Propagation mechanism for a non-preemptive strictly periodic scheduling problem

Clément Pira, Christian Artigues

Propagation mechanism for a non-preemptive strictly periodic scheduling problem

Clément Pira, Christian Artigues

CNRS, LAAS, 7 avenue du colonel Roche, F-31400 Toulouse, France
Univ de Toulouse, LAAS, F-31400 Toulouse, France
{pira,artigues}@laas.fr

Keywords: Periodic scheduling, equilibrium, propagation mechanism

Abstract. We study a non-preemptive strictly periodic scheduling problem. This problem, introduced in [6, 4], arises for example in the avionic field where a set of N periodic tasks (measure of a sensor, etc.) has to be scheduled on P processors distributed on the plane. In the related field of cyclic scheduling [5], some notable success [3] has been recently achieved through the use of specific constraint propagation. In this article, we consider a heuristic, first proposed in [1, 2], which is based on the notion of equilibrium. Following a game theory analogy, each task tries successively to optimize its own schedule and therefore to produce the best response, given the other schedules. We present a propagation mechanism for non-overlapping constraints which significantly improves this heuristic.

1 Periodic scheduling problem and its MILP formulation

We consider a non-preemptive strictly periodic scheduling problem introduced in [6, 4, 1, 2]. Such a problem arises in the avionic field, where a set of N periodic tasks (measure of a sensor, etc.) has to be scheduled on P processors distributed on the plane. In this problem, each task i has a fixed period T_i and a processing time p_i. They are subject to non-overlapping constraints : no two tasks assigned to the same processor can overlap during any time period (see Figure 1). A solution is given by an assignment of the tasks to the processors and, for each task by the start time t_i (offset) of one of its occurrences. In the following, unlike the papers mentioned above, we adopt a more general model in which processing times p_i are generalized by positive latency delays $l_{i,j} \geq 0$ (or time lags). The former case is the particular case where $l_{i,j} = p_i$ for all other tasks j. The proposed heuristic adapts easily to this generalization.

In this paper, we only consider the case where the offsets t_i are integers. This hypothesis is important to prove convergence of the method described in section 2.1. Since the problem is periodic, all the periods T_i also need to be integers[1].

• This work was funded by the French Midi-Pyrenee region (allocation de recherche post-doctorant n°11050523) and the LAAS-CNRS OSEC project (Scheduling in Critical Embedded Systems).

[1] Indeed, if t_i is an integer solution then $t_i + T_i$ should also be an integer solution, hence their difference too, *i.e.* T_i.

However the latency delays need not be integer *a priori*, especially in the case of the optimization problem presented in section 1.1.

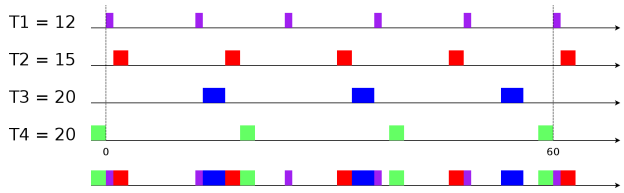

Fig. 1. $N = 4$ non-overlapping periodic tasks on $P = 1$ processor

1.1 Definition of the uniprocessor problem

Non-overlapping constraints In the formulation of a monoprocessor periodic scheduling problem, a latency delay $l_{i,j} \geq 0$ has to be respected whenever an occurrence of a task j starts after an occurrence of a task i. Said differently, the smallest positive difference between two such occurrences has to be greater than $l_{i,j}$. Using Bézout identity, the set $(t_j + T_j\mathbb{Z}) - (t_i + T_i\mathbb{Z})$ of all the possible differences is equal to $(t_j - t_i) + g_{i,j}\mathbb{Z}$ where $g_{i,j} = \gcd(T_i, T_j)$. The smallest positive representative of this set is $(t_j - t_i) \mod g_{i,j}$. This is the only representative of $(t_j - t_i) + g_{i,j}\mathbb{Z}$ which belongs to $[0, g_{i,j} - 1]$ (in particular, we consider a classic positive modulo, and not a signed modulo). Therefore, we simply consider the following constraint :

$$(t_j - t_i) \bmod g_{i,j} \geq l_{i,j}, \qquad \forall (i,j),\ i \neq j \tag{1}$$

Classically, processing times are strictly positive, however our model allows zero delays. Since equation (1) is trivially satisfied when $l_{i,j} = 0$, we introduce an additional graph $\mathcal{G}$ containing the arcs (i,j) for which $l_{i,j} > 0$, and we consider equation (1) only for those couples. As we will see, the constraints associated with (i,j) and (j,i) work naturally together and some complications are introduced when only one of the couples has a strictly positive delay. To avoid special cases, we will suppose in the following that the graph $\mathcal{G}$ is symetric[2] : two delays $l_{i,j}$ and $l_{j,i}$ are either both zero, or both strictly positive. With this hypothesis, a consequence of proposition 1 (see Appendix) is that equation (1) can be replaced by the following interval constraint for each couple $(i,j) \in \mathcal{G}$ with $i < j$:

$$l_{i,j} \leq (t_j - t_i) \bmod g_{i,j} \leq g_{i,j} - l_{j,i}, \qquad \forall (i,j) \in \mathcal{G},\ i < j \tag{2}$$

Note that except for the modulo, these constraints have a clear analogy with classical precedence constraints of the form $t_j - t_i \geq l_{i,j}$. However, they are much harder to handle since they induce an exclusion between tasks[3].

[2] Note that this hypothesis can always be enforced. If $l_{i,j} > 0$, then the tasks i and j are constrained not to start at the same time. Hence the same is true for the tasks j and i. Therefore, $l_{j,i}$ can also be supposed strictly positive.

[3] In particular, a classic disjunctive constraint $t_j - t_i \geq l_{i,j} \vee t_i - t_j \geq l_{i,j}$ can be represented by the two constraints $(t_j - t_i) \bmod T \geq l_{i,j}$ and $(t_i - t_j) \bmod T \geq l_{j,i}$ for a large enough period T.

