedy algorithm: it starts by sorting tasks following a mixed sort which takes into account both the increasing order and a priority level [14]. Then tasks are assigned without any backtracking;

(2) local search algorithm: it uses the condition of the corollary. In order to have an assignment more efficient than the one of the greedy heuristic, we introduce a backtracking process. It is used once a task cannot be assigned to any processor, however, by taking off some assigned tasks from their processors until this task could be assigned and taken off tasks will be assigned to another processors. This backtracking does not change all the assignment but only a part that should not considerably increase the algorithm execution time;

(3) exact algorithm (optimal): it uses the condition of the corollary. This algorithm takes advantage of the Branch & Cut exact method [14].

2.2. Unrolling

The unrolling algorithm consists in repeating each task of the graph (hp/T) times, where the period of this task is T and hp is the hyper-period (Least Common Multiple of all period tasks) [15].

When two tasks are dependent and have not the same period, there are two possibilities. If the period of the consumer task is equal to n times the period of the producer task, then the producer task must be executed n times compared to the consumer task, and the consumer task cannot start its execution until it has received all data from the n executions of the producer task. Notice that the produced data differ from one execution of the producer task to another execution; therefore, data are not duplicated. Reciprocally, if the period of the producer task, is equal to n times the period of the consumer task then the consumer task must be executed n times compared to the producer task. The unrolling algorithm exploits this data transfer mechanism.

2.3. Scheduling

This algorithm distributes and schedules each task of the unrolled graph onto the processor where it has been assigned by the assignment algorithm. In case where the task was assigned to several processors; the algorithm distributes it to the processor which minimizes the makespan. The minimization of the makespan is based on a cost function which takes into account the WCETs of tasks and the communication costs due to dependant tasks scheduled on different processors.

Once a task is scheduled all its instances take start times computed in function of the task period and the number of the instance. In addition, to be scheduled, the first instance of each task must satisfy the condition of the next theorem.

Theorem 2.13. *Let* $\{(t_i : C(t_i), T(t_i), S(t_i)), i = 1 \cdots n\}$ *be n tasks already scheduled on a processor. The task* $\{(t_j : C(t_j), T(t_j)\}$ *is schedulable at the date* $S(t_j)$ *on this processor if and only if*

$$\forall i, \quad S(t_i) \le \left(S(t_j) - S(t_i)\right) \bmod g_i \le \left(g_i - C(t_j)\right) \tag{2.17}$$

such that $g_i = GCD(T(t_i), T(t_j))$.

Proof. In order to prove Theorem 2.13, it suffices to prove that two tasks $\{(t_a : C(t_a), T(t_a), S(t_a))\}$ and $\{(t_b : C(t_b), T(t_b), S(t_b))\}$ (and $g = \mathrm{GCD}(T(t_a), T(t_b))$) are schedulable if and only if

$$C(t_a) \le (S(t_b) - S(t_a)) \bmod g \le \left(g - C(t_b)\right). \tag{2.18}$$

Without any loss of generality, we assume that $S(t_a) = 0$. We start by showing the sufficiency of the Condition 5. Let us consider time intervals $I = \{\forall l \in \mathbb{N}, [0 + lg, g - 1 + lg]$. The first $C(t_a)$ time units of each of these intervals can be allocated for executions of t_a once every $T(t_a)/g$ intervals, and the remaining $g - C(t_a)$ time units only or partly for executions of t_b once every $T(t_b)/g$ intervals. If (2.18) is verified, then the allocated time units suffice to execute t_a and t_b.

In order to prove the necessity of (2.18), let us consider again time intervals $\{\forall l \in \mathbb{N}, [0+lg, g-1+lg]$. If (2.18) is not verified, then the execution of t_b overlaps the first $C(t_a)$ time units once every $T(t_b)/g$ time intervals. We also note that the $C(t_a)$ time units of the intervals are used to execute t_a once every $T(t_a)/g$ time intervals. As $\gcd(T(t_a)/g, T(t_b)/g) = 1$, there will be an interval from I where t_a and t_b are executed together. Hence, if (2.18) is not verified, the tasks t_a and t_b cannot be scheduled on the same processor. This completes the proof of the theorem. □

A complete example of the scheduling heuristic can be found in [14].

3. Performance Evaluation

In order to get the best approach among the three ones proposed into the scheduling heuristic (greedy heuristic, local search, and exact algorithm), we performed two kinds of tests.

