Precedence Constraint Posting for Cyclic Scheduling Problems

Michele Lombardi, Alessio Bonfietti, Michela Milano, and Luca Benini

DEIS, University of Bologna,
Viale del Risorgimento 2, 40136 Bologna, Italy
{michele.lombardi2,alessio.bonfietti,michela.milano,luca.benini}@unibo.it

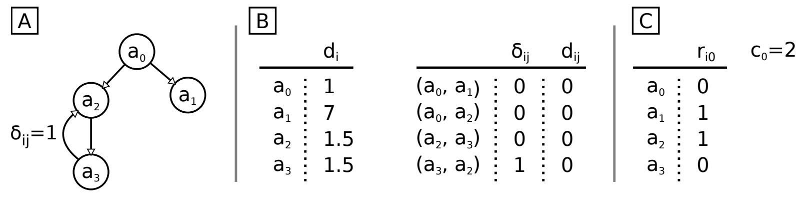
Abstract. Resource constrained cyclic scheduling problems consist in planning the execution over limited resources of a set of activities, *to be indefinitely repeated*. In such a context, the iteration period (i.e. the difference between the completion time of consecutive iterations) naturally replaces the makespan as a quality measure; exploiting inter-iteration overlapping is the primary method to obtain high quality schedules. Classical approaches for cyclic scheduling rely on the fact that, by fixing the iteration period, the problem admits an integer linear model. The optimal solution is then usually obtained iteratively, via linear or binary search on the possible iteration period values. In this paper we follow an alternative approach and provide a port of the key Precedence Constraint Posting ideas in a cyclic scheduling context; the value of the iteration period is not a-priori fixed, but results from conflict resolution decisions. A heuristic search method based on Iterative Flattening is used as a practical demonstrator; this was tested over instances from an industrial problem obtaining encouraging results.

Keywords: Precedence Constraint Posting, Resource Constrained Scheduling, Cyclic Scheduling

1 Introduction

Cyclic scheduling problems arise in a wide variety of real-life contexts, when a set of activities have to be repeated an very large, possibly unknown number of times. In this case a schedule cannot be represented by enumerating start times of all activities, as this would require a huge amount of storage, and a convenient abstraction is to assume that activities will execute an infinite number of times *following a periodic schedule*. In this context the notion of optimality is related to the period of the schedule. A minimal period corresponds to the highest number of activities carried out in average over a large time window.

One of the most successful approaches for solving non cyclic resource constrained scheduling problems is Precedence Constraint Posting (PCP, [23, 5, 19]) that basically adds precedence constraints to the project graph to avoid resource over-usage. In this paper, we explore how to apply the PCP solution method to the cyclic scheduling problems. Porting the key PCP concepts to the cyclic



	d_i
a_0	1
a_1	7
a_2	1.5
a_3	1.5

	δ_{ij}	d_{ij}
(a_0, a_1)	0	0
(a_0, a_2)	0	0
(a_2, a_3)	0	0
(a_3, a_2)	1	0

	r_{i0}
a_0	0
a_1	1
a_2	1
a_3	0

Fig. 1. A simple Project Graph and related data

scheduling domain is however a non-trivial task and requires a number of generalizations. We provide basic definitions concerning resource profiles, Minimal Critical Sets (MCS) and Resolvers that apply to the cyclic case.

As a second contribution of the paper, we implement a PCP solver by porting to the cyclic domain the Iterative Flattening method [4, 3]; this is a heuristic algorithm that repeatedly attempts to build improving solutions by iteratively selecting a MCS and resolving the conflict through the addition of a precedence constraint. We have experimented the proposed approach on a set of industrial benchmarks taken from the field of loop scheduling problems extracted by a compiler. The results are very promising (despite the simplicity of this first implementation) and open perspectives for future improvements.

2 Problem description

Cyclic Scheduling can be considered as an extension of classical Resource Constrained Project Scheduling (RCPSP), as it involves deciding the execution order of a set of activities subject to temporal and resource constraints; unlike in the RCPSP, however, the set of activities is *indefinitely repeated over time*. The regularity of the process flow allows one to partially overlap consecutive executions of the activity set, allowing better resource utilization.

Formally, the problem can be defined over a directed graph (referred to as Project Graph) $\mathcal{G} = \langle A, E \rangle$, where A is the set of activities a_i and E is a set of arcs (a_i, a_j), representing temporal dependencies. We refer as *iteration* to a full execution of the graph; the *makespan* is the time span between the start of the first activity and the end of the last one in a single iteration; the *period* is the difference between the start time of the same task in consecutive iterations; the *completion frequency* is the inverse of the period. Figure 1A shows an example of a Project Graph.

We assume activities are non-interruptible and have fixed duration d_i. Similarly, a minimum time lag d_{ij} is associated to each arc and constrains the minimum time distance between the end of a_i and the beginning of a_j *in specific iterations* (say iteration k and h); in particular a parameter δ_{ij} known as *height* or *delay* specifies the iteration offset (i.e. the difference $h - k$). Formally, let s_i^k be the start time of a_i in iteration k, then an arc (a_i, a_j) enforces:

$$s_j^{k+\delta_{ij}} \geq s_i^k + d_i + d_{ij} \tag{1}$$

it is convenient to think about the precedence constraint and the delay as iteration k of a_i "providing input" to iteration $k + \delta_{ij}$ of activity a_j. Unlike many approaches, we assume $\delta_{ij} \in \mathbb{Z}$, rather than $\in \mathbb{N}$. Figure 1B reports durations, delay and time lag values for the Project Graph on the left; note all d_{ij} are 0 for sake of simplicity.

The presence of delay allows cycles to be consistently defined in the project graph, since some of the precedence constraints along the circular path will target activities from different iterations; in general, any cycle with a *strictly positive* sum of arc delay admits a feasible assignment of start times. As an example, the graph from Figure 1 contains a single cycle (involving nodes a_2 and a_3), with total delay equal to 1 (as δ_{32}); if no delay was specified, there would be no chance to satisfy the cyclic dependence. Note that cyclic scheduling enables cycle treatment, but does not require the graph to actually contain cycles.