Objective to maximize In an uncertain context, a task may last longer than expected, due to failure of the processor. This increase in the duration is naturally proportional to the original processing time. To withstand the worst possible slowdown, it is natural to make all the delays $l_{i,j}$ proportional to a common factor $\alpha \geq 0$ that we try to optimize (see [1, 2]).

$$\begin{aligned} \max \quad & \alpha & & (3) \\ s.t. \quad & l_{i,j}\alpha \leq (t_j - t_i) \bmod g_{i,j} \leq g_{i,j} - l_{j,i}\alpha \qquad \forall (i,j) \in \mathcal{G},\ i < j & (4) \\ & t_i \in \mathbb{Z} \qquad \forall i & (5) \\ & \alpha \geq 0 & (6) \end{aligned}$$

The addition of this objective is also a good way to obtain feasible solutions. In the context of precedence constraints, a way to implement the Bellman-Ford algorithm is to cyclically choose a task and to push it forward, up to the first feasible offset. In section 2.2 we present a method to find the next feasible offset in the case of non-overlapping constraints. However there isn't necessarily a solution and a similar propagation mechanism which would cyclically move one task at a time to the first feasible location would have only a small chance to converge towards a feasible solution. However, with the optimization problem, there is always at least a solution having a zero α-value. We can then try to inflate α, while moving a task, what we call the best-response method : instead of choosing the first feasible location, we search for the best location with respect to this additional objective α. Whenever a solution with $\alpha \geq 1$ is found, we have a solution for the feasibility problem. This is essentially the principle of the heuristic presented in section 2.

Structure of the solution set An optimal solution is completely defined by the offsets, since given the offsets (t_i) we can always compute the best compatible α-value :

$$\alpha = \min_{(i,j)\in\mathcal{G}} \frac{(t_j - t_i) \bmod g_{i,j}}{l_{i,j}} \tag{7}$$

If we try to draw the function $\alpha = (t_i - t_j) \bmod g_{i,j}/l_{j,i}$ representing the optimal value of α respecting one constraint $(t_i - t_j) \bmod g_{i,j} \geq l_{j,i}\alpha$ when t_i takes rational values, we obtain a curve which is piecewise increasing and discontinuous since it becomes zero on $t_j + g_{i,j}\mathbb{Z}$. In the same way, if we draw $\alpha = (t_j - t_i) \bmod g_{i,j}/l_{i,j}$, we obtain a curve which is piecewise decreasing and becomes zero at the same points. It is therefore more natural to consider the two constraints jointly, *i.e.* (4), which gives a continuous curve (see Figure 2) [4].

Since $\mathcal{G}$ is symetric, it can be seen as an undirected graph. We will write $\mathcal{G}(i)$ for the set of neighbors of i, *i.e.* the set of tasks with which i is linked through non-overlapping constraints. With this notation, and given some fixed

[4] Note that if $\mathcal{G}$ was not symetric, we would need to consider some degenerate cases were one of the slope (increasing or decreasing) would be infinite (since $l_{i,j} = 0$ or $l_{j,i} = 0$).

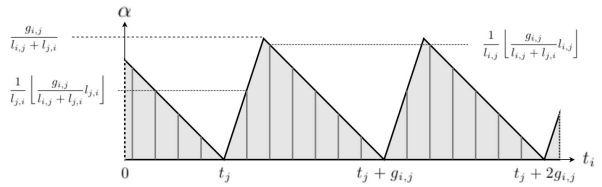

Fig. 2. Possible values for (t_i, α) when constrained by a single task j

offsets $(t_j)_{j\in\mathcal{G}(i)}$, the set of solutions (t_i, α) is the intersection of the curves decribed above for each $j \in \mathcal{G}(i)$ (see Figure 3). It is composed of several adjacent polyhedra. We can give an upper bound on the number n_{poly} of such polyhedra. A polyhedron starts and ends at zero points. For a given constraint j, there is $T_i/g_{i,j}$ zero points in $[0, T_i - 1]$, hence n_{poly} is bounded by $T_i \sum_{j\neq i} \frac{1}{g_{i,j}}$. This upper bound can be reached when the offsets are fractional, since in this case, we can always choose the offsets t_j such that the sets of zero points $(t_j + g_{i,j}\mathbb{Z})_{j\in\mathcal{G}(i)}$ are all disjoint. In the case of integer offsets, there is obvioulsy at most T_i zero points in the interval $[0, T_i - 1]$ and therefore, at most T_i polyhedra.

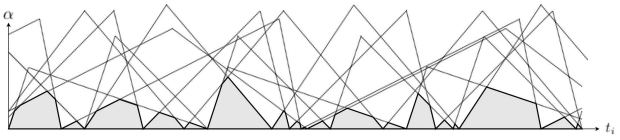

Fig. 3. Possible values for (t_i, α) constrained by all tasks $j \in \mathcal{G}(i)$

We obtain the shape of a two dimensional slice of the solution set, for a given task i, and some fixed $(t_j)_{j\in\mathcal{G}(i)}$. We will see how these two-dimensional slices help to understand the concept of equilibrium and the heuristic : when solving the best-response method, we try to find the new location t_i which maximize the α-value. Following the same idea, the complete full dimensional solution set is composed of several adjacent $N + 1$-dimensional polyhedra.

1.2 MILP formulation of the multiprocessor problem

MILP formulation In this section, we present a MILP formulation of the problem which will be used to compare the performance of our heuristic. As explained above, the modulo $(t_j - t_i) \bmod g_{i,j}$ is the only element $t_j - t_i + g_{i,j}q_{i,j}$ in the interval $[0, g_{i,j})$. Here, $q_{i,j}$ is an additional 'quotient variable' which can always be computed by :

$$q_{i,j} = \lceil (t_i - t_j)/g_{i,j} \rceil \tag{8}$$

Hence for a given couple $(i, j) \in \mathcal{G}$, with $i < j$, the non-overlapping constraint (4) can be rewritten as $l_{i,j}\alpha \leq t_j - t_i + g_{i,j}q_{i,j} \leq g_{i,j} - l_{j,i}\alpha$. In the multiprocessor case, the scheduling problem is coupled with an assignment problem. Thus, we introduce binary variables $a_{i,k}$ which indicate if a task i is assigned to a processor

k, and variables $x_{i,j}$ which indicate if i and j are on different processors. These variables must satisfy (10) and (11). Since the non-overlapping constraints have to be satisfied whenever two tasks are on the same processor, we replace the term $l_{i,j}\alpha$ by $l_{i,j}\alpha - l_{i,j}\alpha_{\max}x_{i,j}$. When i and j are on the same processor ($x_{i,j} = 0$), we find back the original term $l_{i,j}\alpha$, otherwise we obtain a negative term $l_{i,j}\alpha - l_{i,j}\alpha_{\max}$ which makes the constraint trivially satisfiable. This yields constraints (12) and (13).

