Global Cyclic Cumulative Constraint

Alessio Bonfietti, Michele Lombardi, Luca Benini and Michela Milano

DEIS, University of Bologna
Italy

Abstract. This paper proposes a global cumulative constraint for cyclic scheduling problems. In cyclic scheduling a project graph is periodically re-executed on a set of limited capacity resources. The objective is to find an assignment of start times to activities such that the feasible repetition period λ is minimized. Cyclic scheduling is an effective method to maximally exploit available resources by partially overlapping schedule repetitions. In our previous work [4], we have proposed a modular precedence constraint along with its filtering algorithm. The approach was based on the hypothesis that the end times of all activities should be assigned within the period: this allows the use of traditional resource constraints, but may introduce resource inefficiency. The adverse effects are particularly relevant for long activity durations and high resource availability. By relaxing this restriction, the problem becomes much more complicated and specific resource constrained filtering algorithms should be devised. Here, we introduce a global cumulative constraint based on modular arithmetic, that does not require the end times to be within the period. We show the advantages obtained for specific scenarios in terms of solution quality with respect to our previous approach, that was already superior with respect to state of the art techniques.

Keywords: Cyclic scheduling problem, cumulative constraint, filtering algorithm

1 Introduction

Cyclic scheduling problems arise in a number of application areas, such as in hoist scheduling [5], mass production [11],[6], compiler design (implementing scheduling loops on parallel architectures) [14],[10], software pipelining [18], and on data-flow computations in embedded systems [12].

Cyclic scheduling is the problem of assigning starting times of periodic activities such that the periodic repetition modulus (λ) of the overall application is minimal. In other words, the schedule is repeated every λ time units. In a λ-width time window, we may find different repetitions of activities. For example we might have the third repetition of activity i and the fifth repetition of activity j. All activities however should appear once in the period.

Despite traditional constraint-based scheduling techniques have achieved a good level of maturity in the last decade [16], they cannot be applied to cyclic scheduling problems in an efficient way. Two approaches have been proposed:

- the so called blocked scheduling approach [2] that considers only one iteration and repeats it in sequence for an infinite number of times. Since the problem is periodic, and the schedule is iterated infinitely, the method pays a penalty in the quality of the schedule obtained.
- the unfolding approach [17] that schedules a number of consecutive iterations of the application. Unfolding often leads to improved blocked schedules, but it also implies an increased size of the instance.

In our previous work [4], we have proposed a model based on modular arithmetic, taking into account temporal and resource constraints. A modular precedence constraint along with its filtering algorithm was proposed. The main innovation of that paper was that while classical modular approaches fix the modulus and solve the corresponding (non periodic) scheduling problem, in our technique the bounds for the modulus variables are inferred from the activity and iteration variables. We have shown that our technique greatly outperforms both the blocked and the unfolding approaches in terms of solution quality, and also outperforms *non constraint-based* (heuristic) modular approaches, such as Swing Modulo Scheduling [9].

The main drawback of our previous approach was the underlying hypothesis that the end times of all activities should be assigned within the modulus. Thanks to this assumption, we can reuse traditional resource constraints and filtering algorithms. However the solution quality can be improved by relaxing this hypothesis.

In this paper we propose a Global Cyclic Cumulative Constraint (GCCC) that indeed relaxes this hypothesis. We have to schedule all the start times within the modulus λ, but we have no restriction on end times. The resulting problem is far more complicated, as enlarging the modulus produces a reduction of the end time of the activities. Figure 1 explains the concept. Suppose the grey activity requires one unit of a resource of capacity 3. If the modulus value is D, then the activity can be scheduled as usual. If the modulus is reduced to C, the starting time of the activity is the same, while the "modular end time" is c and the resource consumption is 2 between 0 and c. If the modulus is further reduced to B the modular end time increases to b. Finally, if the modulus is reduced to A, the modular end point becomes a and the resource consumption is 3 between 0 and a.

In this paper, we propose a filtering algorithm for the GCCC and we show the advantages in terms of solution quality w.r.t. our previous approach that was already outperforming state of the art techniques.

The paper is structured as follows: in section 2 we formally define the problem considered, and we recall the model from [4]. Section 3 is devoted to the Global Cyclic Cumulative Constraint and its filtering algorithm. Experimental results and related work conclude the paper.

2 The Problem

The cyclic scheduling problem is defined on a directed graph $\mathbb{G}(\mathbb{V}, \mathbb{A})$ called project graph. Elements in $\mathbb{V}$ ($|\mathbb{V}| = n$) are nodes that represent activities with fixed durations d_i, and elements in $\mathbb{A}$ ($|\mathbb{A}| = m$) are arcs representing dependencies between pair of activities. The problem considers a set of limited capacity resources: for each resource k its maximum capacity is R_k. Each activity i has a set of resource requirements $r_{i,k}$ for all resources k required by activity i.

The problem is periodic: thus the project graph (and consequently each activity) is executed an infinite number of times. We refer to $start(i, \omega)$ as the starting time of activity i at repetition ω.