Each activity a_i requires some non-negative amount r_{ih} of resource r_h from a set R (see Figure 1C). Each resource has limited capacity c_h, so that the total consumption due to activities *from any iteration* cannot exceed c_h at any point of time. Formally:

$$\sum_{i,k\ :\ s_i^k \leq t < s_i^k + d_i} r_{ih} \leq c_h \qquad \forall \text{ time instant } t,\ \forall r_h \in R \tag{2}$$

The problem objective is to minimize the average period; given a reference activity a_i, this can be formally defined as:

$$\lambda_i = \lim_{k \to \infty} \frac{\sum_{h=1}^{k} s_i^h - s_i^{h-1}}{k} = \lim_{k \to \infty} \frac{s_i^k}{k} \tag{3}$$

If no activity is unnecessarily delayed, the choice of a_i does not influence the result [13], so that the period has no dependence on i; hence, we always write λ in the followings. Each cycle L in the graph sets a lower bound on the average period; this is given by the so-called *cycle ratio*, i.e. the total cycle length over the total delay. Therefore, the Maximum Cycle Ratio MCR (or *iteration bound* [22]) of the graph provides a inherent bound on the minimum achievable period; let this be λ^*:

$$\lambda^* = \max_{L \in \mathcal{G}} \frac{\sum_{(a_i,a_j) \in L} d_i + d_{ij}}{\sum_{(a_i,a_j) \in L} \delta_{ij}} \tag{4}$$

the MCR value can be computed in polynomial time by means of specialized algorithms [7, 12]; for a graph with n nodes and m arcs the typical runtime is $O(nm \log n)$ or worse. The λ^* value for the graph in Figure 1 is given by the only cycle contained in the graph: $\lambda^* = (d_2 + d_{2,3} + d_3 + d_{3,2})/(\delta_{2,3} + \delta_{3,2}) = 3$.

2.1 Cyclic and Periodic Scheduling

A solution to a cyclic scheduling problem (i.e. a *schedule*) is an assignment of start times to each activity a_i in each iteration k; as a consequence, a schedule has in principle infinite size. A cyclic schedule is periodic if the start times follow a static pattern, repeated with a fixed period; in such case s_i^k can be rewritten as follows:

$$s_i^k = s_i^0 + k \cdot \lambda \tag{5}$$

where λ has the meaning from Equation (3). In this section we disregard resource constraints (they will be considered later on); under such assumption, periodic schedules have two fundamental properties: 1) there exists a feasible schedule if and only if there exists a periodic schedule [21, 14]; 2) the periodic schedule with the minimum period has $\lambda = \lambda^*$ and is therefore optimal [6]. From now on, we focus on periodic schedules and on feasible assignments for the start times at iteration 0; for sake of simplicity, *the notation s_i will be used in place of s_i^0.*

The problem of finding an optimal periodic schedule satisfying all *temporal* constraints can be formalized as follows:

$$\begin{array}{lrll}
\text{(P0)} & \min & \lambda & \\
& \text{subject to:} & s_j + \delta_{ij}\lambda \geq s_i + d_i + d_{ij} & \forall (a_i, a_j) \in E \qquad (6) \\
& & s_i \geq 0 & \forall a_i \in A \\
& & \lambda \geq 0 &
\end{array}$$

where Constraints (6) are derived by combining Equation (1) and (5). Note that P0 is a linear program and could be solved in principle via any general LP technique. It is however more common (and efficient) to use dedicated MCR computation algorithms (see [16, 12, 7]). Solving problem P0 provides no consistency guarantee on resource constraints; as a matter of fact, handling resource constraints is usually the toughest issue in any cyclic scheduling method.

3 Related work

The cyclic scheduling literature mainly arises from industrial and computing contexts. The former includes mass production, chemical and hoist scheduling problems, the latter includes parallel processing, software pipelining and data-flow mapping problems in embedded systems. 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 ([10] is one of the few approaches). A subclass of cyclic scheduling (targeted by most of the existing approaches) is the so-called modulo scheduling [20], where the start times and the period λ are required to assume integer values (this is not the case for our method).

Several heuristic and complete approaches have been proposed for cyclic scheduling. A heuristic algorithm called *iterative modulo scheduling* is proposed in [25] and generates near-optimal schedules. An interesting heuristic approach (*SCAN*) based on a time-indexed ILP model is presented in [2]. Both methods compute a schedule for a single iteration, which is characterized in terms of its makespan and period. The makespan dictates the size of the model, so that schedules with a relatively small period could be difficult to compute due to a possibly high makespan. Our approach is not time indexed and does not suffer from this issue.

Advanced complete formulations are proposed in [11] by Eichenberger and in [8] by Dupont de Dinechin; both the 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 [14, 9].

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. In this paper, we provide an alternative method which does not require the iterative solution of NP-hard subproblems.

4 Cyclic Precedence Constraint Posting

Precedence Constraint Posting (PCP, see [23, 5]) is a scheme for solving scheduling problems with limited resources. Roughly speaking, the method relies on a *constraint model* to ensure temporal consistency, while resource constraints are tackled by iterative 1) identification of possible conflicts and 2) their resolution through the addition of temporal constraints, so that resource constraints are progressively turned into temporal constraints. A solution is provided in the form of a conflict free, temporal consistent *augmented graph*. The main idea roots back to [18, 17] from the stochastic scheduling domain and was successfully applied in an AI context in [4, 3, 24, 19].

The method is very appealing in a cyclic context, where an optimal solution can be computed in polynomial time (although less efficiently that in non-cyclic scheduling) if no resource constraint is taken into account. This section describes how the key PCP concepts can be generalized to a cyclic scheduling context.

4.1 Concurrency and requirement functions

A resource conflict arises when the cumulative usage of overlapping tasks exceeds the capacity; in cyclic scheduling, computing the resource usage requires to take into account overlapping iterations. Figure 2A shows a periodic schedule for the graph from Figure 1; as one can observe, not only activities from different iterations may overlap, but different iterations of the same activity may be running at the same time instant. In the figure, the schedule *period* is 3, while the *makespan* is marked by the end of a_1 and has value 8.