$$\begin{aligned}
\max \quad & \alpha && && (9)\\
& \sum_k a_{i,k} = 1 && \forall i && (10)\\
& x_{i,j} \leq 2 - a_{i,k} - a_{j,k} && \forall k, \quad \forall (i,j) \in \mathcal{G}, i < j && (11)\\
& t_j - t_i + g_{i,j}q_{i,j} \geq l_{i,j}\alpha - \alpha_{\max}l_{i,j}x_{i,j} && \forall (i,j) \in \mathcal{G}, i < j && (12)\\
& t_j - t_i + g_{i,j}q_{i,j} \leq g_{i,j} - l_{j,i}\alpha + \alpha_{\max}l_{j,i}x_{i,j} && \forall (i,j) \in \mathcal{G}, i < j && (13)\\
& t_i \in \mathbb{Z} && \forall i && (14)\\
& q_{i,j} \in \mathbb{Z} && \forall (i,j) \in \mathcal{G}, i < j && (15)\\
& x_{i,j} \in [0,1] && \forall (i,j) \in \mathcal{G}, i < j && (16)\\
& a_{i,k} \in \{0,1\} && \forall k, \forall i && (17)
\end{aligned}$$

Bounds on the offsets and quotients We can see that the previous system is invariant under translation $t_i \mapsto t_i + n_iT_i$: if (t_i) is feasible and if (n_i) is a vector of integers, then the solution (t'_i) defined by $t'_i = t_i + n_iT_i$ is also feasible. Since $t_i \bmod T_i = t_i + n_iT_i$ for some $n_i \in \mathbb{Z}$, we deduce that $(t_i \bmod T_i)$ is also a solution. We can therefore impose additionnal bounds $t_i \in [0, T_i - 1]$. More generally, this remains true if we replace T_i with an updated period :

$$T_i^* = \operatorname{lcm}((g_{i,j})_{j\in\mathcal{G}(i)}) = \gcd(T_i, \operatorname{lcm}((T_j)_{j\in\mathcal{G}(i)})) \tag{18}$$

The updated period T_i^* always divide T_i (which we denote $T_i^*|T_i$) and we can have a strictly smaller value for example if a prime factor p only occurs in one period T_i (or more generally if a factor p occurs with multiplicity m in T_i but with strictly smaller multiplicities in any other periods of a task connected to i). Indeed, even if the (T_i) are the initial parameters, only their GCD appears in the constraints. Therefore, if a prime factor occurs in only one period, then it completely disappear in the $g_{i,j}$. By replacing the initial periods (T_i) by the updated ones (T_i^*), we simply remove some irrelevant factors. In the following, we will suppose that the periods have been updated to have no proper factors, *i.e.* $T_i = T_i^*$. Computing the image of the interval $[0, T_i - 1]$ by expression (8) immediately gives associated bounds on the quotients :

$$1 - \frac{T_j}{g_{i,j}} \leq q_{i,j} \leq \frac{T_i}{g_{i,j}} \tag{19}$$

In particular, if $T_j|T_i$ then $q_{i,j} \geq 0$ (in fact $q_{i,j} \in [0, T_i/T_j]$), which shows that in the harmonic case, we can impose positive variables. Conversely if $T_i|T_j$ then $q_{i,j} \leq 1$. Finally, if $T_i = T_j$, then $q_{i,j} \in \{0, 1\}$ which shows that the monoperiodic case correspond to the case of binary variables.

Upper bound on the α-value Let i and j be two tasks on the same processor. Then, constraint (4) gives the following upper bound : $\alpha \leq g_{i,j}/(l_{i,j}+l_{j,i})$. Taking the integrality assumption into account, we get an even tighter upper bound :

$$\alpha \leq \alpha_{\max}^{i,j} = \max\left(\frac{1}{l_{i,j}}\left\lfloor\frac{g_{i,j}}{l_{i,j}+l_{j,i}}l_{i,j}\right\rfloor \ , \ \frac{1}{l_{j,i}}\left\lfloor\frac{g_{i,j}}{l_{i,j}+l_{j,i}}l_{j,i}\right\rfloor\right) \tag{20}$$

These bounds are illustrated on Figure 2. In the monoprocessor case, we deduce that $\alpha_{\max} = \min_{i,j} \alpha_{\max}^{i,j}$ is an upper bound on the value of α. In the multiprocessor case, we have at least the trivial upper bound $\alpha_{\max} = \max_{i,j} \alpha_{\max}^{i,j}$. However, since $\alpha_{\max}$ is used as a 'big-M' in the MILP formulation, we would like the lowest possible value in order to improve the efficiency of the model. For this, we solve a preliminary model, dealing only with the assignment (hence without variables (t_i) and $(q_{i,j})$), in which non-overlapping constraints (12-13) are replaced by the following weaker constraint :

$$\alpha \leq \alpha_{\max}^{i,j} + (\alpha_{\max} - \alpha_{\max}^{i,j})x_{i,j}, \qquad \forall (i,j) \in \mathcal{G},\ i < j \tag{21}$$

Intuitively, this constraint indicates that α should be less than $\alpha_{\max}^{i,j}$ if i and j are assigned to the same processor ($x_{i,j} = 0$), otherwise it is bounded by $\alpha_{\max}$ which is the trivial upper bound. Solving this model is much faster than for the original one (less that $1s$ for instances with 4 processors and 20 tasks). It gives us a new value $\alpha_{\max}$ which can be used in the original model.

