Alma Mater Studiorum - Università di Bologna

Dottorato di Ricerca in
Ingegneria Elettronica, Informatica e delle Telecomunicazioni

Ciclo XXV

Settore Concorsuale di Afferenza: 09/H1
Settore Scientifico Disciplinare: ING-INF/05

# Constraint based methods for allocation and scheduling of periodic applications

Alessio Bonfietti

Il Coordinatore di Dottorato:

Alessandro Vanelli Coralli

I Relatori:

Paola Mello

Michela Milano

Esame Finale Anno 2013

# Contents

**Abstract**

This work presents exact algorithms for the Resource Allocation and Cyclic Scheduling Problems (RA&CSPs). Cyclic Scheduling Problems arise in a number of application areas, such as in hoist scheduling, mass production, compiler design (implementing scheduling loops on parallel architectures), software pipelining, and in embedded system design. The RA&CS problem concerns time and resource assignment to a set of activities, to be indefinitely repeated, subject to precedence and resource capacity constraints. In this work we present two constraint programming frameworks facing two different types of cyclic problems.

In first instance, we consider the disjunctive RA&CSP, where the allocation problem considers unary resources. The proposed method has broad applicability, but it is mainly motivated by applications in the field of Embedded System Design. Instances are described through the Synchronous Data-flow (SDF) Model of Computation. Data-Flow models are attracting renewed attention because they lend themselves to efficient mapping on multi-core architectures. The key problem of finding a maximum-throughput allocation and scheduling of Synchronous Data-Flow graphs onto a multi-core architecture is NP-hard and has been traditionally solved by means of heuristic (incomplete) algorithms with no guarantee of global optimality. We propose an exact (complete) algorithm for the computation of a maximum-throughput mapping of applications specified as SDFG onto multi-core architectures. Results show that the approach can handle realistic instances in terms of size and complexity. The basic idea of the approach we present is to model the effects of allocation and scheduling choices by means of graph modifications. During the search process, whenever allocation and scheduling decision are taken, the graph is modified accordingly. The efficiency of this approach hinges on an original global throughput constraint.

Next, we tackle the Cyclic Resource-Constrained Scheduling Problem (i.e. CRCSP). We propose a Constraint Programming approach based on modular arithmetic: in particular, we introduce a modular precedence constraint and a global cumulative constraint along with their filtering algorithms. We discuss two possible formulations. The first one (referred to as *CROSS*) models a pure cyclic scheduling problem and makes use of both our novel constraints. The second formulation (referred to as $CROSS^*$) introduces a restrictive assumption to enable the use of classical resources constraints, but may incur a loss of solution quality. Many traditional approaches to cyclic scheduling operate by fixing the period value and then solving a linear problem in a generate-and-test fashion. Conversely, our technique is based on a non-linear model and tackles the problem as a whole: the period value is inferred from the scheduling decisions. The proposed framework has been used in the MPOpt-Cell framework: a High-Performance Data-Flow Programming Environment for the Cell BE Processor by IBM, Sony and Toshiba.

The proposed approaches have been tested on a number of non-trivial synthetic instances and on a set of realistic industrial instances achieving good results on practical size problem. Furthermore, the developed techniques bring significant contributions to combinatorial optimization methods.

# Chapter 1

# Introduction

This work presents exact algorithms for the Resource Allocation and Cyclic Scheduling Problems (RA&CSPs). Cyclic Problems arise in a number of application areas, such as in hoist scheduling [26], mass production [48, 35], compiler design (implementing scheduling loops on parallel architectures) [67, 48], software pipelining [86], and in embedded system design [61, 81, 97]. 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 encoders and decoders 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 (called coding units, packets, pixels, depending on the context) 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.