We devise a formal characterization of the resource usage of an activity, by restricting focus to an arbitrary λ length time span $[k \cdot \lambda, (k+1) \cdot \lambda[$, with k sufficiently large for the schedule to have entered a fully periodic behavior:

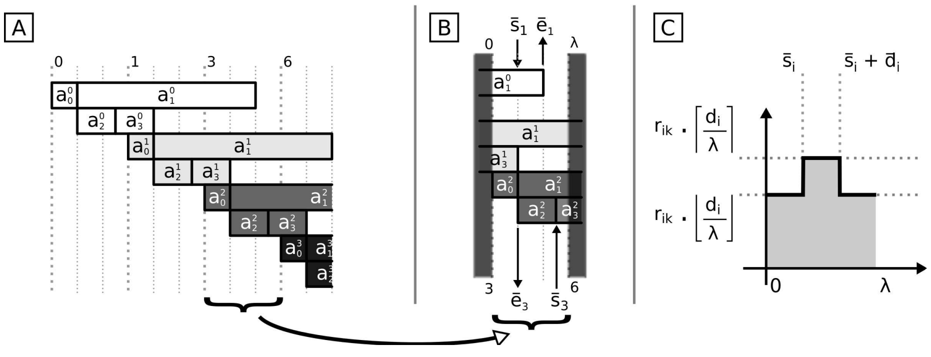


Fig. 2. A) a periodic schedule; B) a periodic schedule on the modular interval; C) modular requirement function

this always happens when $(k + 1) \cdot \lambda$ is greater than the schedule makespan (see Figure 2B). Due to periodicity, the interval is representative of the steady schedule behavior; in the followings, it is referred to as *modular interval* $[0, \lambda[$. A local start time within the modular interval can be associated to each activity via decomposition; namely, we can write:

$$s_i = \bar{s}_i + \beta_i \cdot \lambda \tag{7}$$

with $\bar{s}_i$ in $[0, \lambda[$ and $\beta_i \in \mathbb{N}$; equivalently, we have $\bar{s}_i = \{s_i/\lambda\} \cdot \lambda$, where the notation $\{\cdot\}$ denotes the fractional part; in the example from Figure 2B, we have $\bar{s}_0 = 0, \bar{s}_1 = \bar{s}_2 = 1, \bar{s}_3 = 2.5$. In the followings, we refer to $\bar{s}_i$ as *modular start time* of a_i, because of the analogy between the start time decomposition and the modulo operation over $\mathbb{Z}$. Since s_i represents the start time of iteration 0 of a_i, the β_i value in Equation (7) refers to the number of full periods elapsed before a_i is scheduled for the first time; in the schedule from Figure 2 all β_i are zero, since all s_i are in the interval $[0, \lambda[$. Similarly, we can define a modular end value:

$$\bar{e}_i = \left\{ \frac{s_i + d_i}{\lambda} \right\} \cdot \lambda \quad \text{or, equivalently: } s_i + d_i = \bar{e}_i + \eta_i \cdot \lambda \tag{8}$$

where η_i has a similar meaning to β_i; in Figure 2 we have $\eta_1 = 2, \eta_3 = 1$, while all other η_i values are 0. Observe that $\bar{e}_i$ can be higher *or lower* than $\bar{s}_i$, as one can see in Figure 2.

The number of concurrent executions of each activity within a period can be characterized by relying on modular start/end values; in particular, in the interval $[0, \lambda[$, at least $\lfloor d_i/\lambda \rfloor$ iterations of activity a_i are *always* executing. An extra iteration should be added if the time instant under analysis falls between the modular start and the modular end. Formally, we introduce a concurrency function $\#a_i(t)$ with values in the time interval $[0, \lambda[$ and such that:

$$\#a_i(t) = \left\lfloor \frac{d_i}{\lambda} \right\rfloor + pulse(a_i, t) \tag{9}$$

and:

$$pulse(a_i, t) = \begin{cases} 1 & \text{if } \bar{s}_i \leq t < \bar{e}_i \text{ or } t < \bar{e}_i < \bar{s}_i \text{ or } \bar{e}_i < \bar{s}_i \leq t \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

Function $\#a_i(t)$ ranges in between $\lfloor d_i/\lambda \rfloor$ and $\lceil d_i/\lambda \rceil$ and is therefore constant if d_i is an integer multiple of λ. Equation 9 shows how the concurrency function results from the superimposition of a constant and a *pulse* function; the latter is triggered by a new arrival of a_i and makes a step down when a *previous* iteration of a_i ceases execution. The concurrency function allows a *modular requirement function* $\bar{r}_{ik}(t)$ to be easily defined as $\bar{r}_{ik}(t) = r_{ik} \cdot \#a_i(t)$. Figure 2C provides a pictorial representation of the requirement function. Note that for λ sufficiently large, $\lfloor d_i/\lambda \rfloor$ is 0, $\bar{s}_i = s_i$, $\bar{e}_i = s_i + d_i$ and $\bar{r}_{ik}(t)$ boils down to the rectangular function used in classical scheduling.

4.2 Minimal Critical Sets

In non-cyclic scheduling a Critical Set (CS) is a subset S of activities, collectively overusing a resource r_h in case of simultaneous execution. A Minimal Critical Set (MCS) is CS such that none of its proper subsets is a CS. A precedence constraint posted over a pair of activities a_i, a_j in the same MCS prevents the conflict: hence, by focusing on MCS, one can avoid adding useless precedence constraints. In a cyclic context, several iterations of the same activity may be running concurrently; therefore, we propose to define an MCS as a *multiset*, i.e. a set where objects can appear several times.

Definition 1 (Generalized Critical Set and Minimal Critical Set). *A Generalized Critical Set (GCS) is a multiset M of activities in A, such that, given the current temporal constraints:*

1. *$card(a_i, M)$ iterations of each activity a_i may overlap*
2. *the set of overlapping executions causes an over-usage:*
   $\exists r_h \mid \sum_{a_i \in M} r_{ih} \cdot card(a_i, M) > c_h$

A Generalized Minimal Critical Set is a multiset M such that none of its proper sub-multisets is a GCS.

where $card(a_i, M)$ gives the cardinality of a_i in M; we write $a_i \in M$ iff $card(a_i, M) > 0$; we write $M' \subseteq M$ iff, $\forall a_i \in M'$, it holds $card(a_i, M') \leq card(a_i, M)$. In the followings, we always use the term Critical Set and Minimal Critical Set in the generalized sense. With reference to Figure 2, $\{a_1, a_1, a_1, a_2\}$ is a CS, while $\{a_1, a_1, a_2\}$ or $\{a_1, a_1, a_1\}$ are MCS; note that a MCS may even consist of several iterations of a single activity.