The first one consisted in comparing the scheduling success ratio for the three approaches on different systems. This test has been performed as follows: we compute for each system the value of θ which is the ratio between the number of processors and the number of different and nonmultiple periods. Then, we gather all the systems with the same θ, and we execute for them the three approaches of the proposed heuristic. Finally, we compute the scheduling success ratio for the systems of each θ by using the results of the previous step. This method allows us to underline the impact of the architecture in terms of number of processors, and of the number of periods of tasks. The diagram of Figure 2 depicts the evolution of the success ratio according to the variation of θ. Excluding the exact algorithm which finds always the solution, we notice that the local search heuristic displays an interesting results. In addition, the greedy heuristic is efficient only when $\theta \geq 1$ which means that its use is relative to the targeted systems.

In the second test, the speed of the three approaches of the proposed heuristic isevaluated by varying the size of the systems (both number graph tasks and number of processors). The diagram of Figure 3 shows that the exact algorithm explodes very quickly whereas the local search heuristic keeps a reasonable execution time. In addition the greedy heuristic, as expected, stands for the fastest one. Notice that the algorithm execution time follows a logarithmic scale.

To perform these tests, we generated automatically tasks graphs taking into account the periodic issues with dependent and nondependent tasks. The content itself of the task has no impact on the scheduling, only its WCET and its period are relevant. Also, we generated systems such that the number of different periods is not large relatively to the number of tasks; however, we generated systems with all the possible cases (multiple and not multiple period) in order to obtain more realistic results. In addition, the architecture graphs were generated according to a star topology meaning that any two processors can communicate through the medium without using intermediate processors (no routing).

4. Applications

The first example is a simple version of a “Visual control of autonomous vehicles for platooning”. This application is developed by several teams at INRIA. It consists in replacing the joystick which is used to drive manually a CyCab, by an automatic controller based on a video camera which gathers the necessary information to identify the vehicle ahead and to guarantee a minimum distance between the two vehicles. In order to simplify the original version, we use two major tasks, the first one includes the camera and image processing and the second one includes the distance controller and the tasks performing moving forward, steering, and braking (Figure 4 shows the window containing the algorithm graph). The architecture is also a simple version of the real one with only two processors (Figure 5 shows the window containing the architecture graph). The separate execution of these two tasks showed that the first task produces a data every 1 second, whereas the second task consumes the data every 10 milliseconds. It means that these two dependent tasks are periodic, thus, nonperiodic real-time scheduling algorithms are unable to perform a scheduling which satisfies the periods and ensures the right data transfer between them. A

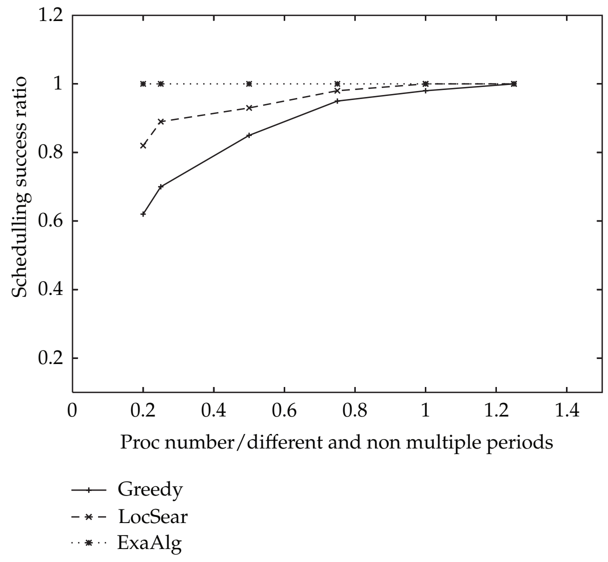


Figure 2: Scheduling success ratio comparison.

shared memory could be an alternative but data sizes and memory access costs represent a significant inconvenient. AAA/SynDEx distributes and schedules onto the architecture these two periodic tasks which implies that the second task is repeated ten times for each execution of the first task. Thereby, the data produced by the first task is diffused to the ten repetitions of the second task (see Figure 6). The result is standing for a timing window corresponding to the predicted real-time behavior of the algorithm running on the architecture following the proposed distribution and scheduling algorithm. It includes one column for each processor and communication medium, describing the distribution (spatial allocation) and the scheduling (temporal allocation) of operations on processors, and of interprocessor data transfers on communication media. Here time flows from top to bottom, the height of each box is proportional to the execution duration of the corresponding operation (periods and execution durations are given or measured by the user).