From a combinatorial optimization standpoint, cyclic resource allocation and scheduling is the problem of assigning resources and starting times of periodic activities such that the periodic repetition (period $\lambda$) of the overall application is minimal and such that the precedence relations and the resource availabilities are respected. In other words, the schedule is repeated every $\lambda$ time units. All activities however should appear once in the period. Note that a minimal period corresponds to the highest number of activities executed on average over a large time window. As a consequence, the minimal period corresponds to the maximum application throughput[1].

Traditional resource-constrained scheduling techniques have achieved a good level of maturity in the last decade [6], but they cannot be trivially applied to cyclic scheduling problems in an efficient way. Hence four different approaches have been proposed to handle this type of cyclic problems:

- the so called blocked scheduling approach [14] that considers only one iteration and repeats it in sequence for an infinite number of times. Since the problem is periodic, but the optimal solution may require mixing activities from different repetitions within a single period, the blocked scheduling method can be highly sub-optimal.

- the unfolding approach [84] that schedules a number of consecutive iterations of the application. Unfolding often leads to improved blocked schedules, but it also implies an increased size of the instance. Moreover it has been not clear how to find the number of unrollings that lead to an optimal solution.

- the self-timed approach [65]; the method computes a static-order schedule[2], that is a set of ordering decisions between activities. The start times are deduced from such decisions in a second phase (usually at

[1]Cyclic scheduling problems are usually constrained with a throughput feasibility threshold.

[2]See Section 2.2 for details.

run-time). A recent heuristic approach (see [97]) uses simulation techniques to compute feasible static-order schedules.

- the modulo scheduling approach [48] that schedules the activities allowing the overlapping between several repetitions. The obtained periodic overlapping usually speeds-up the infinite execution.

## 1.1 Content

RA&CSPs often deal with two types of constraints: temporal dependencies (i.e. precedencies) and resource sharing. In this work we tackle two types of cyclic scheduling problems having the same precedence constraints, but different resources constraints:

- Disjunctive Problem: the RA&CS problem we tackle arises in the field of the embedded system design and it consists in scheduling and allocating a periodic application (i.e. a set of activities) on a set of Multi-Processor System-on-Chip (MPSoC) cores (i.e. unary resources). The applications is modeled through a Synchronous Data-Flow Graph SDFG[3], which is a particular project graph widely adopted in embedded system context. We propose a method that tackle the allocation and scheduling problem as a whole, avoiding the sub-optimality due to decomposition. The method is base on the self-timed techniques.

- Cumulative Problem: in this case, as all the activities share the same cumulative resources. The resource allocation problem is not considered, therefore the problem tackled is the Cyclic Resource-Constrained Scheduling Problem (CRCSP). Instances are described through a project graph presented in Section 2.1. We develop a novel framework based on modulo scheduling techniques. The solver developed is called *CROSS*[4].

[3] See Section 3.2 for a formal and detailed description.
[4] Cyclic Resource-cOnstrained Scheduling Solver.

## 1.2 Contribution

This thesis provides several contributions to the state-of-the-art solving techniques for RA&CSPs.

- A novel approach for solving the Disjunctive RA&CSP. The basic idea of this approach, described in Section 3, is to model the effects of allocation and scheduling choices by means of graph modifications. During the search process, whenever allocation and scheduling decision are taken, the graph is modified accordingly. The efficiency of this approach hinges on a global throughput constraint. This work was published in [21].
  - A new global constraint for throughput filtering algorithm maintaining a tight bound on the maximum achievable throughput based on the current state of the search. We propose two versions of the algorithm: the non-incremental and the incremental version, the second reaching on order of magnitude speed-up with respect to the former with very significant benefits on scalability. This work was published in [18].
- The *CROSS* solver. The solver (described in Section 4) focuses on the CRCSP and is used in the MPOpt-Cell framework (presented in Section 5). The *CROSS* approach is based on modular algebra and its model is enforced by two specific constraints used to model precedences and cumulative resources.
  - An original Modular Precedence Constraint (MPC) and its filtering algorithm. This work was published in [19].
  - A Global Cyclic Cumulative Constraint (GCCC) and its filtering algorithms. This work was published in [20].
  - A random restart based search strategy where we set the upper bound of the period variable (i.e. the modulus) while the lower bound is inferred from the other variables. This is in contrast

with classical modular approaches that fix the period and solve the corresponding scheduling sub-problem.