4.3 MCS Resolvers

In a PCP approach, critical sets are cleared by adding carefully designed constraints, referred to as *resolvers*. Specifically, we adopt the following definition, holding for both classical and generalized MCS:

Definition 2 (Resolver). *Let M be an MCS; a* resolver *for M is any temporal constraint ρ which prevents the multiset from satisfying Definition 1.*

A simple pairwise precedence constraints (i.e. added arc) is an example of valid resolver in non-cyclic scheduling; other types of constraints could be employed as well [15]. A good resolver should allow to efficiently test consistency after it is added to the temporal model: this is the reason why simple precedence constraints are by far the most common resolvers in the literature. In second place, it should be possible to encode an optimal solution as a set of resolvers, so as not to loose the chance of achieving optimality.

By direct application of Definition 2 we get a necessary and sufficient condition, defining a resolver in the most general form:

$$\forall t \in [0, \lambda[: \bigvee_{a_i \in M} [\#a_i(t) < card(a_i, M)] \tag{11}$$

which basically states that for each time instant t in the modular interval, at least one of the concurrency functions for activities in the MCS must be strictly less then the cardinality in M. Equivalently:

$$\forall t \in [0, \lambda[: \sum_{a_i \in M} \#a_i(t) < \sum_{a_i \in M} card(a_i, M) \tag{12}$$

Unfortunately, Constraint 12 does not allow an efficient consistency check, hence a suitable decomposition should be devised. Observe that:

Statement 1 *Let L be a cycle; let us refer as A_L to the set of activities it includes and as Δ_L to its total delay, then L does not allow more than Δ_L iterations of activities in A_L to run concurrently; the same restriction cannot be achieved by employing fewer arcs or a larger delay.*

A formal proof is omitted due to lack of space, but the statement follows quite intuitively from the semantic of arc delays described in Section 2. Now:

Theorem 1. *Given a feasible schedule according to Constraint* (12)*, it is possible to identify a collection $\mathcal{L}$ of unary and binary loops L such that any feasible schedule modification according to $\mathcal{L}$ is feasible according to Constraint* (12)*.*

Proof. It is sufficient to prove that any infeasible schedule modification according to Constraint (12) is infeasible according to precedence constraints in $\mathcal{L}$. For each activity a_i, let us introduce a self-loop with delay $\delta_{ii} = \lceil d_i/\lambda \rceil$; then, for each pair of activities a_i, a_j such that a_i precedes a_j in the modular interval (i.e. $\bar{e}_i \leq \bar{s}_j$), we introduce a binary cycle with $\delta_{ij} = \beta_j - \eta_i$ and $\delta_{ji} = \beta_i - \eta_j + 1$; let the collection of those cycles be $\mathcal{L}$. By construction, all precedence constraints enforced by $\mathcal{L}$ are satisfied. In order to violate Constraint (12), any modification of the target schedule should either increase the amount of overlapping executions for a single activity a_i or break a modular precedence relation; due to Statement 1, this would respectively result in the violation of a unary or binary loop constraint in $\mathcal{L}$. □

Basically, Theorem 1 shows that a properly designed binary or unary loop "covers" part of the feasible space of Constraint (12). By iterating the process for the infinitely many possible valid schedules according to (12), we obtain:

Theorem 2. *Constraint* (12) *is equivalent to a infinite-size disjunction of unary and binary cycles defined over activities in M.*

Hence, *we can focus on resolvers formulated as unary or binary cycles* without loosing the chance of achieving optimality; moreover, adding a cycle to the Project Graph still allows to efficiently compute the best achievable period.

Unary resolver: A unary resolver L over a_i with $\Delta_L < card(a_i, M)$ delay can be built by adding a self-arc (a_i, a_i) with $\delta_{ii} = card(a_i, M) - 1$; any smaller δ_{ii} value would unnecessarily over-constrain the partial solution.

Binary resolver: Building a binary resolver over two activities a_i and a_j requires adding arcs (a_i, a_j), (a_j, a_i) and deciding their delay value so that:

$$\delta_{ij} + \delta_{ji} = card(a_i, M) + card(a_j, M) - 1 \qquad (13)$$

where we recall $\delta_{ij}, \delta_{ji} \in \mathbb{Z}$. This means that *binary cycle resolvers are parametric and an infinite number of possible resolvers can be defined between a pair of activities.* In this work we use a simple heuristic to decide the delay distribution; a deeper investigation of the parameter space is left for future work.

5 Implementing the Cyclic PCP

5.1 Cyclic Iterative Flattening

As a practical demonstrator for the framework proposed, we realized a heuristic PCP solver for period minimization in resource constrained cyclic scheduling. In particular, our solver is a porting of the Iterative Flattening method to cyclic scheduling.

Iterative Flattening (iFlat) is a heuristic solution method for makespan minimization on the RCPSP. The approach is introduced in [4, 3] and makes use of a Temporal Constraint Network to enforce consistency of precedence and time window constraints. In the basic version, iFlat performs a fixed number of iterations; during each of them, a solution is tentatively built by repeatedly 1) selecting a MCS and 2) resolving the conflict through the addition of a precedence constraint; no backtracking is performed. The MCS and resolver selection is randomized so that each iteration explores a different portion of the search space. MCS identification is done [24, 23] either by computing so-called resource usage envelopes, or by scanning the contention peaks arising in the earliest start time schedule (computed by disregarding resource constraints).

The main scheme of our approach matches that of basic iFlat and is reported in Algorithm 1; as one can see, we adopt the peak based MCS identification procedure. Unlike the original iFlat, the presented approach lacks support for time windows, deadlines and maximum time lag constraints, which is planned for future research. In the following, each step will be further detailed.