The second example shows another aspect of AAA/SynDEx multiperiodic utilization. It consists in the "Engine Cooling System", and as for the first example, this application is also a simple version of the original one. This application is composed of two sensors, the first one is a temperature sensor and the second one gives parameters representing the state of the engine. These two sensors are connected to the main task which is the engine control unit, supposed to perform the cooling of the engine. The cooling reads the temperatures sent by the first sensor and combine them with the information representing the operating state of the engine (Figure 7 shows the windows containing the algorithm graph). Thereby, it can predict the change of the temperature and achieve the temperature control (by operating the cooling fan, e.g.). In order to perform more accurate predictions, the main task needs several temperatures which must be taken at equal time intervals and needs also several state information. This is why the temperature sensor is executed every 10 milliseconds (period = 10), the second sensor is executed every 15 milliseconds (Period = 15), and the main task every 30 milliseconds (period = 30). The architecture is composed of two processors similarly to the architecture of the previous example.

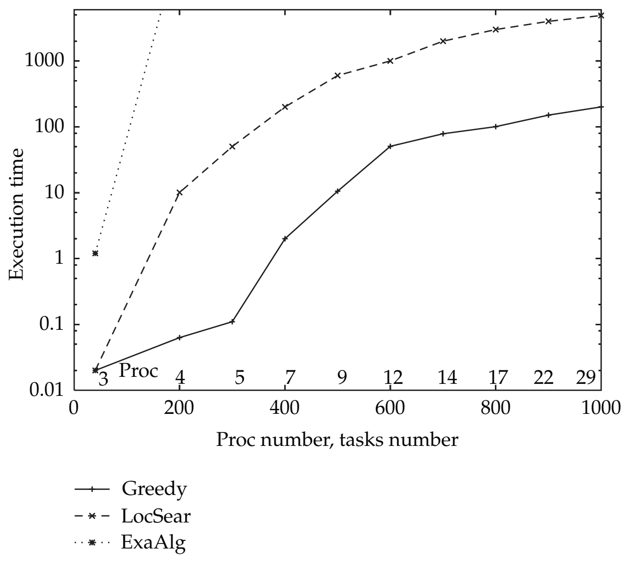


Figure 3: Execution time comparison.

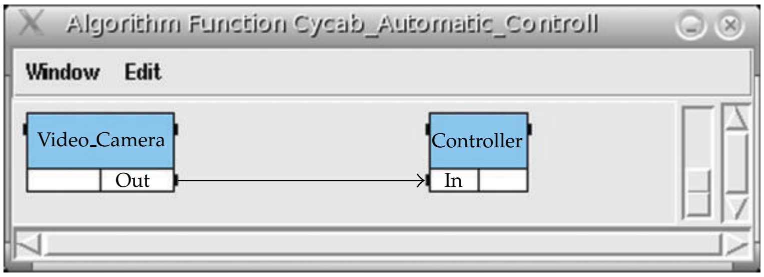


Figure 4: Platooning application (tasks graph).

Figure 8 shows the result of the adequation applied to the algorithm and the architecture.

On Figure 8 we can observe that the temperature sensor is executed three times. These three temperatures and two data producded by the two executions of the state sensor are sent to the main task. This latter, by receiving all these data, predicts the change of the temperature and achieves the temperature control.

Finally, the user can launch the generation of an optimized distributed macro-executive, which produces as much files as there are of processors. This macroexecutive, independent of the processors and of the media, is directly derived from the result of the distribution and the scheduling. It will be macroprocessed with GM4 using executive-kernels which are dependent of the processors and the media, in order to produce source codes. These codes will be compiled with the corresponding tools (several compilers or assemblers corresponding to the different types of processors and/or source languages), then they will be linked and loaded on the actual processors where the applications will ultimately run in realtime.

