A constraint based approach to cyclic RCPSP

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

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

Abstract. A cyclic scheduling problem is specified by a set of activities that are executed an infinite number of times subject to precedence and resource constraints. The cyclic scheduling problem has many applications in manufacturing, production systems, embedded systems, compiler design and chemical systems. This paper proposes a Constraint Programming approach based on Modular Arithmetic, taking into account temporal resource constraints. In particular, we propose an original modular precedence constraint along with its filtering algorithm. Classical "modular" approaches that fix the modulus and solve an integer linear sub-problem in a generate-and-test fashion. Conversely, our technique is based on a non-linear model that faces the problem as a whole: the modulus domain bounds are inferred from the activity-related and iteration-related variables. The method has been extensively tested on a number of non-trivial synthetic instances and on a set of realistic industrial instances. Both the time to compute a solution and its quality have been assessed. The method is extremely fast to find close to optimal solutions in a very short time also for large instances. In addition, we have found a solution for one instance that was previously unsolved and improved the bound of another of a factor of 11.5%.

Keywords: Constraint Resource Constrained Cyclic Scheduling

1 Introduction

The cyclic scheduling problem concerns setting times for a set of activities, to be indefinitely repeated, subject to precedence and resource constraints. It can be found in many application areas. For instance, it arises in compiler design implementing loops on parallel architecture, and on data-flow computations in embedded applications. Moreover, cyclic scheduling can be found in mass production, such as cyclic shop or Hoist scheduling problems.

In cyclic scheduling often the notion of optimality is related to the period of the schedule. A minimal period corresponds to the highest number of activities carried out on average over a large time window.

Optimal cyclic schedulers are lately in great demand, as streaming paradigms are gaining momentum across a wide spectrum of computing platforms, ranging from multi-media encoding and decoding in mobile and consumer devices, to advanced packet processing in network appliances, to high-quality rendering in

game consoles. In stream computing, an application can be abstracted as a set of tasks that have to be performed on incoming items (frames) of a data stream. A typical example is video decoding, where a compressed video stream has to be expanded and rendered. As video compression exploits temporal correlation between successive frames, decoding is not pure process-and-forward and computation on the current frame depends on the previously decoded frame. These dependencies must be taken into account in the scheduling model. In embedded computing contexts, resource constraints (computational units and buffer storage) imposed by the underlying hardware platforms are of great importance. In addition, the computational effort which can be spent to compute an optimal schedule is often limited by cost and time-to-market considerations.

In this paper we introduce a Constraint Programming approach based on modular arithmetic for computing minimum-period resource-constrained cyclic schedules. Our main contribution is an original modular precedence constraint and its filtering algorithm. The solver has several interesting characteristics: it deals effectively with temporal and resource constraints, it computes very high quality solutions in a short time, but it can also be pushed to run complete search. An extensive experimental evaluation on a number of non-trivial synthetic instances and on a set of realistic industrial instances (coming from a compiler for a high performance embedded processor) gave promising results compared with state-of-the art ILP-based (Integer Linear Programming) schedulers which perform iterative tightening of the modulus to converge to the optimal period.

2 The Problem

The cyclic scheduling problem is defined on a directed graph $\mathtt{G}(\mathtt{V},\mathtt{A})$ with n ($|\mathtt{V}| = n$) nodes that represent activities with fixed durations d_i, and m ($|\mathtt{A}| = m$) arcs representing dependency between the activities.

As the problem is periodic (it is executed an infinite number of times) we have an infinite number of repetitions of the same task. We call $start(i,\omega)$ the starting time of activity i at repetition ω. An edge (i,j) in this setting is interpreted as a precedence constraint such that: $start(j,\omega) \geq start(i,\omega) + d_i$. Moreover, a dependency edge from activity i to activity j might be associated with a minimal time lag $\theta_{(i,j)}$ and a repetition distance $\delta_{(i,j)}$. Every edge of the graph can 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 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. For a periodic schedule, the start times follow a static pattern, repeated over iterations: $start(i,\omega) = start(i,0) + \omega \cdot \lambda$, 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, we can deduce that scheduling a periodic task-set implies finding feasible assignments for $start(i,0)$ and a feasible modulus λ. As the start times at iteration 0 might be greater than the modulus, we can

decompose it as: $start(i,0) = \mathtt{s_i} + \mathtt{k_i} \cdot \lambda$, where $\mathtt{s_i}$ is the start time *within the modulus* ($0 \leq \mathtt{s_i} \leq \lambda - \mathtt{d_i}$) and $\mathtt{k_i}$, called iteration number, refers to the number of full periods elapsed before $start(i,0)$ is scheduled.

The key step of our approach is the generalization of equation (1) that enables activities to be assigned to arbitrary iterations (i.e. by taking into account inter-iteration overlappings). Hence, for every (i,j), we have a *modular* precedence constraint:

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

where $\mathtt{k_i}$ and $\mathtt{k_j}$ denote the iteration numbers to which activities i and j belong. Note that, by definition $\mathtt{k}_j \geq \mathtt{k}_i - \delta_{(i,j)}$. The modular precedence relation can be satisfied either by modification of the start times $\mathtt{s_i}$, or by moving the involved activities across iterations.