Algorithm 1 (Cyclic) Iterative Flattening

Input: a problem instance defined over a graph $\mathcal{G} = \langle A, E \rangle$; a number of iterations n
Output: a conflict free augmented graph

```
1: for i = 0 to n-1 do
2:    while at least an MCS exists do
3:       compute contention peaks and sample an MCS
4:       sample a resolver for the MCS
5:       add the resolver and check temporal consistency
6:    if the current $\lambda^*$ improves the best one then
7:       store the current solution and set current $\lambda^*$ as threshold
8: return the best solution
```

5.2 Temporal Consistency Check

Since we do not consider time windows or deadline constraint on specific activities, checking temporal consistency amounts to compute an optimal (resource unaware) periodic schedule and test whether the resulting period λ^* is less than the current threshold. We recall that an optimal schedule is obtained by solving problem P0 from Section 2.1 via MCR computation.

Our MCR algorithm is a variant of the CYCLE class, as described in [12]. The key idea is that, for any fixed λ^0, P0 becomes a longest path computation problem on a cyclic graph; in detail, each arc (a_i, a_j) has a modified length equal to $d_i + d_{ij} - \delta_{ij} \cdot \lambda^0$. At the end of the computation, the longest path to a_i defines the activity earliest start value (i.e. s_i). By starting from a guess value $\lambda^0 \leq \lambda^*$ (i.e. super-optimal) the longest path computation either proves feasibility (hence we deduce $\lambda^0 = \lambda^*$) or identifies an infeasible cycle; in this case, the cycle provides a new lower bound $\lambda^1 > \lambda^0$, which can be used as a new guess to reiterate the process.

Some heuristics can be employed to find a good initial period guess. Here, however, we simply start with $\lambda^0 = 0$; whenever a resolver is added and a new schedule needs to be computed (step 5 from Algorithm 1), we use the λ^* of the current graph to prime the process, making the computation incremental. This is a peculiar advantage of the CYCLE algorithm class, which could not be exploited with the (otherwise faster) algorithm by Young-Tarjan-Orlin [26].

The longest path computation can be performed via Bellmann-Ford like algorithms[1]; specifically, the method we use is reported in Algorithm 2 and is designed to avoid repeated traversals of graph cycles. The procedure keeps a set of nodes to be visited (referred to as Q); for each activity a_i, the algorithm also stores the set $V(a_i)$ of tasks on the critical path to a_i. The output is an assignment of start times s_i and a period bound λ^1 (in case of feasibility) or the bound alone (in case of infeasibility). Algorithm 2 is initialized by setting all start times to 0; the corresponding critical paths contain no node and have 0 delay (line 1); all tasks are enqueued to be processed ($Q \leftarrow T$).

[1] The simpler Dijkstra algorithm is not an option due to the presence of cycles.

Algorithm 2 Find Start Times (longest paths)

Input: the current augmented graph and λ guess λ^0
Output: a lower bound on λ^* and an assignment of start times
Data structures:
- $\forall a_i \in A$: $V(a_i)$ is the set of nodes on the critical path
- $\forall a_i \in A$: s_i is the start time of iteration 0
- $\forall a_i \in A$: δ_i is the delay of the current critical path to a_i
- Q is the set of activities to be visited
- λ^1 is the period lower bound being computed

```
 1: $\forall a_i \in A$: $V(a_i) \leftarrow \emptyset$, $s_i \leftarrow 0$, $\delta_i \leftarrow 0$
 2: $Q \leftarrow A$, $\lambda^1 \leftarrow 0$
 3: while $Q \neq \emptyset$ do
 4:    pick an arbitrary activity $a_i$ from $Q$
 5:    $Q \leftarrow Q \setminus \{a_i\}$, $V(a_i) \leftarrow V(a_i) \cup \{a_i\}$
 6:    for all arcs $(a_i, a_j) \in E$ having $a_i$ as source do
 7:       if $a_j$ is not in $V(a_i)$ then
 8:          if $s_j < s_i + d_i + d_{ij} - \delta_{ij} \cdot \lambda^0$ then //$a_i$ is on the critical path to $a_j$
 9:             $s_j \leftarrow s_i + d_i + d_{ij} - \delta_{ij} \cdot \lambda^0$ //update the critical path to $a_j$
10:             $\delta_j \leftarrow \delta_i + \delta_{ij}$ //update total delay to $a_j$
11:             $V(a_j) \leftarrow V(a_i)$ //transfer set of visited nodes
12:             $Q \leftarrow Q \cup \{a_j\}$ //enqueue $a_j$
13:       else //a cycle $L$ has been identified
14:          let $\Delta_L := \delta_i - \delta_j + \delta_{ij}$ //total delay for cycle $L$
15:          let $D_L := s_i + d_i + d_{ij} - s_j + (\delta_i - \delta_j) \cdot \lambda^0$ //total length of cycle $L$
16:          $\lambda^1 \leftarrow \max\left(\lambda^1, D_L/\Delta_L\right)$ //update bound with cycle ration of $L$
17: if $\lambda^1 > \lambda^0$ then
18:    return the bound $\lambda^1$ and no start time
19: else
20:    return all the $s_i$ and the bound $\lambda^1$
```

At each step an arbitrary task is picked from the Q set (line 5); the choice does not compromise correctness; here, we simply pick the lowest index activity in Q. Then, a_i is marked as part of longest path to itself: this is needed to deal with self-loops in the Project Graph.

Next, all the successors a_j of task a_i are processed; if the successor is not part of the longest path to a_i (line 7) and if arc (a_i, a_j) sets a stronger bound on the start time s_j (line 8), then: 1) s_j is updated, 2) the total delay δ_j is updated and 3) the longest path to a_j becomes $V(a_i)$; finally, a_j is marked as an activity to be processed (at line 12). If a_j is found to be part of the longest path to a_i (line 13), a loop L has been identified; in such a case: 1) either the constraint corresponding to arc (a_i, a_j) is feasible (and the current λ^0 guess is confirmed); or 2) the constraint is infeasible, and we can determine the minimum λ which would provide feasibility.