- A solution strategy with the underlying hypothesis that the end times of all activities should be assigned within the modulus. Thanks to this assumption, we can simplify the model reusing traditional resource constraints and filtering algorithms.

The *CROSS* approach 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 gave promising results compared with a state-of-the art ILP-based (Integer Linear Programming) scheduler and the Swing Modulo Scheduling (SMS) heuristic technique. SMS is a non-complete (heuristic) modular approach adopted in the gcc compiler [45]. In addition, the experiments show that our technique greatly outperforms both the blocked and the unfolding approaches in terms of solution quality.

Our method are based on Constraint Programming (CP) [88], a declarative programming paradigm based on constraint propagation and search (for details see Appendix A)

## 1.3 Outline

The thesis is structured as follows: in Chapter 2 we formally define the problems and we introduce the necessary background terminology. Chapter 3 presents the developed framework and the experimental results for the disjunctive RA&CSP. In Chapter 4 we present the *CROSS* framework based on modulo scheduling techniques.

Note that the state-of-the-art approaches related to the disjunctive RA&CSPs (in Embedded System design context) are presented in Section 3.3.2, while the approaches and the methods related to the CRCSP are described in Section 4.1

Then in Chapter 5 we present the MPOpt-Cell framework [39], a High-Performance Data-Flow Programming Environment for the Cell BE Processor[5] based on *CROSS* solver. Finally Chapter 6 concludes this work with some remarks and directions for possible further research.

[5] `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`

# Chapter 2

# Problem Definition

In this chapter, we introduce terminology and definitions used in this thesis and then we provide some intuitions about the allocation and scheduling problems defining formally the concepts needed to address them.

## 2.1 Background

The problem can be described through a project graph.

**Definition 1.** *A Project Graph* $\mathbb{G}$ *is a directed graph consisting of a pair* $\langle \mathbb{V}, \mathbb{A} \rangle$*, where*

- *elements in* $\mathbb{V}$ *(*$||\mathbb{V}|| = n$*) are nodes that represent activities.*
- *elements in* $\mathbb{A}$ *(*$||\mathbb{A}|| = m$*) are arcs. An arc* $(i, j)$ *represents a temporal dependency between activities* $i$ *and* $j$*.*

We assume that every activity $i \in \mathbb{V}$ in the graph $\mathbb{G}$ has a fixed duration $d_i$.

We refer to $(i, \omega)$ as the $\omega$-th execution of activity $i \in \mathbb{V}$, where $\omega \in \mathbb{Z}$ is called execution number. A set of executions $(i, \omega)$ of all the activities in $\mathbb{V}$ with the same $\omega$ value is referred to as a *repetition*. A *schedule* is defined as an assignment of start times to all executions $(i, \omega)$. We refer to $start(i, \omega)$ as the starting time of activity $i$ at execution $\omega$. Without loss of generality, we also assume that $start(i, \omega) \geq start(i, \omega')$ if $\omega \geq \omega'$ and that $start(i, \omega) \geq 0$.

The most important performance metric of a cyclic schedule is the average inter-execution distance, which is strictly related to the concept of execution frequency (i.e. the throughput).

**Definition 2.** *The average inter-execution distance $\lambda(i)$ of an activity $i$ is defined as the following limit:*

$$\lambda(i) = \lim_{\omega' \to \infty} \frac{\sum_{\omega=0}^{\omega'} (start(i, \omega + 1) - start(i, \omega))}{\omega'} \tag{2.1}$$

Note that, since the activities are repeated indefinitely, the sum can start for $\omega = 0$ without loss of generality, even if $\omega$ is in $\mathbb{Z}$ as from the problem definition.