Finally, each activity i requires a certain amount req_{i,r_k} of one or more renewable resources r_k with capacity cap_{r_k}.

The problem consists in finding a schedule (that is, an assignment of start times $\mathtt{s}$ and iteration values $\mathtt{k}$ to activities), such that no resource capacity is exceed at any point of time and the modulus λ of the schedule (that is the makespan) is minimized.

In the context of chemical processes, the problem described above is described via Petri net's (see [23]), where δ is the number of markers over the places. In embedded system design this model is equivalent to scheduling homogeneous synchronous data-flow graphs (see [18]), where δ is the number of initial tokens over the buffers.

Fig. 1 presents a simple instance with 5 activities with different execution times and resource consumption. Assuming a total resource capacity of 3, the schedule depicted in Fig. 1 and labeled as *solution* is the optimal schedule. The activity coloured in light grey are scheduled at iteration 0 while the darkest at iteration 1. The execution of the schedule is plotted in Fig. 1 labeled as *execution*.

3 Constraint-based approach

We propose a complete constraint-based approach for the cyclic scheduling problem. The model is based on modular arithmetic. The solving algorithm interleaves propagation and search. We have implemented a modular precedence constraint taking into account propagation on iteration variables, start time and the

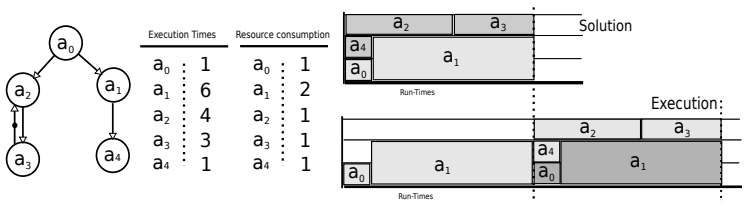

Fig. 1. Example instance and schedule

modulus variable. The search strategy interleaves the instantiation of start and iteration variables.

3.1 Model

The model we devised is based on three classes of variables: activity starting time within the modulus, iterations and the modulus. The activity starting time has a domain [0..MAX_TIME], the iterations have the domain [- $|V|$.. + $|V|$] and the modulus]0..MAX_TIME], where $|V|$ is the number of nodes and MAX_TIME represents the sum of the total execution time of the activities and the sum of the time lags of the edges. These variable are subject to temporal, resource (including buffers) and symmetry breaking constraints. In the following we explain each type of constraints.

Temporal constraints The time model we devised is based on the Simple Temporal Network Model (see [8]) where each node i of the graph is represented with a pair of time points t_i^s, t_i^e with associated time windows, connected by directional binary constraints of the form:

$t_i^s \xrightarrow{[d_i]} t_i^e$

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

Each edge (i, j) of the graph, described by (2), is represented as:

$t_i^e \xrightarrow{[\theta_{(i,j)}, \mathtt{k}_i, \mathtt{k}_j, \delta_{(i,j)}]} t_j^s$

where $\theta_{(i,j)}$ is the minimal time lag between the end of i (t_i^e) and the start of j (t_j^s). The construct also takes in account the iteration numbers $\mathtt{k}_i, \mathtt{k}_j$ and their minimal iteration distance $\delta_{(i,j)}$. This precedence is modeled through a dedicated *Modular Precedence Constraint* whose signature is as follows. Let $e = (i, j) \in \mathtt{A}$ be an edge of the graph.

$$\mathtt{ModPCst}([\mathtt{t^e_i}], [\mathtt{t^s_j}], [\mathtt{k_i}], [\mathtt{k_j}], [\lambda], \theta, \delta)$$

where $[\mathtt{t}_i^e], [\mathtt{t}_j^s], [\mathtt{k}_i], [\mathtt{k}_j], [\lambda]$ are the variables representing the end of activity i, the starting time of activity j, their respective iterations and the modulus, and $\theta_{(i,j)} = \theta$, $\delta_{(i,j)} = \delta$ are constant values. The filtering for a single precedence relation constraint achieves GAC and runs in constant time.

Resource constraints (including buffers) Unary and discrete resources in cyclic scheduling are modeled via traditional resource constraints. In fact, having starting time within a modulus, we can reuse the results achieved in constraint-based scheduling.In addition, in real contexts, the precedence constraint often implies an exchange of intermediate step products between activities that should be stored in buffers. For example, in the embedded system context activities may exchange data packets that should be stored in memory buffers.

Every time the activity i ends, its *product* is accumulated in a *buffer* and whenever the activity j starts, it consumes a *product* from it. It is common

to have size limits for each *buffer*. We model this limit through the following constraint:

$$\mathtt{k}_j - \mathtt{k}_i + (t_i^e \leq t_j^s) \leq \mathtt{B}_{(i,j)} - \delta_{(i,j)} \tag{3}$$

where $\mathtt{B}_{(i,j)}$ is the size limit of the *buffer* and the reified constraint $(t_i^e \leq t_j^s)$ equals one if the condition is satisfied. Obviously $\mathtt{B}_{(i,j)} \geq \delta_{(i,j)}$, otherwise the problem is unsolvable. In fact, the value $\delta_{(i,j)}$ counts the number of *products* already accumulated in the *buffer* (i, j) as initial condition. Inequality (3) limits the number of executions of activity i (*the producer*) before the first execution of j (*the consumer*).