2 An equilibrium-based heuristic

2.1 Heuristic and the mutiprocessor best response method

The multiprocessor best response method The main component of the algorithm is called the best response procedure. It takes its name from a game theory analogy. Each task is seen as an agent which tries to optimize its own assignment and offset, while the other assignments and offsets are fixed. Instead of using a binary vector $(a_{i,k})$ as in the MILP formulation, we represent an assignment more compactly by a variable $a_i \in [1, P]$. For each agent i, we want to define a method MultiProcBestResponse$_i$ which returns the best assignment and offset for the task i, given the current assignment and offsets (t_j, a_j) of all the tasks. In order to choose the assignment, an agent simply tries every possible processor. On a given processor p, it tries to find the best offset, taking into account the non-overlapping contraints with tasks currently one this processor.

More formally, the $\textsc{BestOffset}_i^p$ procedure consists in solving the following program :

$$
\begin{aligned}
(BO_i^p)\ \max \quad & \alpha && && (22)\\
s.t. \quad & l_{j,i}\alpha \leq (t_i - t_j) \bmod g_{i,j} \leq g_{i,j} - l_{i,j}\alpha && \forall_{\substack{j\in\mathcal{G}(i)\\ a_j=p}} && (23)\\
& t_i \in \mathbb{Z} && && (24)\\
& \alpha \geq 0 && && (25)
\end{aligned}
$$

Note that there are only two variables, t_i and α, since the other offsets and assignments $(t_j, a_j)_{j\in\mathcal{G}(i)}$ are parameters. A method to solve this program will be presented in section 2.2. Following the discussion in section 1.2, the previous system is invariant under tranlation by T_i, and even by the possibly smaller value $T_i^p = \text{lcm}((g_{i,j})_{j\in\mathcal{G}(i)|a_j=p})$. Hence, we can impose t_i to belong to $[0, T_i^p - 1]$. In the same way, we can compute an upper bound on this program : $\alpha_{\max}^p = \min_{j\in\mathcal{G}(i)|a_j=p} \alpha_{\max}^{i,j}$. If the current best solution found on previous processors is already better than this upper bound, there is no way to improve the current solution with this processor, hence we can skip it. The multiprocessor best-response procedure is summarized in Algorithm 1.

Algorithm 1 The multiprocessor best-response

```
1: procedure MultiProcBestResponse_i((t_j)_{j∈I}, (a_j)_{j∈I})
2:     ▷ We start with the current processor a_i, which has priority in case of equality
3:     newa_i ← a_i
4:     (newt_i, α) ← BestOffset_i^{a_i}((t_j)_{j∈I}, (a_j)_{j∈I})
5:     for all p ≠ a_i do                                   ▷ We test the other processors
6:         α_max^p ← min_{j∈G(i), a_j=p} α_max^{i,j}  ▷ We compute an upper bound on α when i is on p
7:         if α_max^p > α then                  ▷ An improvement can possibly be found on p
8:             (x, β) ← BestOffset_i^p((t_j)_{j∈I}, (a_j)_{j∈I})
9:             if β > α then                          ▷ An improvement has been found on p
10:                newa_i ← p; newt_i ← x; α ← β;
11:            end if
12:        end if
13:    end for
14:    return (newt_i, newa_i, α)
15: end procedure
```

The concept of equilibrium and principle of the method to find one

Definition 1. *A solution* (t_i, a_i) *is an equilibrium* iff *no task* i *can improve its assignment or offset using procedure* $\textsc{MultiProcBestResponse}_i$.

The heuristic uses a counter N_{stab} to count the number of tasks known to be stable, *i.e.* which cannot be improved. It starts with an initial solution (for example randomly generated) and tries to improves this solution by a succession

of unilateral optimizations. On each round, we choose cyclically a task i and try to optimize its schedule, *i.e.* we apply MultiProcBestReponse$_i$. If no improvement was found, then one more task is stable, otherwise we update the assignment and offset of task i and reinitialize the counter of stable tasks. We continue until N tasks are stable. This is summarized in Algorithm 2.

Remark 1. The main reason for the use of integers is that it allows to guarantee the convergence of the heuristic. The termination proof relies on the fact that there is only a finite number of possible values for α (we refer to [1] for the proof of termination and correction). This is not the case with fractional offsets, for which the heuristic is unlikely to converge in a finite number of steps.

Algorithm 2 The heuristic

```
1: procedure ImproveSolution((t_j)_{j∈I}, (a_j)_{j∈I})
2:     N_stab ← 0                                  ▷ The number of stabilized tasks
3:     i ← 0                                       ▷ The task currently optimized
4:     while N_stab < N do                         ▷ We run until all the tasks are stable
5:         (newt_i, newa_i, α_i) ← MultiProcBestResponse(i, (t_j)_{j∈I})
6:         if newt_i ≠ t_i or newa_i ≠ a_i then
7:             t_i ← newt_i; a_i ← newa_i          ▷ We have a strict improvement for task i
8:             N_stab ← 1                          ▷ We restart counting the stabilized tasks
9:             α ← α_i
10:        else                                    ▷ We do not have a strict improvement
11:            N_stab ← N_stab + 1                 ▷ One more task is stable
12:            α ← min(α_i, α)
13:        end if
14:        i ← (i + 1) mod N                       ▷ We consider the next task
15:    end while
16:    return (α, (t_j)_{j∈I}, (a_i)_{j∈I})
17: end procedure
```

An equilibrium is only an approximate notion of optimum. Hence, in order to find a real optimum, the idea is now to run the previous heuristic several times with different randomly generated initial solutions, and to keep the best result, following a standard multistart scheme.

2.2 The best-offset procedure on a given processor

We now need to implement the BestOffset$_i^p$ method, *i.e.* solve (BO_i^p). Since t_i is integer and can be supposed to belong to $[0, T_i^p - 1]$, we can trivially solve this program by computing the α-value for each of these offsets, using expression (7), and select the best one. This procedure runs in $O(T_i N)$, hence any method should at least be faster. In [1], the authors propose a method consisting in computing the α-value only for a set of precomputed intersection points (in the next section we will see that a fractional optimum is at the intersection of an increasing and a decreasing line). This already improves the trivial procedure. In the following, we present an even more efficient method.