**Definition 3.** *The average inter-execution distance $\lambda$ of a* set *of activities $\mathbb{V}$ is the worst case $\lambda(i)$.*

$$\lambda = \max_{i \in \mathbb{V}} (\lambda(i)) \tag{2.2}$$

The *throughput* of an activity $i$ is defined as the average number of executions of $i$ per time unit, and corresponds to the inverse of the average distance:

$$\textsc{Thp}(i) = \frac{1}{\lambda(i)}$$

Analogously, the throughput of a set of activities is the inverse of $\lambda$. Lower period (i.e. higher throughput) values are to be preferred since they correspond to more efficient schedules.

In practical cases, activities are subject to several restrictions. In particular, there may be temporal dependencies between activities and resource constraints.

### 2.1.1 Resource Constraints

A resource allocation problem consists in assigning activities to resources. The RA&CS problem considers a set $\mathtt{R}$ of limited capacity resources. For each resource $k \in \mathtt{R}$ its maximum capacity is $\mathrm{CAP}_k$. Each activity $i \in \mathbb{V}$ has

a set of resource requirements $r_{i,k}$ for all resources $k$ required by activity $i$. A zero requirement denotes a non-required resource.

A schedule is feasible if and only if, for each resource $k \in \mathtt{R}$, at any point in time $t$, the sum of the activities requirements $r_{i,k}$ do not exceed the capacity $\mathrm{CAP}_k$ of the resource.

$$\sum_{\substack{i \in \mathbb{V},\ \omega \in \mathbb{Z} \\ start(i,\omega) \leq t \\ t < start(i,\omega) + d_i}} r_{i,k} \leq \mathrm{CAP}_k \quad \forall t \in ]-\infty, \infty[, \quad \forall k \in \mathtt{R} \tag{2.3}$$

As stated in Section 1, in this thesis we focus on two problems: the disjunctive RA&CSP and the CRCSP (with cumulative resources). In the disjunctive problem the resources have unary capacity (i.e. $\mathrm{CAP}_k = 1\ \forall k \in \mathtt{R}$), while in CRCSP we consider cumulative discrete resources.

Notice that in the CRCSP the resource allocation problem is not considered. However, a problem considering a set of $p$ unary resources (as the Disjunctive RA&CSP) is analogous to a problem with a single shared resource of capacity $\mathrm{CAP} = p$ (i.e. CRCSP). Therefore the *CROSS* framework, developed for the CRCSP, can be used to tackle the disjunctive RA&CSP.

In fact, the use of a single shared resource of capacity $\mathrm{CAP} = p$, implies that, considering a feasible schedule, the maximum number of activities that can execute concurrently are $p$. Hence, a set of $p$ unary resources is enough to implement the same feasible schedule.

### 2.1.2 Time constraints

Temporal dependencies between activities are represented through arcs of the project graph.

Moreover, in a traditional scheduling problem a temporal dependency links the only existing execution of activity $i$ with the only existing execution of activity $j$. On the other hand, in cyclic scheduling, since activities have multiple executions, the temporal dependency should be augmented so as to take into account the infinite executions.

**Definition 4.** *Each directed edge $(i,j) \in \mathbb{A}$ in the project graph $\mathbb{G}$ is a tuple $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle$, where*

- *$i$ is the source activity*
- *$j$ is the sink activity*
- *$\theta_{(i,j)} \in \mathbb{R}$ is called minimum time lag;*
- *$\delta_{(i,j)} \in \mathbb{Z}$ is called the repetition distance*[1]*;*

*the edge $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle$ enforces the relation:*

$$start(j, \omega) \geq start(i, \omega - \delta_{(i,j)}) + d_i + \theta_{(i,j)} \quad \forall \omega \in \mathbb{Z} \tag{2.4}$$