Symmetry Breaking constraints One important observation is that the assignment of different iteration values to communicating tasks allows one to *break* precedence relation on the modular time horizon. Suppose there exists a precedence between activities (i, j) and suppose we decide to overlap their executions, such that $\mathtt{s}_j \leq \mathtt{s}_i + d_i$ and that $\theta_{(i,j)} = \delta_{(i,j)} = 0$. From (2) we derive:

$$\mathtt{k}_j \geq \mathtt{k}_i + \left\lceil \frac{\mathtt{s}_i + d_i - \mathtt{s}_j}{\lambda} \right\rceil \tag{4}$$

Observe that the precedence relation does indeed hold, but appears to be violated on the modular time horizon. The minimum iteration difference that enables such an apparent order swap to occur is with $\mathtt{k}_j = \mathtt{k}_i + 1$, and any larger number obtains the same effect.

Therefore the iteration value of the activity j should be at most one unit greater than all the predecessors iteration values: the following constraint is used to break symmetries and narrow the search space on $\mathtt{k}$ variables:

$$\mathtt{k}_j \leq \max_{i \in \mathtt{P}_j} (\mathtt{k}_i - \delta_{(i,j)} + \left\lceil \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda} \right\rceil) + 1 \tag{5}$$

where $\mathtt{P}_j : \{i \in \mathtt{P}_j | (i, j) \in \mathtt{A}\}$ is the set of the predecessors of j.

3.2 Constraint Propagation

The filtering on buffer and symmetry breaking constraints is the one embedded in mathematical constraints. In the same way, traditional resource constraints are propagated as usual. What is original in this paper is the filtering algorithm for the modular precedence constraint.

Modular Precedence Constraint The filtering algorithm of the Modular Precedence Constraint has three fundamental components:

- Filtering on the iteration variables $\mathtt{k}$: the goal of this component is to maintain a proper distance between iteration variables.

– Filtering on the start time variables $\mathtt{s}$: the aim of this part is to modify the start times of the involved variables to avoid infeasible overlapping of activities.
– Filtering on the modulus variable λ: this phase computes and sets a lower bound for the modulus.

The algorithm is executed whenever the domain of any variable involved changes.

Referring to the temporal model proposed in section 3.1 we can rewrite the inequality (2) as:

$$\mathtt{t}^{\mathtt{s}}_{\mathtt{j}} + \mathtt{k}_{\mathtt{j}} \cdot \lambda \geq \mathtt{t}^{\mathtt{e}}_{\mathtt{i}} + \theta + (\mathtt{k}_{\mathtt{i}} - \delta) \cdot \lambda \tag{6}$$

Filtering on iteration variables Starting from the equation above, we have

$$\mathtt{k}_{\mathtt{i}} - \mathtt{k}_{\mathtt{j}} - \delta \leq \frac{\mathtt{t}^{\mathtt{s}}_{\mathtt{j}} - \mathtt{t}^{\mathtt{e}}_{\mathtt{i}} - \theta}{\lambda} \tag{7}$$

with $-\lambda \leq \mathtt{t}^{\mathtt{s}}_{\mathtt{j}} - \mathtt{t}^{\mathtt{e}}_{\mathtt{i}} \leq \lambda$ and $\theta \geq 0$.

In the following we refer to $\overline{x}$ and $\underline{x}$ as the highest and the lowest values of the domain of a generic variable x.

Then, we can identify the highest integer value of the right part of the inequality: $\left\lfloor \frac{\overline{\mathtt{t}^{\mathtt{s}}_{\mathtt{j}}} - \underline{\mathtt{t}^{\mathtt{e}}_{\mathtt{i}}} - \theta}{\overline{\lambda}} \right\rfloor$ that is upper-bounded to 1 if $d_j = d_i = 0$, $\theta = 0$ and $\overline{\mathtt{t}^{\mathtt{s}}_{\mathtt{j}}} = \overline{\lambda}$, $\underline{\mathtt{t}^{\mathtt{e}}_{\mathtt{i}}} = 0$.

Hence, we can define two expressions computing bounds over the $\mathtt{k}$ variables:

$$\mathtt{k}_{\mathtt{i}} \leq \overline{\mathtt{k}_{\mathtt{j}}} + \delta + \left\lfloor \frac{\overline{\mathtt{t}^{\mathtt{s}}_{\mathtt{j}}} - \underline{\mathtt{t}^{\mathtt{e}}_{\mathtt{i}}} - \theta}{\overline{\lambda}} \right\rfloor \tag{8}$$

$$\mathtt{k}_{\mathtt{j}} \geq \underline{\mathtt{k}_{\mathtt{i}}} - \delta - \left\lfloor \frac{\overline{\mathtt{t}^{\mathtt{s}}_{\mathtt{j}}} - \underline{\mathtt{t}^{\mathtt{e}}_{\mathtt{i}}} - \theta}{\overline{\lambda}} \right\rfloor \tag{9}$$

As an example, suppose during search two activities i and j connected with a precedence (i, j) temporally overlap and that $\mathtt{t}^{\mathtt{s}}_{\mathtt{j}} = 0, \mathtt{t}^{\mathtt{e}}_{\mathtt{i}} = 3, \delta = 0, \theta = 0$: then the inequality (3.2) appears as follows:

$$\mathtt{k}_{\mathtt{j}} \geq \underline{\mathtt{k}_{\mathtt{i}}} - \left\lfloor \frac{-3}{\overline{\lambda}} \right\rfloor \tag{10}$$

that implies that $\mathtt{k}_{\mathtt{j}} > \mathtt{k}_{\mathtt{i}}$; in fact, two *connected* activities can overlap iff their iteration values differ: in particular $\mathtt{k}_{sinkNode} > \mathtt{k}_{sourceNode}$.