Local (two dimensional) polyhedron We want to compute the local polyhedron which contains a reference offset t_i^*. Locally, the constraint $(t_i - t_j) \bmod g_{i,j} \geq l_{j,i}\alpha$ is linear, of the form $t_i - o_j \geq l_{j,i}\alpha$. Here, o_j is the largest $x \leq t_i^*$ such that $(x - t_j) \bmod g_{i,j} = 0$. In the same way, we can compute the decreasing constraint $o'_j - t_i \geq l_{i,j}\alpha$. In this case o'_j is the smallest $x \geq t_i^*$ such that $(o'_j - x) \bmod g_{i,j} = 0$. By proposition 2 (see Appendix), we have :

$$o_j = t_i^* - (t_i^* - t_j) \bmod g_{i,j} \qquad \text{and} \qquad o'_j = t_i^* + (t_j - t_i^*) \bmod g_{i,j} \tag{26}$$

Therefore, we obtain a local polyhedron (see figure 4). Note that when $(t_i^* - t_j) \bmod g_{i,j} > 0$, we simply have $o'_j = o_j + g_{i,j}$ by proposition 1 (see Appendix). However, when $(t_i^* - t_j) \bmod g_{i,j} = 0$, we have $o_j = o'_j = t_i^*$. In this case, the polyhedron is degenerated since it contains only $\{t_i^*\}$, and the α-value at t_i^* is zero. Instead of computing this polyhedron, we prefer to choose either the polyhedron on the right, or on the left (see figure 5). If we choose the one on the right, this amounts to defining o'_j to be the smallest $x > t_i^*$ which sets the constraint to zero. By proposition 2, we have $o'_j = t_i^* + g_{i,j} - (t_i^* - t_j) \bmod g_{i,j}$. Therefore, this simply amounts to enforcing $o'_j = o_j + g_{i,j}$. Choosing the polyhedron on the left amounts to defining o'_j using (26) and to enforcing $o_j = o'_j - g_{i,j}$.

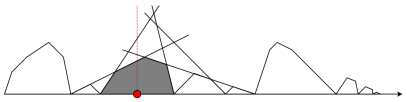

Fig. 4. Selection of the polyhedron containing a reference offset t_i^*

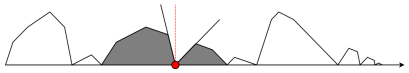

Fig. 5. Two possibilities in the degenerate case $\alpha = 0$

Solving the local best offset problem Once the local polyhedron has been defined, the problem is now to solve the following MILP :

$$\begin{aligned}
(Loc{-}BO_i^p) \qquad \max \quad & \alpha && && (27)\\
s.t. \quad & t_i - l_{j,i}\alpha \geq o_j && \forall j \in \mathcal{G}(i),\ a_j = p && (28)\\
& t_i + l_{i,j}\alpha \leq o'_j && \forall j \in \mathcal{G}(i),\ a_j = p && (29)\\
& t_i \in \mathbb{Z} && && (30)
\end{aligned}$$

We can first search for a fractional solution, and for this we can use any available method of linear programming. However, since the problem is a particular two dimensional program, we can give special implementations of these methods. In the following, we present a simple primal simplex approach, which runs in $O(N^2)$ in the worst case but has a good behaviour in practice. A local polyhedron is delimited by increasing and decreasing lines, and the fractional optimum is at the intersection of two such lines. Hence a natural way to find the optimum is to try all the possible intersections between an increasing line, of the form $x - l_{j,i}\alpha = o_j$, and a decreasing line, of the form $x + l_{i,k}\alpha = o'_k$, and

to select the one with the smallest α-value. The coordinates of these intersection points are given by[5] :

$$x = \frac{l_{j,i}o'_k + l_{i,k}o_j}{l_{j,i} + l_{i,k}} \qquad \text{and} \qquad \alpha = \frac{o'_k - o_j}{l_{j,i} + l_{i,k}} \tag{31}$$

Since there is at most $N-1$ lines of each kinds, the algorithm runs in $O(N^2)$. In practice, a better approach (with the same worst case complexity) is to start with a couple of increasing and decreasing lines, and alternatively to try to improve the decreasing line (see Figure 6), then the increasing one (see Figure 7), and so on, until no improvement is made. The overall solving process is illustrated on Figure 8.

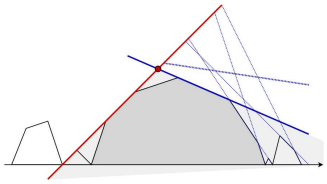

Fig. 6. The lowest intersection point of a fixed increasing line with decreasing lines

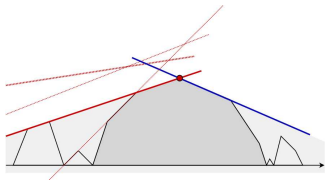

Fig. 7. The lowest intersection point of a fixed decreasing line with increasing lines

Remark 2. Suppose we just improved the decreasing line (Figure 6) and we get a new intersection point. We know that the whole local polyhedron lies inside the cone oriented below and defined by the increasing and the decreasing line. Then all the decreasing lines with a smaller slope than the new decreasing line, *i.e.* with a larger delay $l_{j,i}$, lie completely outside this cone, and therefore cannot be active at the optimum. We can therefore drop these lines in the subsequent rounds. The same is true for the increasing lines.

This gives us a method to compute an integral solution in $O(N^2)$ since once a fractional solution has been found, we can deduce an integral solution with an additional computation in $O(N)$. Indeed, if x is integer, then (x, α) is the desired solution. Otherwise we can compute the α-values α_- and α_+ associated with $\lfloor x \rfloor$ and $\lceil x \rceil$ and take the largest one. Note that since $\lfloor x \rfloor$ (*resp.* $\lceil x \rceil$) is on the increasing phase (*resp.* decreasing phase), only the corresponding constraints are needed to compute α_- (*resp.* α_+) :

$$\alpha_- = \min_{\substack{j \in \mathcal{G}(i) \\ a_j = p}} (\lfloor x \rfloor - o_j)/l_{j,i} \qquad \text{and} \qquad \alpha_+ = \min_{\substack{k \in \mathcal{G}(i) \\ a_k = p}} (o'_k - \lceil x \rceil)/l_{i,k} \tag{32}$$

We call this a primal approach since this is essentially the application of the primal simplex algorithm. However the primal simplex is not applied on $(Loc{-}BO_i^p)$ but on its dual. A dual approach is illustrated on Figure 9. From

[5] Note that in the computation only the α-coordinate is needed since the x-coordinate of the selected intersection point can be computed afterward by $x = o_j + l_{j,i}\alpha$

a theoretical perspective, this approach is outperformed by Megiddo algorithm [7] which allows to find a solution in $O(N)$. However Megiddo algorithm is more complex and generally slower in practice.