Observe that, in a cyclic problem, a temporal dependency connects an infinite number of distinct pairs of executions of $i$ and $j$. The value $\delta_{(i,j)}$ acts as a repetition offset: it declares the distance in terms of repetitions between the executions of the connected activities. Moreover, if $\delta_{(i,j)} = 0$, the edge $\langle i, j, \theta_{(i,j)}, 0 \rangle$ is called *intra-repetition* edge while if $\delta_{(i,j)} \neq 0$, we call $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle$ *inter-repetition* edge. The time lag $\theta_{(i,j)}$ specifies the minimal temporal distance between the start of execution $(j, \omega)$ and the end of execution $(i, \omega - \delta_{(i,j)})$.

Note that, the presence of inter-repetition dependencies may create feasible cycles in the project graph.

Consider the following simple example about building a skyscraper, depicted in Figure 2.1. The project graph contains the two activities $floor$ and $pillars$ (having respectively duration 5 and 2) and two arcs:

$\langle floor, pillars, 2, 0 \rangle$ and $\langle pillars, floor, 1, 1 \rangle$.

Both arcs have a positive time lag. Activity $floor$ represents the act of building the floor frame and activity $pillars$ represents the edification of the pillars for the next floor. The first temporal dependency ensures that that the pillars are built after the floor completion, while the second one conveys

[1]The repetition distance $\delta$ in embedded system design represents data packets that are produced and consumed by the activities. Such packets are called *tokens*.

that without the pillars of the previous floor a new floor frame cannot be built. Note that $\langle pillars, floor, 1, 1\rangle$ is an inter-repetition dependency.

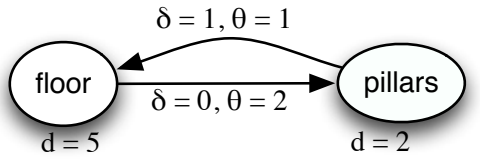


Figure 2.1: Project Graph Example

Based on Equation (2.4), we can formally define both the dependencies:

$$\langle floor, pillars, 2, 0\rangle \rightarrow start(pillars, \omega) \geq start(floor, \omega) + 5 + 2 \quad \forall \omega \in \mathbb{Z} \quad (2.5)$$

$$\langle pillars, floor, 1, 1\rangle \rightarrow start(floor, \omega) \geq start(pillars, \omega - 1) + 2 + 1 \quad \forall \omega \in \mathbb{Z} \quad (2.6)$$

We now proceed by providing a numeric example to clarify the mechanics of the temporal dependencies. Since a cyclic schedule is infinite, we can choose for convenience a reference repetition and start time. Fixing the start time of one activity does not compromise completeness since $\omega \in \mathbb{Z}$. Specifically, let us assume that $start(floor, 0) = 0$ (i.e. the workers immediately start to build the ground floor ). The following equations and Figure 2.2 show how the start times of different executions are (lower) bounded by temporal dependencies:

$$\begin{aligned} &\textbf{Step 1, } \omega = 0: \\ \langle floor, pillars, 2, 0\rangle \rightarrow \quad & start(j, 0) \geq start(floor, 0) + 7 \\ & start(pillars, 0) \geq 7 \end{aligned}$$

$$
\begin{aligned}
&\textbf{Step 2, } \omega = 1: \\
\langle pillars, floor, 1, 1\rangle \rightarrow \quad & start(floor, 1) \geq start(pillars, 0) + 3 \\
& start(floor, 1) \geq 7 + 3 \\
\langle floor, pillars, 2, 0\rangle \rightarrow \quad & start(pillars, 1) \geq start(floor, 1) + 7 \\
& start(pillars, 1) \geq 10 + 7 \\
&\textbf{Step 3, } \omega = 2: \\
\langle pillars, floor, 1, 1\rangle \rightarrow \quad & start(floor, 2) \geq start(pillars, 1) + 3 \\
& start(i, 2) \geq 17 + 3 \\
\langle floor, pillars, 2, 0\rangle \rightarrow \quad & start(pillars, 2) \geq start(floor, 2) + 7 \\
& start(pillars, 2) \geq 20 + 7 \\
& \ldots
\end{aligned}
$$