Filtering on starting variables Let now $\Delta_{\mathtt{k}} = \mathtt{k}_{\mathtt{i}} - \mathtt{k}_{\mathtt{j}} - \delta$ and $\underline{\Delta_{\mathtt{k}}} = \underline{\mathtt{k}_{\mathtt{i}}} - \overline{\mathtt{k}_{\mathtt{j}}} - \delta$, the constraint (6) could be written as:

$$\mathtt{t}^{\mathtt{s}}_{\mathtt{j}} - \mathtt{t}^{\mathtt{e}}_{\mathtt{i}} - \theta \geq \Delta_{\mathtt{k}} \cdot \lambda \tag{11}$$

It is trivial to prove that $\Delta_{\mathtt{k}} \leq 0$: in fact $\Delta_{\mathtt{k}} > 0$ implies that $\mathtt{k}_{\mathtt{j}} < \mathtt{k}_{\mathtt{i}} - \delta$ which is, by definition, impossible.

If $\Delta_{\mathtt{k}} \leq 0$, we can deduce two inequalities and their relative bound:

$$\mathtt{t}^{\mathtt{s}}_{\mathtt{j}} \geq \mathtt{t}^{\mathtt{e}}_{\mathtt{i}} + \theta + \Delta_{\mathtt{k}} \cdot \lambda \geq \underline{\mathtt{t}^{\mathtt{e}}_{\mathtt{i}}} + \theta + \Delta_{\mathtt{k}} \cdot \overline{\lambda} \tag{12}$$

$$\mathtt{t}_\mathtt{i}^\mathtt{e} \leq \mathtt{t}_\mathtt{j}^\mathtt{s} - \theta - \Delta_\mathtt{k} \cdot \lambda \leq \overline{\mathtt{t}_\mathtt{j}^\mathtt{s}} - \theta - \Delta_\mathtt{k} \cdot \overline{\lambda} \tag{13}$$

Note that, if $\Delta_\mathtt{k} = 0$, the modular constraint is turned into a simple time constraint: $\mathtt{t}_\mathtt{j}^\mathtt{s} \geq \mathtt{t}_\mathtt{i}^\mathtt{e} + \theta$. In fact, two *connected* activities with the same iteration value cannot overlap and the inequality (12) *pushes* the destination activity j after the end time of i plus the arc time lag θ.

Filtering on the modulus The filtering on the modulus variable can be obtained only in one case, namely when $\Delta_\mathtt{k} < 0$: in fact, we can derive from formula (11) the following inequality, that computes a lower bound on the modulus variable:

$$\lambda \geq \frac{\mathtt{t}_\mathtt{i}^\mathtt{e} - \mathtt{t}_\mathtt{j}^\mathtt{s} + \theta}{-\Delta_\mathtt{k}} = \left\lceil \frac{\mathtt{t}_\mathtt{i}^\mathtt{e} - \overline{\mathtt{t}_\mathtt{j}^\mathtt{s}} + \theta}{\overline{\mathtt{k}_\mathtt{j}} - \underline{\mathtt{k}_\mathtt{i}} + \delta} \right\rceil \tag{14}$$

3.3 Search

The solver is based on *tree search* adopting a *schedule or postpone* approach (described in [17]).

The main idea underlying the search strategy is the following: since the schedule is periodic, the starting time values of each activity can be positioned with respect to an arbitrarily chosen reference; we select one activity as reference and assign its starting time to zero. The choice is guided by a simple heuristics; in particular, the candidates to become the reference node should have no in-going arc with a strictly positive δ. Formally, for node i to be a candidate, it must hold $\nexists (j,i) \in \mathtt{A}$ such that $\delta_{(j,i)} > 0$. Then we accord preference to nodes with a high number of outgoing arcs. The rationale behind this choice is that it seems to ease propagation of symmetry breaking constraints. Hence, the reference node assumes standard values: $\mathtt{t}_{src}^s = 0, \mathtt{k}_{src} = 0$. Note that fixing the start time of one activity to 0 does not compromise completeness as the iteration variables $\mathtt{k} \in \mathbb{Z}$ can assume negative values: therefore optimality is guaranteed.

At each search step a new node is selected among the activities connected with the already considered nodes. Two activities are connected if there exists al least an edge between them. This method improves the efficiency of the propagation of symmetry breaking constraints. Other variable selection strategies radically worsen the performance of the solver.