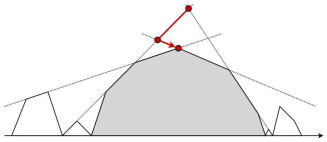

Fig. 8. Finding the fractional optimum with a primal simplex approach

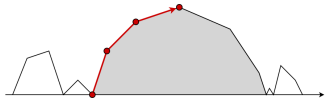

Fig. 9. Finding the fractional optimum with a dual simplex approach

The case of processing times In the case of processing times, no such refinement as Megiddo algorithm is needed in order to obtain a complexity in $O(N)$. In fact, in this case the primal simplex method already runs in $O(N)$. The reason is that in this case, the algorithm stop after three phases. Indeed, we first search for a better decreasing line. However, in the case of processing times, we have $l_{i,j} = p_i$. Since the delays $l_{i,j}$ gives the slope of the decreasing lines, this implies that all of them have the same slope $-1/p_i$ (see Figure 10). Thus, even if initially there are $N-1$ decreasing lines with equations $o'_j - x = \alpha p_i$, the one with the smallest o'_j will be selected after this first phase. In a second phase, we search for a better increasing line. In the third phase, we will not find a better decreasing line, therefore we have the optimal value.

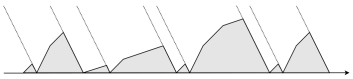

Fig. 10. All the decreasing lines are parallel in the case of processing times

Finding the next improving offset In section 1.1, we explained how the equilibrium-based heuristic replaces a mechanism to push a task to the next feasible location, by one which search for the best location with respect to an additional objective α. The former has the drawback that it does not always produce a solution. However, such a procedure is interesting anyway to implement the best-response method. Indeed, after a first local optimization, we obtain a current solution x_{ref} with a value $\alpha_{\min}$. In the rest of the procedure, we are only interested by polyhedra which could improve this value. As $\alpha_{\min}$ is improved, more and more polyhedra will lie completely below this level and will be skipped (see darker polyhedra in Figure 11).

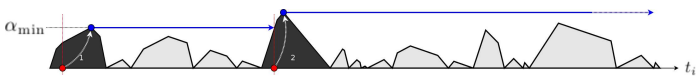

Fig. 11. Propagating up to the next strictly improving offset

Starting from a current solution x_{ref} with value $\alpha_{\min}$, we want to find a new integer solution x greater than the current solution which strictly improves its value. Hence, this solution should satisfy :

$$\alpha_{\min} l_{j,i} < (x - t_j) \bmod g_{i,j} < g_{i,j} - \alpha_{\min} l_{i,j} \qquad \substack{\forall j \in \mathcal{G}(i) \\ a_j = p} \tag{33}$$

However, since the middle expression gives an integer value, we can round the bounds and obtain the equivalent interval constraint (35). Moreover, among all the possible solutions, we would like to find the smallest one :

$$\min \quad x \tag{34}$$

$$s.t. \quad \lfloor \alpha_{\min} l_{j,i} \rfloor + 1 \leq (x - t_j) \bmod g_{i,j} \leq g_{i,j} - \lfloor \alpha_{\min} l_{i,j} \rfloor - 1 \quad \substack{\forall j \in \mathcal{G}(i) \\ a_j = p} \tag{35}$$

$$x \in [x_{\text{ref}}, x_{\text{end}}) \tag{36}$$

Here, x_{end} (which will be defined in the next section) indicates when to stop the search (since there is possibly no strictly improving solution). In order to solve this program, we will start with $x = x_{\text{ref}}$ and propagate until we find a feasible solution. Since we are only interested by contraints associated with tasks $j \in \mathcal{G}(i)$ currently on processor p, *i.e.* satisfying $a_j = p$, we define N_i^p to be the number of such contraints, and we suppose that these contraints are numbered from 0 to $N_i^p - 1$. On a given round, we choose a constraint j and check if it is satisfied. We could check the two inequalities of the interval constraint (35), however we can also remark that this is equivalent to :

$$(x - t_j - \lfloor \alpha_{\min} l_{j,i} \rfloor - 1) \bmod g_{i,j} \leq g_{i,j} - \lfloor \alpha_{\min} l_{i,j} \rfloor - \lfloor \alpha_{\min} l_{j,i} \rfloor - 2 \tag{37}$$

Therefore we can compute $r = g_{i,j} - (x - t_j - \lfloor \alpha_{\min} l_{j,i} \rfloor - 1) \bmod g_{i,j}$. If $r < \lfloor \alpha_{\min} l_{i,j} \rfloor + \lfloor \alpha_{\min} l_{j,i} \rfloor + 2$, the constraint is violated. In this case, we compute the smallest offset x' strictly greater than the current one, and which satisfies the current constraint, *i.e.* we compute $x' > x$ such that $(x' - t_j - \lfloor \alpha_{\min} l_{j,i} \rfloor - 1) \bmod g_{i,j} = 0$. By proposition 2, we have $x' = x + r$. For this offset x', the current constraint is now verified. We set $x = x'$ and continue the process with the next constraint. We cycle along the constraints, until no update is made during N_i^p successive rounds or the offset x becomes greater or equal than x_{end}. In the former case, all the constraints are satisfied by the current offset, otherwise there is no solution and we return the special value $\varnothing$. This procedure is summarized in algorithm 3. It runs in $O(N_i^p n'_{\text{poly}})$ where n'_{poly} is the number of polyhedra on the interval $[x_{\text{start}}, x_{\text{end}})$ [6]. In the worst case, all the tasks are on the same

[6] Let us define a phase to be N_i^p consecutive iterations and let us show that after 2 phases, the current offset x doesn't lie in the same local polyhedron. Note that after a phase, the algorithm either stops or an update has been made. Consider a constraint j updated during the second phase. If j was not updated during the first phase, then this contraint was satisfied during the first phase (hence the current offset was in a given polyhedron of Figure 2), but violated during the second. Therefore, at the end of second phase, the offset has been pushed to the next polyhedron of Figure 2. If j was already updated during the first phase, then the current offset has been pushed forward to the next feasible solution two times for this constraint. Therefore, the local polyhedron has also changed.

processor and we traverse the whole period, hence a complexity in $O(Nn_{\text{poly}})$ which is bounded by $O(NT_i)$.