Arcs in the graph represent precedence constraints: an arc (i, j) in the graph can be interpreted as $start(j, \omega) \geq start(i, \omega) + d_i$. More precisely, an edge (i, j) in the graph $\mathbb{G}$ might be associated with a minimal time lag $\theta_{(i,j)}$ and a repetition distance $\delta_{(i,j)}$. Every edge of the graph can therefore be formally represented as:

$$start(j, \omega) \geq start(i, \omega - \delta_{(i,j)}) + d_i + \theta_{(i,j)} \tag{1}$$

In other words, the start time of activity j at iteration ω must be higher than the sum of the time lag θ and the end time of i at ω shifted by the repetition distance δ of the arc. Note that, since $end(j, \omega) = start(j, \omega) + d_i$, the equation 1 can be rewritten as $start(j, \omega) \geq end(i, \omega - \delta_{(i,j)}) + \theta_{(i,j)}$

In a periodic schedule, the start times of different repetitions of the same activity follow a static pattern: $start(i, \omega) = start(i, 0) + \omega \cdot \lambda$, where $\lambda > 0$ is the duration of an iteration (i.e. the iteration period, or *modulus*) and $start(i, 0)$ is the start time of the first execution. Hence, a cyclic scheduling problem consists of finding a feasible assignments for $start(i, 0)$ such that all precedence constraints are consistent, no resource capacity is exceeded and the modulus λ is minimized.

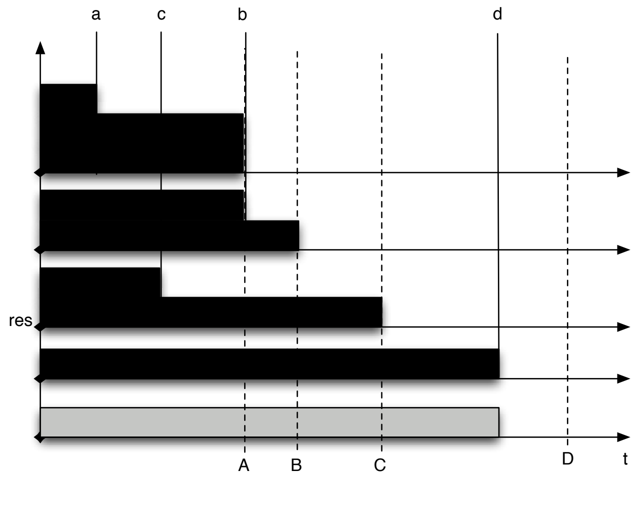

Fig. 1. Resource Profiles w.r.t different modulus values

For this problem, in a previous paper [4], we have proposed a model based on modular arithmetic for dealing with periodic schedules. We proposed a cyclic precedence constraint along with a filtering algorithm. In that paper, however, we made a simplifying hypothesis: the end time of each activity has to be placed within the modulus. Since, with this simplification, activities are not scheduled across iterations, a traditional cumulative constraint can be used to model resources. Note that, since the schedule is repeated every λ time units, the resource requirement at a time point t may be caused by activities from different schedule repetitions (i.e. with different ω values). By relaxing the restriction on the end times, we can obtain substantially better schedules in terms of resource usage and overlapping among iterations, but the resulting problem is far more complicated.

2.1 Modular Representation for Cyclic Schedules

In this section, we recall some modular arithmetic notions that constitute the foundations of our cyclic scheduling solver. The main underlying idea is to focus on a λ-width time window in the periodic phase. First, consider that the start time of iteration 0 of activities i (i.e. $start(i,0)$) can be expressed as:

$$start(i,0) = s_i + \beta_i \cdot \lambda$$

where s_i is a value in the half-open interval $[0, \lambda[$ and β_i is an integer number. In practice, β_i is the period repetition when activity i is first scheduled and s_i is its relative start time within the period, i.e. its *modular start time*. Analogously, the end time $end(i,0)$ can be decomposed into a *modular end time* e_i and an iteration number η_i.

In [4] the end and the start time of an activity i were forced to belong to the same iteration, i.e. $\beta_i = \eta_i$. In this paper we relax such hypothesis and we allow $\eta_i \geq \beta_i$. As a side effect, this allows e_i to be less than s_i, i.e. an activity can be executed across different iterations. Since $end(i,0) = start(i,0) + d_i$, we have $e_i + \eta_i \cdot \lambda = s_i + \beta_i \cdot \lambda + d_i$ and hence:

$$d_i = e_i - s_i + (\eta_i - \beta_i) \cdot \lambda$$

Moreover, we have $\eta_i - \beta_i = \left\lfloor \frac{d_i}{\lambda} \right\rfloor$, which means that η_i is unambiguously determined once β_i and λ are known. In Figure 1, the modular start time is 0 and remains constant when the modulus is changed. The modular end time is a, b, c or d depending on the modulus value.