The search interleaves the assignment of start times and iteration values. The algorithm assigns to an activity its earliest start time. In backtracking, the decision is postponed until its earliest start time has been removed. This removal can occur as a result of a combination of search decisions and constraints propagation. Hence, the algorithm, considering the same activity, assigns to it an iteration value. Note that the iteration assigned is always the lowest absolute value in the variable domain. In case of failure, the value is removed from the domain and a higher number is assigned. In fact, if a solution results infeasible with $\mathtt{k}_i = \phi$, it is trivial to prove that it is infeasible for any value $\phi' \leq \phi$ (remember that $\mathtt{k}_{sinkNode} \geq \mathtt{k}_{sourceNode} - \delta$).

4 Experimental Results

The main purpose of the experimental evaluation is to show our approach to cyclic scheduling is viable; in particular, the focus is to assess the effectiveness on practically significant benchmarks.

In this work we show four groups of experimental results: the first (1) considers an industrial set of 36 instruction scheduling instances for the ST200 processor by STmicroelectronics [2]. The second (2) group considers a set of synthetic instances and compares the best solution obtained within 300 seconds and the (ideal) lower bound of the instance. The third (3) set of experiments considers a set of a synthetic instances and compares the solution quality of our *modulo* approach with a classic *blocked* approach on cyclic scheduling (see [3]). Finally, the fourth (4) group of results is used to assess the efficiency of the combined propagation of the buffer constraints and the symmetry breaking constraints.

The system described so far was implemented on top of ILOG Solver 6.7; all tests were run on a 2.8GHz Core2Duo machine with 4GB of RAM. The synthetic instances were built by means of a internally developed task-graph generator, designed to produce graphs with realistic structure and parameters. Designing the generator itself was a non-trivial task, but it is out of the scope of this paper.

4.1 Industrial Instances

The first set of 36 instances refers to compilers for VLIW architectures. These instances belong to compilers when optimizing inner loops at instruction level for very-long instruction word (VLIW) parallel processors. Since the resource consumption is always unary, to obtain a more challenging set, in [2] the authors replaced the original resource consumption with a random integer number bounded by the resource capacity. Nodes represent operations and their execution time is unary: the smallest instance features 10 nodes and 42 arcs, while the largest one features 214 nodes and 1063 arcs. In [2] the authors present two ILP formulations for the resource-constrained modulo scheduling problem (RCMSP), which is equivalent to a cyclic RCPSP. As described in [2], both ILP approaches adopt a *dual* process by iteratively increasing an infeasible lower bound; as a consequence, the method does not provide any feasible solution before the optimum is reached. Given a large time limit (604800 seconds) their solvers found the optimal solution for almost all the instances: our experiments compare the optimal value and the solution found by our method within a 300 sec. time limit. In addition, to empathize the quality of our solutions, we compare also with a state of the art heuristic approach: the *Swing Modulo Scheduling (SMS)*, presented in [20], used by the gcc compiler [13].

The set of instances is composed by two subsets: the easiest one containing industrial instances and the more difficult one generated with random resource consumptions. Tab. 1 reports a summary of the experiments results: the first three columns describe the instances (name, number of nodes and arcs), the third shows the run-times of the fastest ILP approach (in [2]), the fourth reports the quality of our solutions achieved within a second and 300 seconds and

the fifth presents the quality of the solutions computed with the *SMS* heuristic approach. The remaining three columns report the same figures referred to the modified set of instances. For the *easiest* set, our method computes the optimal value within one second for all but one instance (`adpcm-st231.2`) whose optimal gap is 2.44%. We also found a solution for the `gsm-st231.18` instance that was previously unsolved; clearly, we cannot evaluate its quality as the optimal solution has never been found.

We also compared our approach with the *SMS* heuristic that presents an average optimality gap of 14.58%. Its average solution time is lower than ten seconds; the highest computation time refers to instance `gsm-st231.18` and is 58 seconds.

The last three columns of Tab. 1 report the experimental results on the *modified* set of instances. Again, `time` refers to the fastest ILP approach, while `gap(%) and SMS(%)` report the optimality gap of our approach and the *SMS* heuristic respectively. Within a second we found the optimal value of 89% of the instances and the average gap is 0.813%. Within 300 seconds its value improves to 0.61%. *Swing Modulo Schedule* solves all the instances with an average gap of 3.03% within few tens of seconds.

Finally, referring to instances `gsm-st231.25` and `gsm-st231.33`, the authors of [2] claim to compute two sub-optimal solutions in 604800 seconds (though no details are given on how the dual process they propose converges to sub-optimal solutions). For instance `gsm-st231.25` both the modular and the SMS approaches find the same solution as the bound computed in the original paper. Instead, in `gsm-st231.33`, the solution proposed in [2] has value 52. While the heuristic solver finds the same solution, the modular method finds in one second a solution with value 47 and within 56 seconds a solution of value 46.

Note that, within the time limit we prove optimality in 12.5% of the instances: we are currently investigating how to improve the efficiency of the proof of optimality, even though the optimality gap is so narrow to reduce the significance of finding (and proving) the optimal solution.

4.2 Evaluating the solution quality on synthetic benchmarks

The second set of experiments targets a task scheduling problem over a multiprocessor platform; this set contains 1200 synthetic instances with 20 to 100 activities: each instance corresponds to a cyclic graph with a high concurrency between the activities. This form has been empirically proven to be the hardest structure for the solver developed. The generated instances contain a single cumulative resource with capacity 6 and activities have a unary consumption.