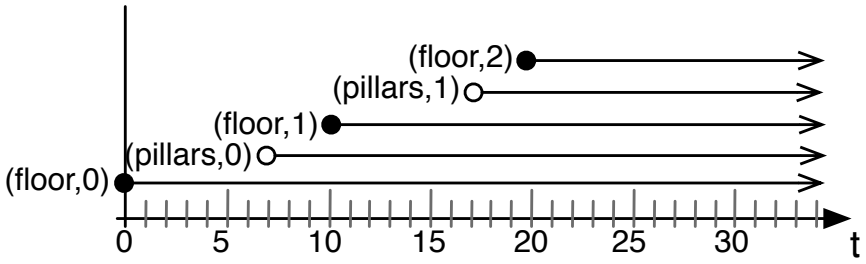


Figure 2.2: Precedence constraints in action in a modulo scheduling approach

### 2.1.3 Paths and Cycles

Let $\vec{e} = \langle i, j, \theta_{(i,j)}, \delta_{(i,j)}\rangle \in \mathbb{A}$ be an arc of the project graph. The arc $\vec{e}$ connects the activity $i$, called source $src(\vec{e})$, with the activity $j$, called sink $snk(\vec{e})$.

**Definition 5.** *A path $p^i_j$ in the project graph is a finite, nonempty sequence $(\vec{e_1}, \vec{e_2}, ..., \vec{e_m})$, where each $\vec{e}$ is a member of $\mathbb{A}$, $src(\vec{e_1}) = i$, $snk(\vec{e_m}) = j$, and $snk(\vec{e_1}) = src(\vec{e_2})$, $snk(\vec{e_2}) = src(\vec{e_3})$, ..., $snk(\vec{e_{m-1}}) = src(\vec{e_m})$.*

Moreover we say that $p_j^i$ is **directed** from $i$ to $j$. A project graph $\mathbb{G}$ is *strongly connected* (or *strictly connected*) if for each pair of distinct nodes $i, j$, there is a path directed from $i$ to $j$ (i.e. $p_j^i$) and there is a path directed from $j$ to $i$ (i.e. $p_i^j$). We say that an activity $i$ precedes $j$, $i \prec j$, if it exists a path directed from $i$ to $j$. A path that is directed from a node to itself is called a *cycle*.

#### 2.1.3.1 Iteration Bound

Let $\mathbb{C}$ be the set of all cycles of a project graph $\mathbb{G}$, $c(\mathbb{V})$ and $c(\mathbb{A})$ respectively the set of activities and edges that belong to a cycle $c \in \mathbb{C}$.

**Definition 6.** *The cycle bound* $\mathtt{CB}(c)$ *of a cycle* $c$ *is the minimum time required in a periodic schedule to execute all the activities in* $c$*. This is equal to:*

$$\mathtt{CB}(c) = \frac{\mathtt{Ex(c)}}{\mathtt{\Delta(c)}}$$

*where:*

- $\mathtt{Ex(c)} = \sum_{i \in \mathbb{V}(c)} d_i$ *is the sum of the durations of the activities in* $\mathbb{V}(c)$
- $\mathtt{\Delta(c)} = \sum_{(i,j) \in \mathbb{A}(c)} \delta_{(i,j)}$ *is the sum of the repetition distances of arcs in* $\mathbb{A}(c)$.

Note that the sum of the repetition distances must be strictly positive $\mathtt{\Delta(c)} > 0$, otherwise no feasible schedule exists. This can be intuitively checked since $\mathtt{CB(c)} \to \infty$ as $\mathtt{\Delta(c)} \to 0$: more details and formal proofs can be found in [43]. As a consequence, at least an arc in the cycle must have a positive $\delta$. A graph with $\mathtt{\Delta(c)} > 0 \quad \forall c \in \mathbb{C}$ is called *deadlock-free*.