Algorithm 3 A propagation procedure to find a feasible integral solution

```
1: procedure FindImprovingIntegralSolution(α_min, x_ref, x_end)
2:     x ← x_ref                                          ▷ The current offset
3:     N_sat ← 0                                ▷ The number of satisfied constraints
4:     j ← 0                                      ▷ The current constraint evaluated
5:     while N_sat < N_i^p do
6:         r ← g_{i,j} − (x − t_j − ⌊α_min l_{j,i}⌋ − 1) mod g_{i,j}
7:         if r ≥ ⌊l_{i,j} α_min⌋ + ⌊l_{j,i} α_min⌋ + 2 then
8:             N_sat ← N_sat + 1                    ▷ One more constraint is satisfied
9:         else
10:            x ← x + r                ▷ Otherwise, we go to the first feasible offset
11:            if x ≥ x_end then return ∅          ▷ We reach the end without solution
12:            N_sat ← 1                ▷ We restart counting the satisfied constraint
13:        end if
14:        j ← (j + 1) mod N_i^p                    ▷ We consider the next constraint
15:    end while
16:    return x
17: end procedure
```

Solving the best response problem We are now able to describe a procedure which solves (BO_i^p). We saw that t_i can be supposed to belong to $[0, T_i^p - 1]$. More generally we can start at any initial offset x_{start}, for example the current value of t_i, and we run on the right until we reach the offset $x_{\text{end}} = x_{\text{start}} + T_i^p$. We can compute the local polyhedron (on the right) which contains the current offset. Using the primal simplex method, we solve the associated problem $(Loc{-}BO_i^p)$. We obtain a new local optimum $(x_{\text{ref}}, \alpha_{\min})$. We then use the propagation procedure to reach the next improving solution. We are in a new polyhedron and we restart the local optimization at this point, which gives us a better value. We continue until the propagation mechanism reaches x_{end} (see Figure 11). In the end, we obtain a best offset $t_i \in [x_{\text{start}}, x_{\text{end}})$. If needed, we can consider $t_i \bmod T_i^p$ which is an equivalent solution in $[0, T_i^p - 1]$.

If we use Megiddo algorithm to solve the local problems, or if we use the primal simplex approach in the case of processing times, this procedure runs in $O(Nn_{\text{poly}})$. If we use the primal simplex approach, the local optimizations run in $O(N^2)$, however most of the polyhedra are skipped by the propagation mechanism which runs in $O(Nn_{\text{poly}})$.

3 Results

We test the method on non-harmonic instances generated using the procedure described in [4] : the periods are choosen in the set $\{2^x 3^y 50 \mid x \in [0,4], y \in [0,3]\}$ and the processing times are generated following an exponential distribution and averaging at about 20% of the period of the task.

Table 1 presents the results on 15 instances with $N = 20$ tasks and $P = 4$ processors. Columns 8-10 contain the results of our new version of the heuristic. The value $\mathbf{start}_{10s}$ represents the number of time the heuristic was launched with different initial solutions during 10s. The value $\mathbf{start}_{sol}$ represents the number of starts needed to obtain the best result. Hence, the quantity $\mathbf{time}_{sol} = 10\mathbf{start}_{sol}/\mathbf{start}_{10s}$ gives approximately the time needed to obtain this best result. Column $\mathbf{time}_{sol}$ of the MILP formulation (columns 2-3) represents the time needed by the Gurobi solver to obtain the best solution during a period of 200s[7]. In addition to being much faster, the heuristic sometimes obtains better results (see instances 1 and 3). This table also includes the results of the original heuristic presented in [1, 2] (columns 4-7). In fact, we test the same instances that were used by these authors. However, the way they measure performances differs from ours. For their results, the heuristic was started several times with different initial solutions until a bayesian test was satisfied, which gives a value $\mathbf{time}_{stop}$. We abandoned this bayesian test because the results on column 4 show that on a lot of instances, the process stops with a solution which is far from the best solution found by the MILP. However, since the value $\mathbf{time}_{single}$ represents the time needed for a single run of their version of the heuristic, we can compute a value $\mathbf{starts}_{10s} = 10/\mathbf{time}_{single}$ which measures the average number of starts performed by the original heuristic in $10s$. Compared with the equivalent value $\mathbf{start}_{10s}$ (column 10), we see that our version of the heuristic is incomparably faster (about 3200 times on these instances).

	MILP (200s)		Original heuristic [2] (bayesian test)				New heuristic (10s)			
id	α_{MILP}	$\mathbf{time}_{sol}$	$\alpha_{heuristic}$	$\mathbf{time}_{stop}$	$\mathbf{time}_{single}$	$\mathbf{starts}_{10s}$	$\alpha_{heuristic}$	$\mathbf{starts}_{sol}$	$\mathbf{starts}_{10s}$	$\mathbf{time}_{sol}$
0	2.5	159	2.3	101.33	1.43	7	2.5	30	15062	0.01992
1	2	18	2.01091	5064.67	3.27	3.06	2.01091	15	15262	0.00983
2	1.6	6	1.40455	869.45	1.52	6.58	1.6	2	11018	0.00182
3	1.6	4	1.6	8704.45	4.34	2.3	1.64324	45	11970	0.03759
4	2	5	1.92	1115.51	3.48	2.87	2	1	10748	0.00093
5*	3	7	1.43413	1498.21	1.63	6.13	3	1	20428	0.00049
6	2.5	54	2.3	101.25	1.44	6.94	2.5	30	20664	0.01452
7	2	19	2	302.27	0.23	43.48	2	1	14040	0.00071
8	2.12222	8	1.75794	871.8	1.03	9.71	2.12222	3	17365	0.00173
9*	2	11	2	3541.79	2.42	4.13	2	3	26304	0.00114
10	1.12	6	0.87	368.44	0.72	13.89	1.12	69	28778	0.02398
11	2.81098	20	0.847368	478.63	3.78	2.65	2.81098	7	13355	0.00524
12	1.5	7	1.5	313.74	0.27	37.04	1.5	4	11444	0.00350
13	1.56833	49	1.5	3293.33	1.77	5.65	1.56833	1	25997	0.00038
14	2	8	2	3873	1.85	5.41	2	2	21606	0.00093