Tests run with a time limit of 300 seconds: Tab. 2 shows the average, best and worst gap between the best solution found within a time limit (reported in the first column) and the ideal lower bound of the instance.

The lower bound is the following:
$lb = \left\lceil \max\left(ib, \frac{\sum_{i \in V} d_i}{cap}\right) \right\rceil$,

			Industrial			Modified		
Instances	nodes	arcs	time(sec)	**Gap(%)**	SMS(%)	time(sec)	**Gap(%)**	SMS(%)
adpcm-st231.1	86	405	14400	0%	19.23%	X	X	X
adpcm-st231.2	142	722	582362	2.44/2.44%	0%	X	X	X
gsm-st231.1	30	190	0.05	0%	0%	250	10.7/10.7%	10.7%
gsm-st231.2	101	462	79362	0%	0%	X	X	X
gsm-st231.5	44	192	0.05	0%	13.33%	280	0%	5.26%
gsm-st231.6	30	130	17	0%	31.25%	152	0%	0%
gsm-st231.7	44	192	0.05	0%	41.66%	92	0%	2.38%
gsm-st231.8	14	66	0.05	0%	31.25%	0.27	0%	0%
gsm-st231.9	34	154	0.05	0%	0%	0.56	5.88/0%	8.57%
gsm-st231.10	10	42	0.05	0%	0%	0.1	0%	0%
gsm-st231.11	26	137	0.05	0%	0%	0.37	0%	0%
gsm-st231.12	15	70	0.05	0%	0%	12.65	0%	0%
gsm-st231.13	46	210	1856	0%	0%	985.03	0%	0%
gsm-st231.14	39	176	301.25	0%	17.39%	220	2.94/2.94%	0%
gsm-st231.15	15	70	0.05	0%	28.57%	12.36	0%	8.33%
gsm-st231.16	65	323	7520	0%	0%	X	X	X
gsm-st231.17	38	173	0.05	0%	23.81%	90	0%	0%
gsm-st231.18	214	1063	X	0%	30.76%	X	X	X
gsm-st231.19	19	86	0.05	0%	0%	38.23	0%	6.25%
gsm-st231.20	23	102	0.05	0%	0%	123	3.23/3.23%	4.76%
gsm-st231.21	33	154	0.05	0%	45.45%	42.06	0%	3.24%
gsm-st231.22	31	146	0.05	0%	0%	80.36	0%	0%
gsm-st231.25	60	273	3652	0%	0%	(604800)	0%	1.75%
gsm-st231.29	44	192	12.6	0%	23.81%	210	0%	0%
gsm-st231.30	30	130	12	0%	0%	58	0%	3.84%
gsm-st231.31	44	192	47	0%	41.67%	142	0%	2.5%
gsm-st231.32	32	138	0.05	0%	31.25%	0.25	0	0%
gsm-st231.33	59	266	2365	0%	11.76%	(604800)	0%	0%
gsm-st231.34	10	42	0.05	0%	6.25%	5.05	0%	0%
gsm-st231.35	18	80	0.05	0%	0%	52	0%	0%
gsm-st231.36	31	143	27	0%	14.29%	230	0%	7.69%
gsm-st231.39	26	118	0.05	0%	0%	95	0%	4.55%
gsm-st231.40	21	103	0.05	0%	0%	15	0%	5.56%
gsm-st231.41	60	315	2356	0%	0%	X	X	X
gsm-st231.42	23	102	0.05	0%	0%	12	0%	14.29%
gsm-st231.43	26	115	0.05	0%	21.73%	15	0%	9.1%

Table 1. Run-Times/Gaps of Industrial/Modified instances

that is the maximum between the intrinsic iteration bound[1] ib of the graph and the ratio between the sum of the execution times and the total capacity.

The first row of the table reports that within one second run-time the solver finds a solution which is about 3.7% distant from the ideal optimal value; at the end of the time limit the gap is decreased to 2.9%. Fig. 2 depicts a zoom of the progress of the gap values.

From the results of these experiments, we can conclude that our approach converges very quickly close to a value that is an ideal optimal. The optimal value lies somewhere in-between the two values and therefore is even closer to the solution found within 300 seconds.

[1] The iteration bound of the graph is in relation with the cycles, in particular with the maximum cycle mean; details in [7, 12].

time(s)	avg(%)	best(%)	worst(%)
1	3.706%	2.28%	5.18%
2	3.68%	2.105%	5.04%
5	3.51%	1.81%	5.015%
10	3.37%	1.538%	4.98%
60	3.14%	1.102%	4.83%
300	2.9%	0.518%	4.73%

Table 2. Solution quality

4.3 Modulo Vs Unfolding Scheduling

The aim of this third experimentation is to investigate the impact of the overlapped schedule (namely a schedule explicitly designed such that successive iterations in the application overlap) w.r.t. the so called *blocked* classic approach that considers only one iteration. Since the problem is periodic, and the schedule is iterated essentially infinitely, the latter method pays a penalty in the quality of the schedule obtained. A technique often used to exploit inter-iteration parallelism is **unfolding** (see [22]). The unfolding strategy schedules u iterations of the application, where u is called the blocking factor. Unfolding often leads to improved blocked schedules, but it also implies an increased size of the instance.