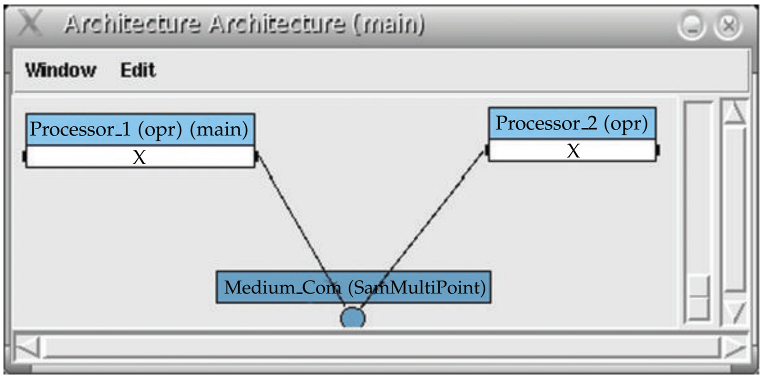


Figure 5: Platooning application (architecture graph).

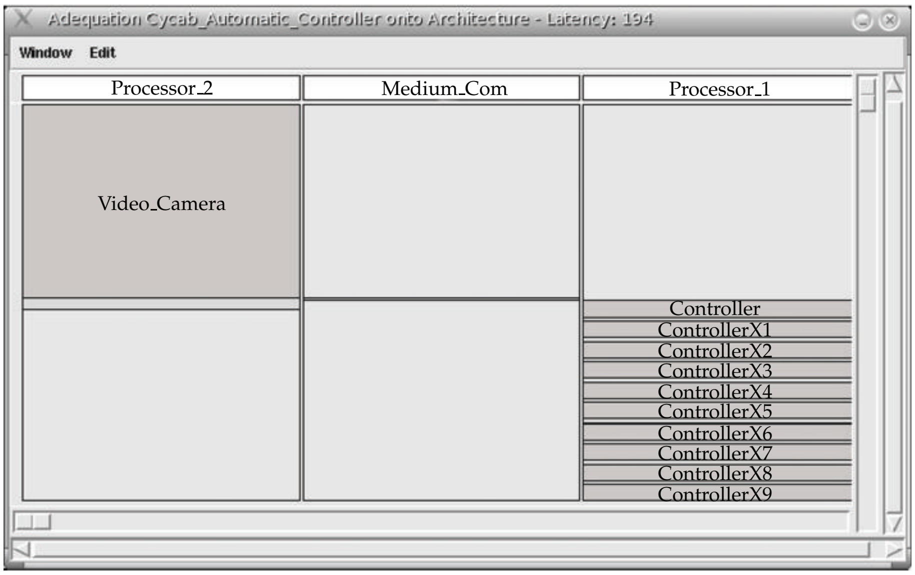


Figure 6: Platooning application (distribution and scheduling).

5. Load and Memory Balancing

We proposed to improve the new distributed real-time scheduling heuristic in two main directions:

(i) minimizing the makespan of distributed applications by equalizing the workloads of processors (even thought a first attempt was carried out in the proposed heuristic),

(ii) efficient utilization of the memory resource.

Towards these purposes we proposed, a load and memory balancing heuristic for homogeneous distributed real-time embedded applications with dependence and strict periodicity constraints. It deals with applications involving N tasks and M processors. Each task a has an execution time E_a, a start time S_a computed by the distributed scheduling

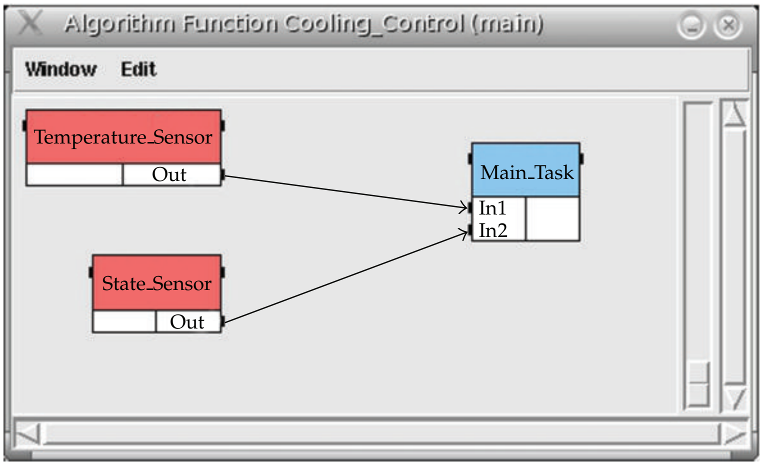


Figure 7: Cooling control (tasks graph).