This is done by computing the Cycle Ratio of the identified loop; in particular, the total duration D_L of activities and arcs in the loop can be computed as shown at line 15; the total delay on the loop is referred to as Δ_L, hence the cycle ratio is D_L/Δ_L (line 16) and the bound λ^1 is updated. If at the end of the process (when Q becomes empty) λ^1 is higher than the current throughput

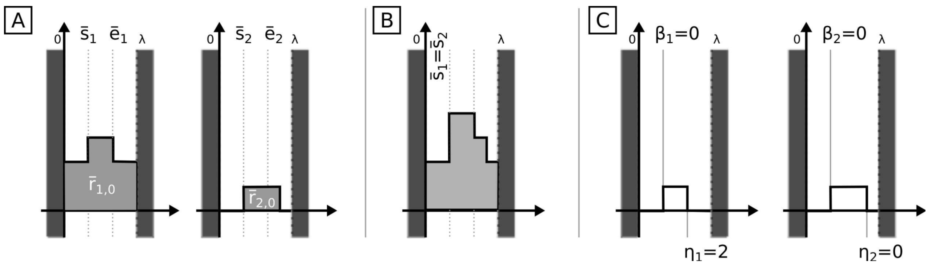


Fig. 3. A,B) effect of combining different usage functions; C) activity pulses

guess λ^0 an infeasibility has been detected and the algorithm repeats (line 18). Otherwise, all nodes have been assigned a feasible start time and a valid lower bound on the period has been computed. Minor modifications are done in the actual implementation to improve the runtime; as for any other CYCLE algorithm, known bounds on the asymptotic complexity are very loose.

5.3 Detecting Contention Peaks and MCS

Here, we define a contention peak as a maximal set of overlapping iterations of activities *in the current schedule*, collectively overusing a resource r_k. Each contention peak is a multiset and corresponds to a (Generalized) CS occurring in a specific schedule. In our algorithm, we choose the (Generalized) MCS to be resolved by: 1) identification of all the contention peaks for each resource r_k; 2) extraction of an MCS from each peak by randomly reducing the cardinality of activities in the multi-set, until minimality is met; 3) choosing an MCS from the whole pool uniformly at random. Of course some clever heuristics may be used to bias the choice towards promising MCS: this is left for future research.

Our peak detection procedure (in Algorithm 3) is an extension of the one presented in [23] for non-cyclic schedules; the method operates on the modular interval $[0, \lambda[$ and is designed to take into account the peculiar profile of the modular requirement function $\bar{r}_{ik}(t)$, consisting of a constant and a *pulse* component (see Equation 9 in Section 4.1). The main idea is to focus on the maximal overlapping of the *pulse* functions; in Figure 3 one can check how the individual modular requirement functions $\bar{r}_{1,0}$ and $\bar{r}_{2,0}$ contribute to the usage profile of resource r_0 in the modular interval $[0, \lambda[$ in our example problem. Peaks in the resource usage are due to the *pulse* components; specifically, the schedule has a single maximal peak at time 1.

The peak detection algorithm processes activities by increasing modular start time (i.e. pulse start time) and keeps track via a set X of pulses currently in execution. New pulses are added to X when they start (i.e. $\bar{s}_i = t$, line 4) and removed from X when they are over (i.e. $\bar{e}_i \leq t$). The X set is initialized before the main loop so as to include pulses that cross the λ boundary (i.e. such that $\bar{e}_i < \bar{s}_i$, in line 2). A set Rm is used to prevent the crossing activities from being removed twice from X. A peak is collected whenever some activity needs

Algorithm 3 (Cyclic) Peak Detection

Input: a schedule and a target resource r_k
Output: a list of detected peaks
Data structures:
- Q: vector of activities requiring r_k
- X: set of executing activities in execution at the current time instant
- Rm: set of activities removed from X

```
 1: sort activities in $Q$ by increasing modular start time $\bar{s}_i$
 2: $X$ = set of activities in $Q$ such that $\bar{e}_i < \bar{s}_i$
 3: for i = 0 to $|Q|$ do
 4:     if $i < |Q|$ then $a_i$ = i-th activity in $Q$; current time $t = \bar{s}_i$
 5:     else current time $t = \lambda$
 6:     if there is any activity $a_j \in X$ such that $\bar{e}_j \leq t$ and $a_j \notin Rm$ then
 7:         if $\sum_{a_j \in Q} \bar{r}_{jk}(t) > c_k$ then
 8:             build a peak with all activities $a_j$ in $Q$, each with cardinality $\#a_j(t)$
 9:     remove from $X$ all activities $a_j$ such that $\bar{e}_j \leq t$ and $a_j \notin Rm$
10:     add to $Rm$ all the removed activities
11:     if $i < |Q|$ then add $a_i$ to $X$
12: return the list of detected peaks
```

to be removed from X; in such a case, the peak contains each activity with the cardinality specified by the value of the concurrency function $\#a_i(t)$. The final step of the algorithm always processes time λ and collects the last peak.

5.4 Sampling and Adding a Resolver

Once an MCS M is selected, we sample a binary or unary resolver uniformly at random. The original iFlat algorithm can exploit temporal constraint propagation to identify trivially infeasible resolvers; since our temporal model does not provide any actual propagation mechanism (i.e. there is no time window for any activity), the sampled resolver is tested with one-step look ahead: in case the λ^* value of the resulting graph is higher then the current threshold, the resolver is discarded and a new one is sampled. In the worst case, the process stops when there are no more resolvers and the current algorithm iteration ends.

As described in Section 4.3, cyclic resolvers are either unary or binary cycles. While the delay distribution for a unary resolver is fixed; binary resolvers have parametric delay. Here, we use a heuristic to deterministically choose a distribution, given an ordered pair of activities a_i, a_j. The main underlying idea is that a binary cycle prevents the two involved activities a_i, a_j from simultaneously reaching a maximum concurrency; intuitively, this means to prevent the *pulse* components from the corresponding $\#a_i(t)$ function from overlapping. This can be done by either forcing $\bar{e}_i$ to "precede" $\bar{s}_i$, or vice-versa.

We recall the end of a_i pulse is associated to the iteration number η_i, while the beginning of a_j pulse to the iteration number β_j. In our approach, when posting a resolver on the ordered pair a_i, a_j we always heuristically set the delay of the added arc (a_i, a_j) to:

$$\delta_{ij} = \beta_j - \eta_i \tag{14}$$

intuitively, this amounts to force iteration β_j of a_j to wait for the end of iteration η_i of a_i. The δ_{ji} delay is fixed according to Equation (13). Figure 3D shows the β and η values for (the pulses of) a_1 and a_2; their computation is done as described in Section 4.1. When posting a resolver between a_1 and a_2 for the MCS $\{a_1, a_1, a_2\}$, our heuristics would set δ_{ij} to $0 - 2 = -2$ and δ_{ji} to $2 - \delta_{ij} = 4$.