Using the modular formulation, a precedence constraint $start(j,\omega) \geq start(i, \omega - \delta_{(i,j)}) + d_i + \theta_{(i,j)}$ is rewritten as:

$$s_j + \beta_j \cdot \lambda + \omega \cdot \lambda \geq s_i + \beta_i \cdot \lambda + (\omega - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)}$$

performing the usual eliminations we have the following inequality, no longer depending on ω:

$$s_j + \beta_j \cdot \lambda \geq s_i + (\beta_i - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)} \tag{2}$$

In [15] we have shown that the amount of a resource k requested by activity i in a λ-width time window at instant t is given by the following expression:

$$rq_{i,k}(s_i,t,\lambda) = \begin{cases} r_{i,k} \cdot \left(\left\lfloor \dfrac{d_i}{\lambda} \right\rfloor + 1 \right) & \text{if } s_i \leq t < e_i \text{ or } e_i < s_i \leq t \text{ or } t < e_i < s_i \\ r_{i,k} \cdot \left\lfloor \dfrac{d_i}{\lambda} \right\rfloor & \text{otherwise} \end{cases}$$

In other words, the resource usage is given by a constant factor $r_{i,k} \cdot \lfloor \frac{d_i}{\lambda} \rfloor$, plus an additional $r_{i,k}$ in case the considered time point lies within the modular duration, defined as $d_i \ mod \ \lambda$. Note that the case where $e_i < s_i$ is also taken into account. In Figure 1, the constant usage factor is 1 in case the modulus is B, C or D and 2 when the modulus is A. Observe that if $\beta_i = \eta_i$, the constant usage factor is zero and $rq_{i,k}(s_i,t,\lambda)$ becomes a classical resource usage function. This explains why forcing the end times to be within the modulus allows the use of classical resource constraints.

2.2 Model

The model we devised is based on three classes of variables: two of them are related to single activities and the last one to the whole schedule. For each activity i we have the starting time within the modulus s_i (also called modular starting time) and the iteration β_i. The modular starting time of each activity has an initial domain $[0..\lambda[$, while the iterations have the domain [- $\|\mathbb{V}\|$.. + $\|\mathbb{V}\|$] where $\|\mathbb{V}\|$ is the number of nodes.

Each activity i is characterized during search by its earliest start time EST_i and its latest end time LST_i (i.e. respectively the minimum and maximum values in the domain). Clearly, having each activity a fixed duration, the end time of the activity has the following domain bounds: the earliest end time $EET_i = EST_i + d_i$ and the latest end time $LET_i = LST_i + d_i$. Since we are working on a circular time wheel, we consider the modular earliest end time $mEET_i = EET_i \mod \lambda$. In addition, we have a variable related to the whole schedule: the modulus decision variable whose domain is λ $]0..MAX\_TIME]$ where MAX_TIME represents the sum of the execution times of the activities and the sum of the time lags of the edges.

The time model we devised is an extension of the Simple Temporal Network Model (STNM). Each node i of the graph is represented with a pair of time points s_i, e_i with associated time windows, connected by a directional binary constraints of the form:

$$s_i \xrightarrow{[d_i]} e_i$$

where d_i (the execution time of activity i) is the distance between the activity starting point s_i and the activity endpoint e_i, meaning that $e_i = s_i + d_i$.

We extend the STNM with a new precedence edge formulation: each edge (i, j) of the graph, described by (2), is represented as:

$$e_i \xrightarrow{[\theta_{(i,j)},\eta_i,\beta_j,\delta_{(i,j)}]} s_j$$

where $\theta_{(i,j)}$ is the minimal time lag between the end of i (e_i) and the start of j (s_j). The construct also takes in account the iteration numbers η_i, β_j and their minimal iteration distance $\delta_{(i,j)}$. This precedence is modeled through a dedicated *Modular Precedence Constraint* [4]. The filtering for a single precedence relation constraint achieves GAC and runs in constant time.

Dealing with resources in cyclic scheduling where activities can be scheduled across iterations implies modelling resources with a new resource constraint. The main contribution of this work is the development of a new cyclic cumulative resource constraint that we describe in the next section.

3 Global Cyclic Cumulative Constraint GCCC

The Global Cyclic Cumulative Constraint for resource k ensures consistency in the use of the resource:

$$\sum_{i \in \mathbb{V}} rq_{i,k}(s_i, t, \lambda) \leq R_k \quad \forall t \in [0, ..\lambda[$$

As the GCCC refers to a single resource, for the sake of readability, we remove the k index from the requirement functions. Hence $r_{i,k}$ becomes r_i and R_k becomes R. The constraint is inspired by the timetable filtering for the cumulative constraint ([16]). On this purpose, the function $rq_i(s_i, t, \lambda)$ can be generalized as follows:

$$\overline{rq}_i(EST_i, LST_i, t, \lambda)$$

If $EST_i = LST_i$ the generalized function boils down to the $rq_i(s_i, t, \lambda)$ function while if $EST_i + d_i \leq LST_i$ then $\overline{rq}_i(EST_i, LST_i, t, \lambda) = 0$. Otherwise the function returns the resource consumption of the activity as if it started at LST_i and executed for $EST_i + d_i - LST_i$ time units.

The constraint is composed by three procedures:

- **Trigger**: the procedure is executed whenever any variable domain is changed. The aim of this algorithm is to update the time tabling data structure.
- **Core**: the algorithm is executed at the end of all trigger procedures and it is structured in two independent phases:
  - Start Time Propagation: it propagates the lower bound of the start time variables.
  - Modulus Propagation: this phase computes the minimum lambda needed to guarantee the feasibility of the solution.
- **Coherence**: the procedure is executed whenever the modulus upper bound changes. The procedure modifies the data structure to guarantee the coherence with the new λ bound.