The third set of instances contains 220 instances with an activity number from 14 to 65. We have divided these instances into three classes: small instances featuring 14 to 24 nodes, medium-size instances (25 to 44 activities) and big instances with 45 to 65 activities. Also we have considered eight solver configurations: the *blocked* one (scheduling only one iteration) and seven configurations called *UnfoldX* where X is the number of iteration scheduled. Tab. 3 shows the average gap between the above mentioned configurations and our approach. Obviously, the worst gap is relative to the blocked schedule, while the unfolded ones tend to have an oscillatory behaviour. Fig. 3 depicts the relation

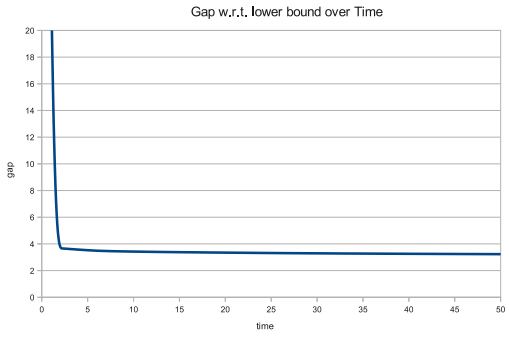

Fig. 2. Graphical representation of the optimality gap

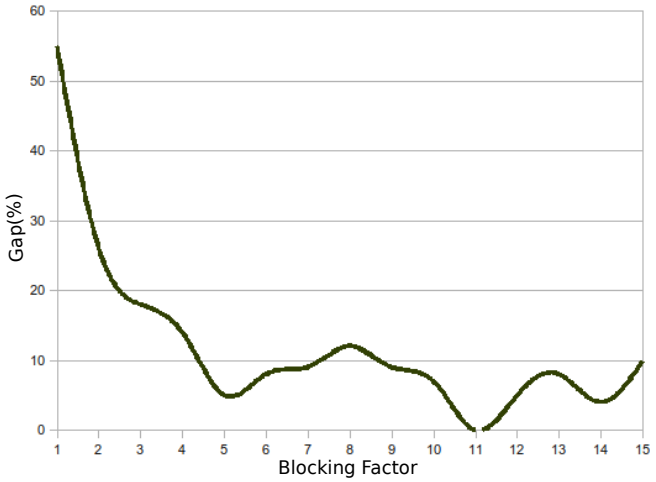

Fig. 3. Graphical representation of the optimality gap

between the gap (`Y` axes) and the blocking factor (`X` axes) of a selected instance with 30 nodes. The figure highlights the waveform of the gap. With unfolding factor $u = 11$ the solver found a solution equivalent to the overlapped one, with the difference that the unfolded problem consists of 330 activities. The last column presents the average optimality gap over the whole experimental set.

Note that for the instance analysed in Fig. 3 there exists a blocking factor that enables to find the optimal solution. However, Parhi and Messerschmitt in [22] widely studied the unfolding techniques and provide an example of an application for which no blocked schedule can be found, even allowing unfolding.

4.4 Buffer-Size Constraints Tests

The last set of tests contains 400 instances with 20 nodes; for each instance, the resource free optimal solution has buffer request from 3 to 6. This property was obtained via careful tuning of the instance generation problem.

The purpose of this experimental section is to highlight the efficiency of the buffer and symmetry breaking propagation. The results show that reasonable limits on the buffer size do not compromise solution quality, while providing a tremendous boost to the search time. All instances were solved with five different buffer sizes (1,2,3,6,9). Tab. 4 reports the average and median running time. The first two rows refer to tightly constrained instances, while the last one refers to loosely constrained instances.

The second and the third columns show respectively the average and the median run times in seconds to find the optimal solution. Note that the propagation drastically impacts the run time of the search.

Another interesting aspect to observe is that since value 6 is the intrinsic buffer size of the instances, a solution with that buffer limit is considered as

	Solution Gap (%)			
Solver	[14-20]	[25-40]	[45-65]	AVG
Blocked	108.16%	65.45%	38.83%	**55.32%**
Unfold2	55.92%	26.06%	19.89%	**26.23%**
Unfold3	33.31%	16.15%	9.99%	**18.6%**
Unfold4	29.41%	14.27%	6.278	**14.13%**
Unfold5	21.35%	5.33%	8.76%	**5.67%**
Unfold6	39.06%	8.67%	4.39%	**8.67%**
Unfold8	78.31%	10.71%	7.65%	**12.44%**
Unfold10	16.95%	10.21%	10.03%	**8.65%**

Table 3. Unfolding set results

buffesSize	avg(s)	median(s)	gap%
1	1.1423	0.05	4.925%
2	52.1894	0.1	0.052%
3	157.4673	0.31	0%
6	599.9671	1.215	0%
9	791.5552	1.83	0%

Table 4. Buffer set results

the *reference* solution. The last column presents the gap between the optimal found and the *reference*. The interesting aspect is that the trade-off between the run-time speed-up and the optimality loss is not linear: in fact, despite the tests with buffer limit 2 run over 10 times faster the solution, remains close to the *reference* one.