5.5 Experimental Results

The cyclic iFlat approach has been implemented in C++ and tested on a benchmark consisting of industrial problems from the instruction scheduling domain [8]. In particular, the instances represent loop scheduling problems extracted by the compiler for the ST200 processor, by STMicroelectronics; each activity represent an instruction, requiring one or more CPU components for its execution. All instructions have unary duration; the maximum δ_{ij} on the whole benchmark is 4; the maximum d_{ij} is 3. With the objective to make the benchmark more challenging, in [1] the authors replaced the original resource consumption with a random number, thus providing a modified data set.

The benchmark is employed in [1] to perform a through comparison of ILP based complete approaches; given a large time limit (604800 seconds) the compared solvers found the optimal solution for almost all the instances. Our experiment focus is on assessing how close to the known optima the PCP heuristics can go; to this end, it must be mentioned that in the original problem the start times and the period were required to be integer (i.e. it was actually a modulo scheduling problem). Our approach does not make such an assumption and has in principle the chance to find better solutions; nevertheless, since all durations are unary, the optimal values found in [1] are a pretty reliable estimate of the minimal period values our heuristics could ever achieve.

All experiments are performed on an Intel Core 2 T7200, by running 1200 iFlat iterations for each instance. Table 1 shows the result of the evaluation for both the data sets (industrial and modified). Next to the instance name, the number of activities and arcs is reported. Then, for each data set, the table shows the optimal λ value (for the corresponding modulo scheduling problem), the best λ found by our heuristics within one second and the sequence number of the iteration when such solution was found. The table content is completed by the best overall λ, the iteration and the time when it was found, the total solution time. Some of the instances from the original paper are missing, due to problems during the conversion to our solver input format. For the instances reporting a "—" in the *opt* column an optimal solution was not found by the reference ILP solvers.

A number in bold face highlights the cases where our heuristics hits the actual optimum; in general cyclic iFlat obtains very good results, reaching optimal or close to optimal solutions within 1 second. In a few cases (e.g. gsm-st231.14) the heuristics appears to beat the optimal solver: this is only a consequence of the integrality requirement assumed by the complete approach used as a reference. The modified instances are remarkably more difficult than their counterparts and set tougher challenges to iFlat, which nevertheless obtains pretty good results.

			INDUSTRIAL INSTANCES							MODIFIED INSTANCES						
				1 sec		best					1 sec		best			
inst	acts	arcs	opt*	λ	it	λ	it	time	tot. time	opt*	λ	it	λ	it	time	tot. time
adpcm-st231.1	86	405	21	22.00	13	**21.00**	121	4.17	39.04	—						
adpcm-st231.2	142	722	40	43.00	20	41.00	158	6.38	40.46	—						
gsm-st231.2	101	462	26	27.00	5	**26.00**	98	3.89	47.39	—						
gsm-st231.6	30	130	7	**7.00**	19	7.00	19	0.05	4.02	27	**27.00**	63	27.00	63	0.37	7.26
gsm-st231.7	44	192	11	**11.00**	4	11.00	4	0.03	9.58	41	**41.00**	53	41.00	53	0.88	18.98
gsm-st231.9	34	154	28	**28.00**	1	28.00	1	0.00	0.35	32	**32.00**	59	32.00	59	0.22	3.63
gsm-st231.10	10	42	4	**4.00**	0	4.00	0	0.00	0.26	8	**8.00**	41	8.00	41	0.01	0.33
gsm-st231.11	26	137	20	**9.00**	0	9.00	0	0.00	0.08	24	**24.00**	4	24.00	4	0.01	4.58
gsm-st231.12	15	70	8	**8.00**	40	8.00	40	0.02	0.63	13	**13.00**	65	13.00	65	0.03	0.63
gsm-st231.13	46	210	19	**19.00**	0	19.00	0	0.00	0.57	43	**43.00**	24	43.00	24	0.44	22.54
gsm-st231.14	39	176	10	**9.50**	17	9.50	17	0.12	9.21	33	36.00	17	35.00	774	9.96	15.42
gsm-st231.15	15	70	8	**8.00**	3	8.00	3	0.00	0.63	12	**12.00**	585	12.00	585	0.28	0.62
gsm-st231.16	65	323	16	**16.00**	15	16.00	15	0.52	36.48	—						
gsm-st231.17	38	173	9	**9.00**	20	9.00	20	0.14	8.21	33	34.00	3	**33.00**	262	2.80	12.46
gsm-st231.19	19	86	8	**8.00**	14	8.00	14	0.01	0.87	15	**15.00**	149	15.00	149	0.25	1.79
gsm-st231.20	23	102	6	**5.33**	186	5.33	186	0.52	3.35	20	**20.00**	48	20.00	48	0.17	4.06
gsm-st231.21	33	154	18	**18.00**	1	18.00	1	0.00	0.33	30	**29.00**	211	29.00	211	0.72	3.92
gsm-st231.22	31	146	18	**18.00**	1	18.00	1	0.00	0.30	29	**29.00**	36	29.00	36	0.15	3.10
gsm-st231.25	60	273	16	**16.00**	38	16.00	38	0.42	13.12	—						
gsm-st231.29	44	192	11	**11.00**	4	11.00	4	0.03	9.45	42	**42.00**	6	42.00	6	0.17	19.26
gsm-st231.30	30	130	7	**7.00**	19	7.00	19	0.05	4.02	25	26.00	0	**25.00**	461	2.88	7.52
gsm-st231.31	44	192	11	**11.00**	4	11.00	4	0.02	9.58	39	41.00	14	40.00	284	4.66	18.94
gsm-st231.32	32	138	15	**15.00**	0	15.00	0	0.00	0.10	30	**30.00**	13	30.00	13	0.05	8.11
gsm-st231.33	59	266	15	**15.00**	11	14.50	324	3.06	11.25	—						
gsm-st231.34	10	42	4	**4.00**	2	4.00	2	0.00	0.25	7	**7.00**	82	7.00	82	0.03	0.32
gsm-st231.35	18	80	6	**6.00**	13	6.00	13	0.01	0.67	14	15.00	33	**14.00**	1109	1.07	1.15
gsm-st231.36	31	143	10	**10.00**	28	10.00	28	0.05	1.99	24	27.00	8	26.00	757	3.56	5.77
gsm-st231.39	26	118	8	**8.00**	10	8.00	10	0.01	1.84	21	22.00	105	22.00	105	0.43	4.79
gsm-st231.40	21	103	10	**10.00**	17	10.00	17	0.02	1.13	17	**17.00**	32	17.00	32	0.06	2.00
gsm-st231.41	60	315	18	20.00	54	18.00	139	2.06	17.75	—						
gsm-st231.42	23	102	6	**5.33**	186	5.33	186	0.51	3.31	18	19.00	109	**18.00**	882	2.65	3.69
gsm-st231.43	26	115	8	9.00	8	9.00	8	0.01	0.16	20	22.00	52	21.00	1153	1.93	2.01