Table 1. Results of the MILP, the heuristic of [1], and the new version of the heuristic on instances with $P = 4$ processors and $N = 20$ tasks

These good results have encouraged us to perform additional tests on big instances (50 processors, 1000 tasks). Table 2 presents the results for 10 instances, where $\mathbf{starts}_{1000s}$ is the number of runs performed during 1000s, $\mathbf{starts}_{sol}$ is the

[7] We fix a timeout of 200s because the solver almost never stop even after 1h of CPU time (except for instances 5 and 9 for which optimality has been proved in $7s$ and $20s$ respectively).

round during which the best solution was found, $\mathbf{time}_{\text{sol}}$ is the corresponding time, and $\mathbf{time}_{\text{single}}$ is the average time for one run. This shows that our heuristic can give feasible solutions ($\alpha \geq 1$) in about $1min$, while these instances cannot even be loaded by the MILP solver. In order to evaluate the contribution of the propagation mechanism to the solving process, we also present results where the propagation has been replaced by a simpler mechanism : once we are at a local optimum, we follow the decreasing line active at this point, until we reach the x-axis; this gives us a next reference offset and therefore a next polyhedron. While the impact of the propagation is quite small in the case of small instances, we see on these big instances that the propagation mechanism accelerates the process by a factor of 37 (average ratio of the two $\mathbf{time}_{\text{single}}$ values).

	With propagation					Without propagation				
id	$\alpha_{\text{heuristic}}$	$\mathbf{starts}_{\text{sol}}$	$\mathbf{starts}^{1000s}$	$\mathbf{time}_{\text{sol}}$	$\mathbf{time}_{\text{single}}$	$\alpha_{\text{heuristic}}$	$\mathbf{starts}_{\text{sol}}$	$\mathbf{starts}^{1000s}$	$\mathbf{time}_{\text{sol}}$	$\mathbf{time}_{\text{single}}$
0	1	6	386	13.2	2.59	1	6	16	405.45	66.07
1	1.16452	35	200	197.6	5.01	1.05	5	6	885.06	177.9
2	1.1	78	111	694.17	9.04	1.04	1	4	405.32	261.19
3	1.21795	8	86	62.57	11.69	1.176	1	4	157.18	300.03
4	1	1	105	7.24	9.57	1	1	4	306.34	309.70
5	1.66555	114	121	952.62	8.30	1.58795	1	2	800.95	798.01
6	1	6	240	26.49	4.17	1	6	15	403.05	72.29
7	1.2	37	88	484.91	11.54	1	1	4	278.17	235.44
8	1.39733	47	83	468.93	12.09	1.21127	2	2	1237.5	618.75
9	1.2	1	77	20.58	13.29	1.2	1	2	420.58	487.03

Table 2. Results of the new version of the heuristic on instances with $P = 50$ processors and $N = 1000$ tasks, with $timeout = 1000s$

4 Conclusion

In this paper, we have proposed an enhanced version of a heuristic, first presented in [1, 2], and allowing to solve a NP-hard strictly periodic scheduling problem. More specifically, we present an efficient way to solve the best-response problem. This solution procedure alternates between local optimizations and an efficient propagation mechanism which allows to skip most of the polyhedra. The results show that the new heuristic greatly improves the original one and compares favorably with MILP solutions. In particular, it can handle instances out of reach of the MILP formulation.

A Appendix

Proposition 1. *If* $x \bmod a = 0$*, then* $(-x) \bmod a = 0$*, otherwise* $(-x) \bmod a = a - x \bmod a$*. In particular* $x \bmod a > 0 \Leftrightarrow (-x) \bmod a > 0$.

Proposition 2. *(1) The smallest* $y \geq a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a + (b - a) \bmod c$*. (2) The smallest* $y > a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a + c - (a - b) \bmod c$*. (3) The largest* $y \leq a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a - (a - b) \bmod c$*. (4) The largest* $y < a$ *such that* $(y - b) \bmod c = 0$ *is given by* $y = a - c + (b - a) \bmod c$.

References

1. A. Al Sheikh. Resource allocation in hard real-time avionic systems - Scheduling and routing problems. PhD thesis, LAAS, Toulouse, France, 2011.
2. A. Al Sheikh, O. Brun, P.E. Hladik, B. Prabhu. Strictly periodic scheduling in IMA-based architectures. Real Time Systems, Vol 48, N°4, pp.359-386, 2012.
3. A. Bonfietti, M. Lombardi, L. Benini, M. Milano, Global Cyclic Cumulative Constraint, Proceedings of CPAIOR, pp.81-96, 2012
4. F. Eisenbrand, K. Kesavan, R.S. Mattikalli, M. Niemeier, A.W. Nordsieck, M. Skutella, J. Verschae, A. Wiese. Solving an Avionics Real-Time Scheduling Problem by Advanced IP-Methods. ESA 2010, pp.11-22, 2010.
5. C. Hanen, A. Munier. Cyclic Scheduling on Parallel Processors: An Overview. Université P. et M. Curie, 1994.
6. J. Korst. Periodic multiprocessors scheduling. PhD thesis, Eindhoven university of technology, Eindhoven, the Netherlands, 1992.
7. N. Megiddo. Linear-Time Algorithms for Linear Programming in R^3 and Related Problems. SIAM J. Comput., Vol 12, N°4, pp.759-776, 1983.