3.1 Start Time Filtering Algorithm

The filtering algorithm guarantees that the start time of each activity is not lower than the minimum instant where enough resources are available:

$$s_i \geq \min_{t \in [0, \lambda[} : \sum_{j \in \mathbb{V} \setminus \{i\}} \overline{rq}_j(EST_j, LST_j, t', \lambda) \leq R - rq_i(t, t', \lambda) \quad \forall t' \in [t, t + \overline{d}_i]$$

Similarly to the timetable approach, we adopt a data structure to store the following quantity

$$\sum_{i \in \mathbb{V}} \overline{rq}_i(EST_i, LST_i, t', \lambda)$$

and this value is maintained for the LST_i and $mEET_i$ of all the activities.

Intuitively the algorithm proceeds as follows: for each unbounded activity x, starting from its Earliest Start Time (EST_x), the algorithm searches the resource profile for a schedulability window. A schedulability window is a time slice large enough and with enough resources to guarantee the activity execution. The detection stops when a window is found or the search exceeded the Latest Start Time (LST_x). As the solver is based on modular arithmetic, the detection procedure follows a modular time wheel. Hence, the times are represented by a circular queue modulated by the upper bound[1] of the modulus variable $\overline{\lambda}$. Whenever a time point t exceeds the modulus, it can be expressed as the sum of the modular time $t' = t \mod \overline{\lambda}$ and its remaining offset $\overline{\lambda} \cdot \lfloor \frac{t}{\overline{\lambda}} \rfloor$. The filtering algorithm has an asymptotic complexity of $O(n^2)$.

Data Structure As stated in section 2.2, each activity $x \in \mathbb{V}$ has 5 time indexes: two of them are related to the start time point, namely the Earliest Start Time EST_x and the Latest Start Time LST_x, and three related to the end time point, the Earliest End Time EET_x, the modular Earliest End Time $mEET_x$ and the Latest End Time LET_x.

The constraint relies on an ordered circular queue $\Omega[0..(\|\mathbb{V}\| * 2)]$ where each activity $x \in \mathbb{V}$ is represented via two queue items, respectively corresponding to its LST_x and its $mEET_x$. Each item $\Omega(\tau)$ stores a time value $\Omega[\tau].time$ and the total resource usage at such time instant:

$$\Omega[\tau].res = \sum_{i \in \mathbb{V}} \overline{rq}_i(EST_i, LST_i, \Omega[\tau].time, \lambda)$$

Additionally, we store whether the item corresponds to a start time (LST) or to and end time ($mEET$). Finally with $\Omega[\tau].activity$ we refer to the activity the time point $\Omega[\tau]$ belongs to. This information is needed to perform filtering on the λ variable (see Section 3.2).

[1] Note that we use the upper bound of the lambda variable as it is the least constraining value in the domain.

```
    Data: Let $S_i$ be the set of activities not already scheduled
 1  begin
 2      forall the $x \in S$ do
 3          $canStart = EST_x$
 4          $\tau_0 = FindElement(EST_x)$
 5          $feasible = true$
 6          if $\Omega[\tau_0].res + rq_x(canStart, \Omega[\tau_0].time, \overline{\lambda}) > R$ then
 7              $feasible = false$
 8          forall the $\tau \in \Omega$ $starting\ from\ \tau_0$ do
 9              $offset = 0$
10              if $\tau < \tau_0$ then
11                  $offset = \overline{\lambda}$
12              if $feasible$ && $canStart + d_x \leq offset + \Omega[\tau].time$ then
13                  $EST_x \leftarrow canStart$
14                  $Stop\ Propagation\ on\ x$
15              $r_x^* = rq_x(canStart, \Omega[\tau].time, \overline{\lambda}) - \overline{rq}_x(EST_x, LST_x, \Omega[\tau].time, \overline{\lambda})$
16              if $\Omega[\tau].res + r_x^* > R$ then
17                  if $offset + \Omega[\tau].time > LST_x$ then
18                      $fail()$
19                  $feasible = false$
20                  $canStart = \Omega[\tau].time$
21              else
22                  if $not\ feasible$ then
23                      $feasible = true$
24                      $canStart = \Omega[\tau].time$
```

Algorithm 1: Core: Start Times Filtering Algorithm

The algorithm The pseudo-code is reported in **Algorithm** 1 where S is the set of unscheduled activities. The start time variable *canStart* initially assumes the value of the Earliest Start Time of the selected unbounded activity x. It represents the candidate start time of the schedulability window and is associated to the flag $feasible$.

The function τ $FindElement(t)$ returns the index of the element $y = \Omega[\tau]$ in the vector, such that $y = \mathrm{argmax}_{\tau \in \Omega}\{\Omega[\tau].time \mid \Omega[\tau].time \leq t\}$. Intuitively, the function returns the index of the maximum time point that precedes the activity (or the maximum time point having the same time). τ_0 (line 4) is the index of the time point EST_x.

In lines 5-7 the algorithm verifies if the time $\Omega[\tau_0].time$ is feasible for the activity x: as stated in section 2.1, r_x is the amount of resource requested by the activity x while R represents the total capacity. Note that the feasibility at time $\Omega[\tau_0].time$ implies the feasibility at time *canStart*, since the resource requested is the same. The value $rq_x(canStart, \Omega[\tau_0].time, \overline{\lambda})$ is the amount of resource requested by the activity x at time $\Omega[\tau_0].time$ assuming it is scheduled at $canStart$.