Table 1. Results for the benchmark from [1]; *: the reported optimum is for for the modulo scheduling problem, where the start times and the period are constrained to be integer

The delay distribution heuristics from Section 5.4 seems to be a key factor to get high quality solutions.

6 Conclusion and Future Work

The main contribution of this work is the generalization of the key PCP concepts to the cyclic scheduling domain; in this context, the use of PCP avoids the repeated resolution of NP-hard subproblem, which is a common trait of most state of the art approaches. As a practical demonstrator, we implemented a simple and yet effective cyclic version of the Iterative Flattening heuristics.

Many interesting research directions remain open: the parameter space of binary resolvers should be characterized so as to narrow the range of possible delay distribution choices; resource and temporal propagation techniques for cyclic problems should be defined; effective MCS selection heuristics should be tested and other search schemes investigated.

References

1. Maria Ayala and Christian Artigues. On integer linear programming formulations for the resource-constrained modulo scheduling problem., 2010.
2. Florent Blachot, B. de Dinechin, and G. Huard. Scan: a heuristic for near-optimal software pipelining. *Lecture notes in computer science*, pages 289–298, 2006.
3. A. Cesta, A. Oddi, and S. F. Smith. Iterative Flattening: A Scalable Method for Solving Multi-Capacity Scheduling Problems. In *Proc. of AAAI/IAAI*, pages 742–747, 2000.
4. Amedeo Cesta, Angelo Oddi, and S.F. Smith. Scheduling Multi-Capacitated Resources under Complex Temporal Constraints. *Proc. of CP*, pages 465–465, 1998.
5. Amedeo Cesta, Angelo Oddi, and Stephen F Smith. A Constraint-Based Method for Project Scheduling with Time Windows. *Journal of Heuristics*, 8(1):109–136, 2002.
6. P. Chrétienne. Transient and limiting behavior of timed event graphs. *RAIRO Techniques et Sciences Informatiques*, 4:127–192, 1985.
7. Ali Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic*, 9(4):385–418, October 2004.
8. B.D. de Dinechin. From machine scheduling to VLIW instruction scheduling. *ST Journal of Research*, 1(2):1–35, 2004.
9. Benoit Dupont de Dinechin, C. Artigues, and S. Azem. *Resource-Constrained Modulo Scheduling*, chapter 18. ISTE, London, UK, 2010.
10. 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.
11. A.E. Eichenberger and E.S. Davidson. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices*, 32(5):194–205, 1997.
12. Loukas Georgiadis, Andrew V. Golberg, Robert E. Tarjan, and Renator F. Werneck. An experimental study of minimum mean cycle algorithms. In *Proc. of ALENEX*. Citeseer, 2009.
13. Amir Hossein Ghamarian, Marc Geilen, Sander Stuijk, Twan Basten, Bart D Theelen, Mohammad Reza Mousavi, A J M Moonen, and Marco Bekooij. Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. of ACSD*, pages 25–36, 2006.
14. C. Hanen and A. Munier. *Cyclic scheduling on parallel processors: an overview*, chapter 4. Wiley, 1994.
15. R. Heilmann. A branch-and-bound procedure for the multi-mode resource-constrained project scheduling problem with minimum and maximum time lags. *European Journal of Operational Research*, 144(2):348–365, January 2003.
16. R. A. Howard. *Dynamic Programming and Markov Processes.* Wiley, New York, 1960.
17. G. Igelmund and F. J. Radermacher. Algorithmic approaches to preselective strategies for stochastic scheduling problems. *Networks*, 13(1):29–48, January 1983.
18. G. Igelmund and F. J. Radermacher. Preselective strategies for the optimization of stochastic project networks under resource constraints. *Networks*, 13(1):1–28, January 1983.
19. P. Laborie. Complete MCS-Based Search: Application to Resource Constrained Project Scheduling. In *Proc. of IJCAI*, pages 181–186. Professional Book Center, 2005.
20. M. Lam. Software pipelining: An effective scheduling technique for VLIW machines. In *Proc. of ACM SIGPLAN 1988*, volume 23, pages 318–328. ACM, July 1988.

21. S.T. McCormick and U.S. Rao. Some complexity results in cyclic scheduling. *Mathematical and Computer Modelling*, 20(2):107–122, 1994.
22. K.K. Parhi and D.G. Messerschmitt. Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs. In *Proc. of ISCAS*, number 217, pages 1923–1928. Ieee, 1989.
23. N. Policella, A. Cesta, A. Oddi, and S. F. Smith. From precedence constraint posting to partial order schedules: A CSP approach to Robust Scheduling. *AI Communications*, 20(3):163–180, 2007.
24. N. Policella, S. F. Smith, A. Cesta, and A. Oddi. Generating Robust Schedules through Temporal Flexibility. In *Proc. of ICAPS*, pages 209–218, 2004.
25. B.R. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of MICRO*, pages 63–74. ACM, 1994.
26. Neal Young, Robert Tarjan, and James Orlin. Faster Parametric Shortest Path and Minimum Balance Algorithms. *Networks*, 21:205–221, May 2002.