5 Related Work

The static cyclic scheduling literature arises from two main contexts: the industrial and the computing contexts. The former includes mass production (i.e. [15],[9]), chemical (i.e. [19]), robot flow-shop (i.e. [6]) and hoist scheduling problems(i.e. [5]), the latter includes parallel processing(i.e. [21],[14]), software pipelining(i.e. [24]) and the emerging embedded system data-flow problems (i.e. [16]).

There is a considerable body of work on the problem available in the OR literature, while from the AI community, the problem has not received much focus.

An important subset of cyclic scheduling problems is the modulo scheduling: here, start times and modulus are required to assume integer values; as stated in section 2 we make the same assumption.

Several heuristic and complete approaches have been proposed since many years to solve this problem. An heuristic approach is described in [24], 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 [4]. The latter method is based on an ILP model. A state of the art incomplete method is *Swing Modulo Scheduling* approach, described in [20], [21], and currently adopted in the GCC compiler [13].

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* makespan could be extremely large, with implications on the size of the model. Our model, instead, is compact, since both *horizon* and *initiation* makespan coincide in the modulus value.

Advanced complete formulations were proposed in [11, 10], both report ILP models; the first comprises both binary and integer variables while the latter includes only binary variables. In [2] the authors report an excellent overview of the state-of-the-art formulations (including Eichenberger and Dupont de Dinechin models) and present a new formulation issued from Danzig-Wolfe Decomposition. Finally, good overviews of complete methods can be found in [14, 1].

To the best of our knowledge the state-of-the-art complete approaches are based on iteratively solving the problem with a fixed modulus. Modeling the modulus as a decision variable yields non-linear mathematical programs; on the other hand, with a fixed value, the resulting problem is a linear mathematical program. Hence, fixing the modulus and iteratively solving an ILP model is a common formulation for solving cyclic RCPSP.

Our methodology is constraint-based and tackles the non-linear problem as a whole: the modulus value is inferred from the others variables avoiding the iterative solving procedure thus increasing efficiency.

6 Conclusions

We propose a constraint formulation based on modular arithmetic solving the cyclic resource-constraint project scheduling problem. Keys of the efficiency are three different set of constraints: the buffer, the symmetry breaking and the modular precedence constraints. In particular, for the latter we devise an original filtering algorithm.

The experiments highlight a good performance and a solution quality that converges close to the optimal very quickly. In particular, the solver is extremely effective in finding a solution with a negligible optimality loss in terms of a few seconds; conversely, the optimality proof takes much longer to complete.

As a natural extension, future works will focus on allowing an activity to be scheduled across different iterations. Consequently a new cumulative filtering algorithm should be studied.

References

1. C. Artigues, S. Demassey, and E. Néron. *Resource-constrained project scheduling - Models, Algorithms, Extensions and Applications.* Wiley, 2008.
2. 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.
3. Shuvra S. Bhattacharyya and Sundararajan Sriram. *Embedded Multiprocessors - Scheduling and Synchronization (Signal Processing and Communications) (2nd Edition).* CRC Press, 2009.
4. 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.

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. Y. Crama, V. Kats, J. van de Klundert, and E. Levner. Cyclic scheduling in robotic flowshops. *Annals of Operations Research*, 96:97–124, 2000. 10.1023/A:1018995317468.
7. A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 9(4):385–418, 2004.
8. R. Dechter. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
9. 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.
10. B. Dupont de Dinechin. From Machine Scheduling to VLIW Instruction Scheduling, 2004. http://www.cri.ensmp.fr/classement/doc/A-352.ps.
11. A.E. Eichenberger and E.S. Davidson. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices*, 32(5):194–205, 1997.
12. L. Georgiadis, A.V. Goldberg, R.E. Tarjan, and R.F. Werneck. An experimental study of minimum mean cycle algorithms. In *Proc. of ALENEX*, page 13, 2009.
13. Mostafa Hagog and Ayal Zaks. Swing modulo scheduling for gcc, 2004.
14. C. Hanen and A. Munier. *Cyclic scheduling on parallel processors: an overview*, pages 193–226. John Wiley & Sons Ltd,, 1994.
15. Claire Hanen. Study of a np-hard cyclic scheduling problem: The recurrent job-shop. *European Journal of Operational Research*, 72(1):82 – 101, 1994.
16. 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.
17. C. Le Pape and P. Couronné. Time-versus-capacity compromises in project scheduling. In *In Proc. of the 13th Workshop of the UK Planning Special Interest Group*, 1994.
18. E.A. Lee and D.G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.
19. Zukui Li and Marianthi Ierapetritou. Process scheduling under uncertainty: Review and challenges. *Computers and Chemical Engineering*, 32(4-5):715 – 727, 2008. Festschrift devoted to Rex Reklaitis on his 65th Birthday.
20. Josep Llosa, Antonio Gonzalez, Eduard Ayguade, and Mateo Valero. Swing Modulo Scheduling: A Lifetime-Sensitive Approach. In *PACT'96*, pages 80–87, 1996.
21. 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.
22. 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.
23. J.L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, February 1981.
24. R.B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proc. of MICRO-27*, pages 63–74. ACM, 1994.