At line 9, the *schedulability window* search phase starts. Starting from the activity at index $\tau = \tau_0$, the algorithm traverses (with increasing time order) the whole circular queue Ω. Whenever the index τ refers to an element that temporally precedes the starting point τ_0, the *offset* is set to $\overline{\lambda}$. In fact, $\tau < \tau_0$ implies that τ has already crossed the modulus value $\overline{\lambda}$.

At every time point $\Omega[\tau].time$ the algorithm tests the feasibility for the activity x (lines 12-14). The schedulability window is defined starting from the candidate start time $canStart$ and finishes at the current time $offset + \Omega[\tau].time$. If the initial time point is feasible ($feasible = true$) and the window is at least large as the execution time of the activity, the algorithm has found a feasible window. Then, it sets $EST_x = canStart$ and proceeds with another unbounded activity.

At line 15 the algorithm computes r^*_x that is the difference between (1) the resource request at the current time assuming x is scheduled at time $canStart$ and (2) the resource request at the current time considering only the obligatory region of the activity. Note that $\Omega[\tau].res$ is the sum of all resource requests (considering the obligatory region of the activities) at time $\Omega[\tau].time$ and it already covers the amount (2). $r^*_x + \Omega[\tau].res$ is the total consumption in the hypothesis that x is scheduled at time $canStart$. If the amount exceeds the capacity, the candidate start time variable and its flag are updated; moreover, if the current time $offset + \Omega[\tau].time$ exceeds the LST_x the activity cannot be scheduled and the constraint fails (lines 17-18).

Finally, if the resource request is satisfied at time $\Omega[\tau].time$ (line 21) and $feasible = false$, the variable $canStart$ and the flag $feasible$ are updated.

3.2 Modulus Filtering Algorithm

In cyclic scheduling, it is possible to reduce the cumulative usage at time t by increasing the modulus. As a consequence, unlike in classical scheduling, the obligatory parts in the current schedule may enforce a non-trivial lower bound on the feasible λ. The goal of lambda filtering is to find the minimum instant where sufficient resources are available for the current schedule. Formally:

$$\lambda \geq \min_{t \in [0,\lambda[} : \sum_{i \in \mathbb{V}} \overline{rq}_j(EST_i, LST_i, t, \lambda) \leq R \quad \forall t \in [0, \lambda[$$

The algorithm makes use of the same data structure Ω as before. However, in this case the modular end times $mEET_i$ are computed based on the modulus *lower* bound[2], i.e. $\underline{\lambda}$. Figure 2 shows two different resource profiles of the same schedule. The former corresponding to the maximum modulus value $\overline{\lambda}$ and the latter corresponding to the minimum value $\underline{\lambda}$. Note that, with $\underline{\lambda}$, activities A and B now cross the modulus, increasing the resource consumption at time 0. This causes a resource over-usage, represented by the shaded area.

[2] The modulus lower bound corresponds to the most constrained resource profile.

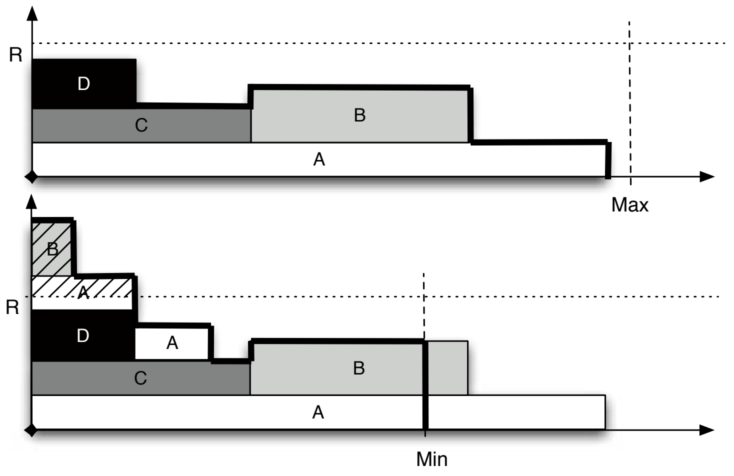

Fig. 2. Resource Profile of a partial allocation with minimum and maximum modulus.

The modulus filtering algorithm reduces the over-usage by increasing the lower bound $\underline{\lambda}$. This is done in an iterative fashion, by repeatedly computing the over-usage integral and pushing the λ lower bound. The filtering algorithm has an asymptotic complexity of $O(k \cdot n \cdot log_n)$ where k is the number of iterations.

Algorithm 2 reports the pseudo-code for the filtering procedure: the process is repeated until the resource over-usage becomes 0. The cumulative over-usage amount at each iteration is referred to as En. As a first step, the algorithm updates and reorders the data structure Ω (lines 4-16). This step is necessary since $\underline{\lambda}$ is changed at each iteration, causing a modification of all the modular end times $mEET$. Then the algorithm computes the resource request at the first time point (lines 6-7). This is the schedule starting consumption: when each of the following Ω items is processed, the resource is increased or decreased depending on whether the item corresponds to a start or an end time (lines 13-16). The step at line 10 is necessary to avoid counting twice the requirement of $\Omega(\tau)$ items with $\Omega(\tau).time = \Omega(0).time$.

At line 17, the procedure checks if the resource consumption of the previous $(\tau - 1)$ time point exceeds the resource. In this case, the cumulative resource over-usage on the time window $[\Omega(\tau - 1).time, \Omega(\tau).time[$ is summed to the current En quantity. At line 19, the algorithm computes a new lower bound on λ by dividing the cumulative over-usage amount by the capacity of the resource.