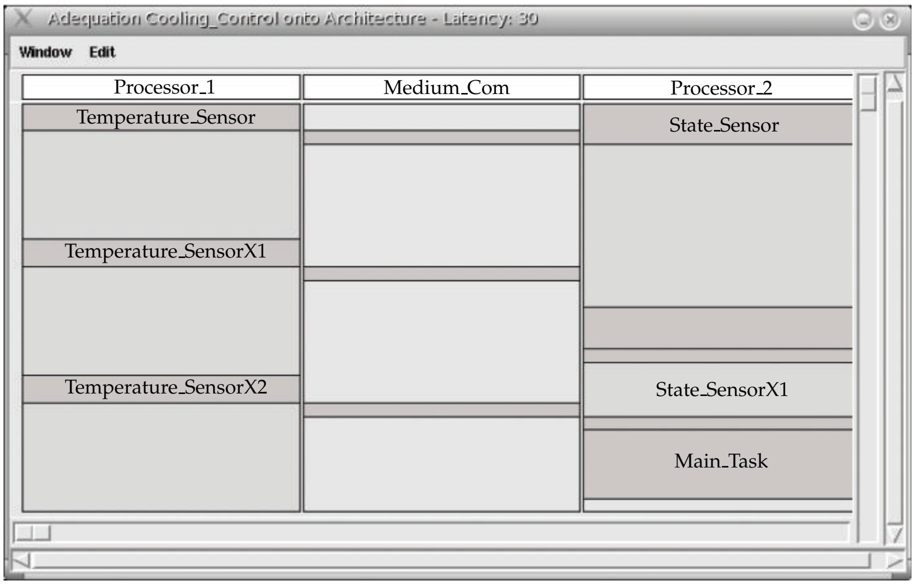


Figure 8: Cooling control (distribution and scheduling).

heuristic, and a required memory amount m_a. The required memory amount may be different for every task. It represents the memory space necessary to store the data managed by the task, that is, all the variables necessary for the task according to their types.

For each processor, the proposed heuristic starts by building blocks from tasks distributed and scheduled onto this processor. Then, each block A is processed according to the increasing order of their start times. This process consists in computing the cost function λ (defined in the next paragraph) for the processors whose end time of the last block scheduled on these processors is less than or equal to the start time of the block A, and in seeking the processor which maximizes λ. Moreover, a block is moved to that processor if the periodicity of the other blocs on this processor is verified, otherwise that processor is no longer considered, and the heuristic seeks again another processor which maximizes λ. If the moved block belongs to the first category and $\lambda > 0$, then this block will decrease its start time. In order to keep its strict periodicity constraint satisfied, the heuristic looks through the remaining blocks and updates the start times of the blocks containing tasks whose instances are in the moved block. This heuristic is applied to the output of the scheduling heuristic already presented.

The heuristic is based then on a *Cost Function* $\lambda_{P_i \to P_j}(A)$ which is computed for a block A initially scheduled on processor P_i and a processor P_j

$$\lambda_{P_i \to P_j}(A) = \begin{cases} \mathbb{G}_{P_i \to P_j}(A) & \text{if no block has been moved to } P_j, \\ \dfrac{\mathbb{G}_{P_i \to P_j}(A) + 1}{\sum_{i=1}^{i=k} m_{B_i}} & \text{otherwise.} \end{cases} \tag{5.1}$$

It combines $\mathbb{G}_{P_i \to P_j}(A)$ and the sum of required memory amounts by the k blocks $B_1, \ldots, B_k$ already moved to this processor P_j. Notice that $\mathbb{G}$ is the gain in terms of time due to the move of this block.

A detailed example and a complexity and a theoretical performance studies can be found in [16].

6. Conclusion

We presented a new feature for AAA/SynDEx tool which allows the scheduling of multiperiodic dependent nonpreemptive tasks onto a multiprocessor. The new feature of AAA/SynDEx provides a unique scheduling algorithm since it is based on a schedulability analysis which allows for distributing and scheduling the tasks while satisfying their dependences and their strict periodicity. The analysis proposed here is composed in several stages to reach the main result which is a necessary and sufficient condition we obtain at the end through a corollary. Such schedulability analysis can be used in suboptimal heuristics to find an assignment of the tasks for each processor when partitioned multiprocessor scheduling is intended.