4 Experimental Results

Cyclic scheduling allows to improve resource efficiency by partially overlapping different schedule iterations. In particular, it is possible to exploit available resources to reduce the period, even when the makespan cannot be further minimized (e.g. due to precedence constraints). Loops in the project graph limit

```
Data: Let $\Omega[\tau].activity$ be the activity the time point $\Omega[\tau].time$ refers to
1  begin
2    repeat
3      $En = 0$
4      $\Omega[0].res = 0$
5      Update and reorder the modular end times in $\Omega$, given the new $\underline{\lambda}$
6      forall the $x \in \mathbb{V}$ do
7        $\Omega[0].res = \Omega[0].res + \overline{rq}_x(EST_x, LST_x, \Omega[0].time, \underline{\lambda})$
8      for $\tau = 1; \tau < \|\Omega\|; \tau = \tau + 1$ do
9        let $y$ be the activity corresponding to $\Omega(\tau)$
10       if $\Omega[0].time = \Omega[\tau].time$ then
11         $\Omega[\tau].res = \Omega[0].res$
12       else
13         if $\Omega[\tau]$ corresponds to $LST_i$ then
14           $\Omega[\tau].res = \Omega[\tau-1].res + r_y$
15         else
16           $\Omega[\tau].res = \Omega[\tau-1].res - r_y$
17       if $\Omega[\tau-1].res > R$ then
18         $En = En + (\Omega[\tau-1].res - R) \cdot (\Omega[\tau].time - \Omega[\tau-1].time)$
19     $\underline{\lambda} \leftarrow \underline{\lambda} + \frac{En}{R}$
20   until $En = 0$
```

Algorithm 2: Core: Modulus Filtering Algorithm

the degree of such an improvement, since the period is lower bounded by the Maximum Cycle Ratio. If the graph is acyclic, however, the throughput can be arbitrarily increased by adding resource capacity. This is not just a theoretical consideration: a large number of practical problems (e.g. in VLIW compilation or stream computing) is described by project graphs with small cycles or no cycle at all. In such a case, identifying the optimal throughput/resource-usage trade-off if *the* primary optimization problem.

By allowing activities to cross different iterations, our approach enables to better exploit resources. Moreover, the period can now be smaller than the activity durations. Since this comes at the price of more complex filtering, investigating the achievable improvements in term of solution quality is of primary importance. On this purpose, we compared the approach described in this paper with the one we presented in [4], where start/end time of each activity were constrained to belong to the same iteration.

Benchmarks The comparison is performed on two groups of 20 synthetically generated project graphs, respectively consisting of cyclic and acyclic graphs. Durations are unevenly distributed: in particular, around 10% of the activities in each graph is unusually long (one order of magnitude more than the others). Since we are interested in investigating throughout/resource trade-offs, the

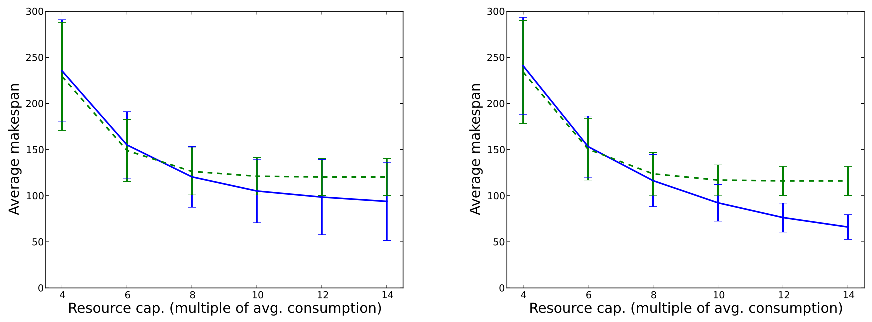

Fig. 3. Makespan over resource capacity for cyclic (left) and acylic (right) graphs.

graphs have fixed size (30 activities) and we performed experiments by varying the resource availability.

Search Methods In [4], we have proposed a search strategy based on the generalization of the classical Schedule or Postpone method [16]. The method leverages resource propagation and symmetry breaking constraints and is shown to be very effective. Unfortunately, as described in Section 2, if activities are allowed to cross different iterations, resource propagation becomes dependent on λ. In particular, small reductions of the period upper bound $\overline{\lambda}$ (e.g. due to bounding constraints when a feasible solution is found) result in negligible changes in resource propagation. As a side effect, the effectiveness of the generalized Schedule or Postpone method is severely reduced. To cope with this issue, we adopt for our new approach an iterative search process: in particular we perform binary search on the bound $\overline{\lambda}$ and use generalized Schedule or Postpone within each iteration.

Testing Platform Both the approach in [4] and the GCCC one are implemented in IBM ILOG Solver and Scheduler 6.7. All the experiments are performed on a 3.2GHz machine with 8GB of RAM. A 300 seconds time limit was set on each solution attempt.

On the purpose of investigating the throughput/resource trade-off, we solved a set of period minimization problems with different resource availability levels. In detail, we considered a single resource and activities a_i in each graph were labeled with random resource requirements r_i, following a normal distribution. The resource capacity R ranges between 4 times and 14 times the average consumption level. The minimum value is chosen so as to guarantee problem feasibility, while the maximum one is set to assess the solution quality in case of abundant resources.

Figure 3 shows the average makespan (at the end of the solution process) over a varying resource capacity. The time limit was hit in all cases. The vertical bars report the corresponding standard deviation. The solid line corresponds to the current approach and the dashed one to [4]. The approach proposed in this paper obtains considerably better results for higher capacity values, i.e. the scenario when we expected the highest benefit from allowing activities to cross iterations. The gap is larger for acyclic graphs, where the lack of loops enables to fully exploit available resources.

The makespan difference corresponds to a much larger gap in terms of total resource idle time, especially in case of large resource capacities. This is reported in Figure 4, where the amount of idleness is shown to grow according to a roughly quadratic law for the previous approach. The growth is much slower for the GCCC one (in fact, it is approximately constant for acyclic graphs). Idle time is an important measure of how efficiently the resources are used and directly translates to platform/machine costs in a practical setting.

Interestingly, the two approaches have comparable performance for small capacity values. This suggest that the time limit is not severely limiting the search effectiveness. This is a relevant remark, since we expected the GCCC approach to be considerably slower in finding good solutions. More details are reported in the histograms (Figure 5), that show the instance count, grouped by the time (in seconds) employed by each method to get 1% close to the final best solution. As one can see, our previous approach is indeed faster on average, but both methods manage to provide high quality schedules in a matter of few seconds.

5 Related Works

The cyclic scheduling literature mainly arises in industrial and computing contexts. While there is a considerable body of work on cyclic scheduling in the OR literature, the problem has not received much focus from the AI community ([6] is one of the few approaches).

When coping with periodic scheduling, one can basically re-use constraint-based scheduling techniques in two ways: the first adopts the so called blocked

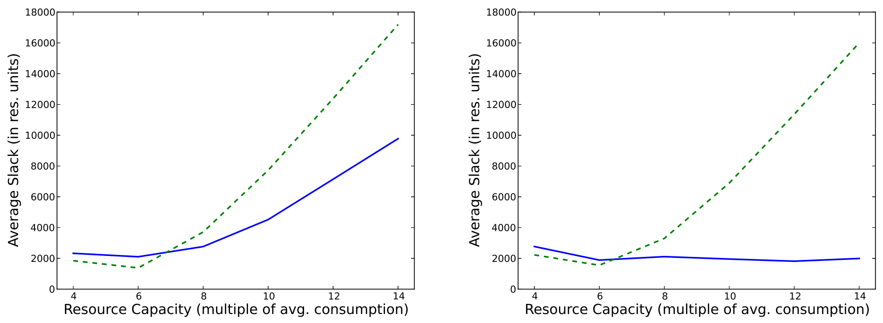

Fig. 4. Idless over resource capacity for cyclic (left) and acylic (right) graphs.

scheduling approach [2]. Only one iteration of the application is scheduled. Then it is repeated an infinite number of times. The advantage of this method is the use of traditional scheduling techniques. The main substantial drawback is that the method pays a penalty in the quality of the schedule obtained. In fact, there is no overlap among iterations, and this implies a under-utilization of available resources.

To partially overcome this issue, the unfolding approach [17] has been proposed. The idea is to schedule a number n of application iterations. The resulting unfolded application is scheduled and repeated as a whole. Clearly the overlapping among the n considered iterations is possible. The solution quality is improved w.r.t. blocked schedules, thanks to limited resource under-utilization. However, unfolding also implies an increased size of the instance. Being the problem NP-complete, multipliying by n the number of activities to be scheduled leads to an exponential explosion in the solution time.

An alternative is to abandon the idea of re-using traditional scheduling techniques and adopt a cyclic scheduling approach. Advanced complete formulations are proposed in [8] by Eichenberger and in [7] by Dupont de Dinechin; both approaches are based on a time-indexed ILP model; the former exploits a decomposition of start times to overcome the issue with large makespan values, while the latter has no such advantage, but provides a better LP relaxation. In [1] the authors report an excellent overview of the state-of-the-art formulations and present a new model issued from Danzig-Wolfe Decomposition. Other good overviews of complete methods can be found in [10].

To the best of our knowledge, most of the state-of-the-art approaches are based on iteratively solving resource subproblems obtained by fixing the period value; fixing λ allows solving the resource constrained cyclic scheduling problem via an integer linear program (while modeling λ as an explicit decision variable yields non-linear models). The obvious drawback is that a resource constrained scheduling problem needs to be repeatedly solved for different λ values.

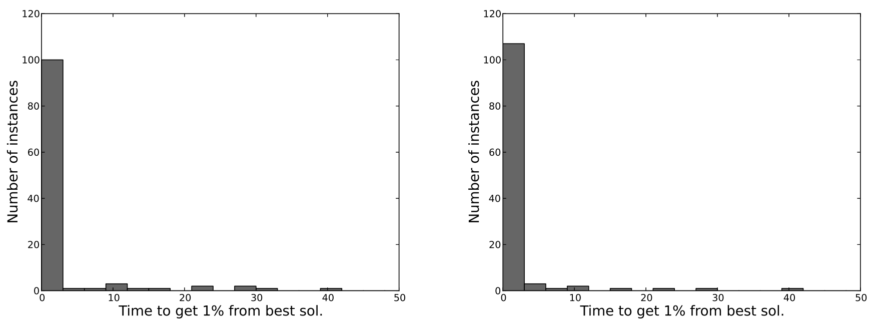

Fig. 5. Number of instances, grouped by the time to get 1% close the best solution found, for the GCCC approach (left) and the one from [4] (right).