When dependent tasks with two different periods are distributed and scheduled onto two different processors the proposed heuristic handles correctly the interprocessor communications.

Since memory is limited in embedded systems, it must be efficiently used and, also, the total execution time must be minimized since the systems we are dealing with include feedback control loops. Thus, we improved the presented heuristic in order to perform load Balancing and efficient memory usage of homogeneous distributed real-time embedded systems. This is achieved by grouping the tasks into blocks, and moving them to the processor such that the block start time decreases, and this processor has enough memory capacity to execute the tasks of the block.

The study of this paper stands for a first step from a more global work which targets resolving all kinds of periodicity constraints.

Acknowledgment

This paper is a part of my thesis works which has been supported by French National institute For Research in Computer Science and Control (INRIA) under the supervision of Yves Sorel.

References

[1] T. Grandpierre, C. Lavarenne, and Y. Sorel, "Optimized rapid prototyping for real-time embedded heterogeneous multiprocessors," in *Proceedings of the 7th International Workshop on Hardware/Software Co-Design (CODES '99)*, pp. 74–78, Rome, Italy, May 1999.

[2] Y. Sorel, "Syndex: system-level cad software for optimizing distributed real-time embedded systems," *Journal ERCIM News*, vol. 59, pp. 68–69, 2004.

[3] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM*, 1973.

[4] M. Bekooij, O. Moreira, P. Poplavko, B. Mesman, M. Pastrnak, and J. van Meerbergen, "Predictable embedded multiprocessor system design," in *Software and Compilers for Embedded Systems*, vol. 3199, pp. 77–91, 2004.

[5] L. Cucu and Y. Sorel, "Non-preemptive multiprocessor scheduling for strict periodic systems with precedence constraints," in *Proceedings of the 23rd Annual Workshop of the UK Planning and Scheduling Special Interest Group (PLANSIG '04)*, Cork, Ireland, December 2004.

[6] T. Grandpierre and Y. Sorel, "From algorithm and architecture specification to automatic generation of distributed real-time executives: a seamless flow of graphs transformations," in *Proceedings of the 1st ACM and IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE '03)*, Mont Saint-Michel, France, June 2003.

[7] K. Jeffay, D. F. Stanat, and C. U. Martel, "On non-preemptive scheduling of periodic and sporadic tasks," in *Proceedings of the 12th IEEE Symposium on Real-Time Systems Symposium*, pp. 129–139, December 1991.

[8] F. Balarin, L. Lavagno, P. Murthy, and A. Sangiovanni-vincentelli, "Scheduling for embedded real-time systems," *IEEE Design and Test of Computers*, vol. 15, no. 1, pp. 71–82, 1998.

[9] J. W. S. W. Liu, *Real-Time Systems*, Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[10] S. K. Dhall and C. L. Liu, "On a real-time scheduling problem," *Operations Research*, vol. 26, no. 1, pp. 127–140, 1978.

[11] T. P. Baker, "Multiprocessor edf and deadline monotonic schedulability analysis," in *Proceedings of the 24th IEEE International Real-Time Systems Symposium (RTSS '03)*, p. 120, IEEE Computer Society, Washington, DC, USA, 2003.

[12] P. Pop, P. Eles, Z. Peng, and T. Pop, "Analysis and optimization of distributed real-time embedded systems," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 593–625, 2006.

[13] S. K. Baruah, "The non-preemptive scheduling of periodic tasks upon multiprocessors," *Real-Time Systems*, vol. 32, no. 1-2, pp. 9–20, 2006.

[14] O. Kermia and Y. Sorel, "A rapid heuristic for scheduling non-preemptive dependent periodic tasks onto multiprocessor," in *Proceedings of the 20th ISCA international Conference on Parallel and Distributed Computing Systems (PDCS '07)*, Las Vegas, Nev, USA, September 2007.

[15] K. Ramamritham, "Allocation and scheduling of precedence-related periodic tasks," *IEEE Transactions on Parallel and Distributed Systems*, vol. 6, no. 4, pp. 412–420, 1995.

[16] O. Kermia and Y. Sorel, "Load balancing and efficient memory usage for homogeneous distributed real-time embedded systems," in *Proceedings of the 4th International Workshop on Scheduling and Resource Management for Parallel and Distributed Systems (SRMPDS '08)*, Portland, Ore, USA, September 2008.