Compared to these approaches, our method does not require to fix a λ value, thanks to the use of a global constraint to model resources restrictions. In the context of a binary search scheme for the period optimization, this considerably reduces the number of search steps.

Several heuristic approaches have been proposed. An heuristic approach is described in [18], wherein the algorithm, called *iterative modulo scheduling*, generates near-optimal schedules. Another interesting heuristic approach, called *SCAN* and in part based on the previous one, is presented in [3]. The latter method is based on an ILP model. A state of the art incomplete method is *Swing Modulo Scheduling* approach, described in [13], [14], and currently adopted in the GCC compiler [9].

Heuristic approaches compute a schedule for a single iteration of the application: the schedule is characterized by the value of the makespan (the *horizon*) and by an *initiation interval* which defines the real throughput. However, the *horizon* could be extremely large, with implications on the size of the model. Our model is considerably more compact, since we schedule a λ-width window with no need to explicitly consider the *horizon*.

6 Conclusions

In this paper we present a new global cumulative constraints GCCC that models discrete and finite resources in cyclic scheduling problems. We relax the hypothesis stated in [4] to have the end time and the start times of each activity belonging to the same iteration. The resulting problem is far more complex and requires the definition of new filtering algorithms on activity start times and on the modulus variable.

We show the advantages in terms of solution quality w.r.t. our previous approach that was already outperforming state of the art techniques. The experiments highlight that our approach obtains considerably better results in terms of solution quality for high capacity values. Moreover, the results show that, working with acyclic graphs, the GCCC approach obtains an approximately constant resource idle time.

Further investigation will be devoted to the design of cyclic scheduling heuristic algorithms and their comparison with complete approaches.

Acknowledgement

The work described in this publication was supported by the SMECY Project, JTI ARTEMIS, Grant agreement no.: 100230

References

1. M. Ayala and C. Artigues. On integer linear programming formulations for the resource-constrained modulo scheduling problem, 2010. http://hal.archives-ouvertes.fr/docs/00/53/88/21/PDF/ArticuloChristianMaria.pdf.

2. Shuvra S. Bhattacharyya and Sundararajan Sriram. *Embedded Multiprocessors - Scheduling and Synchronization (Signal Processing and Communications) (2nd Edition).* CRC Press, 2009.
3. F. Blachot, B. Dupont de Dinechin, and G. Huard. SCAN: A Heuristic for Near-Optimal Software Pipelining. *In Euro-Par 2006 Parallel Processing, Lecture Notes in Computer Science*, 4128:289–298, 2006.
4. Alessio Bonfietti, Michele Lombardi, Luca Benini, and Michela Milano. A Constraint Based Approach to Cyclic RCPSP. In *CP2011*, pages 130–144, 2011.
5. Haoxun Chen, Chengbin Chu, and J.-M. Proth. Cyclic scheduling of a hoist with time window constraints. *Robotics and Automation, IEEE Transactions on*, 14(1):144 –152, feb 1998.
6. D.L. Draper, A.K. Jonsson, D.P. Clements, and D.E. Joslin. Cyclic scheduling. In *Proc. of IJCAI*, pages 1016–1021. Morgan Kaufmann Publishers Inc., 1999.
7. B. Dupont de Dinechin. From Machine Scheduling to VLIW Instruction Scheduling, 2004. http://www.cri.ensmp.fr/classement/doc/A-352.ps.
8. A.E. Eichenberger and E.S. Davidson. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices*, 32(5):194–205, 1997.
9. Mostafa Hagog and Ayal Zaks. Swing modulo scheduling for gcc, 2004.
10. C. Hanen and A. Munier. *Cyclic scheduling on parallel processors: an overview*, pages 193–226. John Wiley & Sons Ltd,, 1994.
11. Claire Hanen. Study of a np-hard cyclic scheduling problem: The recurrent job-shop. *European Journal of Operational Research*, 72(1):82 – 101, 1994.
12. Manjunath Kudlur and Scott Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of PLDI*, volume 43, pages 114–124, May 2008.
13. Josep Llosa, Antonio Gonzalez, Eduard Ayguade, and Mateo Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT'96*, pages 80–87, 1996.
14. Josep Llosa, Antonio Gonzalez, Eduard Ayguade, Mateo Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. on Comps.*, 50(3):234 – 249, 2001.
15. Michele Lombardi, Alessio Bonfietti, Michela Milano, and Luca Benini. Precedence constraint posting for cyclic scheduling problems. In *CPAIOR*, pages 137–153, 2011.
16. Baptiste P., Le Pape C., and Nuijten W. *Constrains-based scheduling: applying Constraint Programming to Scheduling.* Kluwer, 2001.
17. K.K. Parhi and D.G. Messerschmitt. Static rate-optimal scheduling of iterative data-flow programs via optimum unfolding. *IEEE Transactions on Computers*, 40(2):178–195, 1991.
18. R.B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of MICRO-27*, pages 63–74. ACM, 1994.