**Definition 7.** *The Iteration Bound* $\mathtt{IB}$ *is the maximum of the cycle bounds:*

$$\mathtt{IB} = \max_{c \in \mathbb{C}} (\mathtt{CB}(c))$$

The iteration bound is related to the concept of critical path in traditional scheduling and to the concept of Maximum Cycle Mean (MCM) (and the relative Maximum Cycle Ratio, MCR) in performance analysis of synchronous

and asynchronous digital systems (including rate analysis of embedded systems) and in graph theory. Dasdan and Gupta [32] provide a comprehensive overview of algorithms for computing maximum cycle mean.

The iteration bound `IB` is the intrinsic lower bound on the iteration period; we can never achieve an iteration period $\lambda$ less than `IB`, even with infinite resources.

Periodic schedules are said to be *periodically optimal* if the iteration period $\lambda$ is the same as the iteration bound `IB`. If there are no resource restrictions, then a periodically optimal schedule is guaranteed to exist. This is not true once we add resource constraints to the problem. Further details can be found in [31, 42].

## 2.2 Cyclic Scheduling Strategies

There exist many different cyclic scheduling techniques. An in-depth survey of these techniques can be found in [14]. Three of the most widely used classes of cyclic scheduling techniques are:

- Static Techniques

  **(1)** static-time scheduling;

  **(2)** static-order scheduling;

- Dynamic Techniques

  **(3)** dynamic scheduling.

Static-time scheduling (called also *fully static* scheduling) techniques determine at design-time the start time of each activity executing. Static-order scheduling (called also *ordered-transaction* or *self-timed* scheduling) techniques determine at design-time only the ordering in which activities are executed. The actual start times are determined at run-time based on the availability of resources. The two static techniques are closely related to each other: (1) the ordering can be extracted from the start times of a static-time

schedule and (2) the start times can be deduces from the order of a static-order schedule. The third class of scheduling techniques, dynamic schedulers, do not take any decision at design-time. Both the order in which activities are executed as well as their start times are determined at run-time. It is therefore not practical for a dynamic schedule to make globally optimal scheduling decisions. A dynamic schedule will be forced to take locally optimal decisions. Static schedules on the other hand can take more information into account as these are constructed at design-time. Therefore, the performance (i.e. throughput) of a static schedule will typically be better than the performance of a dynamic schedule [14].

In this thesis we focus on static schedules. The disjunctive RA&CSP is tackled with a static-order scheduling technique, while the CRCSP with a static-time scheduling one.

Note that a schedule has infinite size in principle, because in a cyclic problem each activity executes an infinite number of times. However, since building an infinite schedule is impossible in practice, we should find a way to compute a more compact problem solution. Typically, (1) one wants to build a periodic static-time schedule (called also periodic schedule), i.e. a schedule where activities are executed regularly with a fixed period. Such an approach requires to specify only a start time for each activity, plus the period value. Periodic static-time schedules will be formally defined in Section 2.2.1. On the other side (2), a compact solution adopting a static-order scheduling technique is to define an order between the activities (e.g. posting precedences) such as an order between two activities constraints all their infinite executions. Periodic static-order schedules will be formally defined in Section 2.2.2.

### 2.2.1 Periodic Static-Time Schedule

A common way to compute a compact solution to the problem is to consider a schedule where activities are executed regularly with a fixed period (i.e. periodic schedule). Note that for a periodic schedule the fixed period (called

*iteration period*) is the same as the the maximum average inter-execution distance $\lambda$ from Definition (3). Formally:

**Definition 8.** *A periodic schedule is pair* $\langle \mathbb{L}, \lambda \rangle$ *where* $\lambda$ *is the iteration period and* $\mathbb{L}$ *is a vector containing the start times for the execution 0 of all the activities, i.e.:*

$$\mathbb{L}[i] = start(i, 0) \quad \forall i \in \mathbb{V}$$

*A periodic schedule obeys the following restriction:*

$$start(i, \omega) = start(i, 0) + \omega \cdot \lambda \quad \forall \omega \in \mathbb{Z} \quad \forall i \in \mathbb{V} \tag{2.7}$$

The iteration period of the schedule is defined as the distance between the start times of two consecutive executions of the same activity. We use the notation $\lambda$, since for a periodic schedule the iteration period is the same as the maximum average inter-execution distance. This can be checked by combining Equations (2.1) and (2.2) from Section 2.1 with Equation (2.7). Finally, the makespan for a periodic schedule is defined as the distance between the start of the first activity and the end of the last activity in every repetition. For this reason, it will be referred in the following as *schedule length*. To avoid confusion, we will use the same terminology also for other, non strictly periodic, approaches.

### Overlapped schedules

Cyclic schedules can be either *non-overlapped* or *overlapped*.

**Definition 9.** *A schedule is said to be* non-overlapped*, if the execution of any activity* $i$ *of repetition* $\omega + 1$*,* $start(i, \omega + 1)$*, starts after all activities of repetition* $\omega$ *have been executed (i.e. once repetition* $\omega$ *is over);*

$$start(i, \omega + 1) \geq start(j, \omega) + d_j \quad \forall i, j \in \mathbb{V} \tag{2.8}$$

**Definition 10.** *A schedule is* overlapped *if there exists at least one activity whose* $(\omega + 1) - th$ *repetition starts before the end of the* $\omega - th$ *repetition of all activities.*

$$\exists\ i, j \in \mathbb{V},\ i \neq j \quad | \ start(i, \omega + 1) < start(j, \omega) + d_j \tag{2.9}$$

Periodic schedules are said to be *periodically-optimal* if the iteration period $\lambda$ is the same as the iteration bound `IB`.

### Cyclic Static-Time Scheduling Techniques

Three main approaches have been proposed for the CRCSP. The first belongs to the non-overlapped class, while the other two exploit inter-repetition overlaps:

- The so called *blocked scheduling* approach (see Section 2.2.1.1) builds a schedule for a single repetition and assumes that the period is equal to the schedule length. As a consequence, consecutive repetitions are not allowed to overlap, with an obvious loss in terms of solution quality.
- The *unfolding* approach (see Section 2.2.1.2) schedules a number of consecutive repetitions and then repeats the block similarly to the previous technique. Unfolding often leads to higher quality schedules, but it also requires to solve problem instances with artificially increased size.
- The *modulo scheduling* approach (see Section 2.2.1.3) schedules a single repetition, which is however repeated every $\lambda$ time units. The value $\lambda$ is called the *modulus* and it is the same as the iteration period. By exploiting repetition overlaps, the modulus can be made much smaller than the schedule length, obtaining considerable speed-ups.

Figure 2.3 depicts a simple example of a CRCSP. All minimum time lags $\theta_{i,j}$ are assumed to be 0 and the repetition distance $\delta_{i,j}$ is 0 whenever not

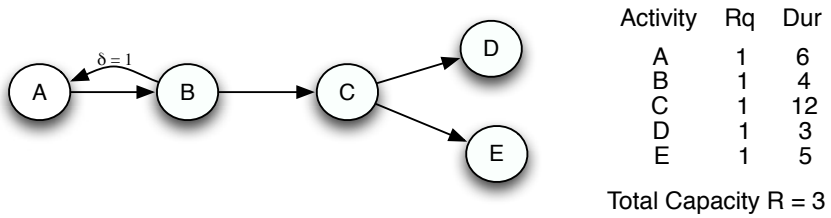


Figure 2.3: Annotated representation of a cyclic graph

explicitly mentioned. In the following subsections we use this instance to show how the three approaches described above work.

#### 2.2.1.1 Blocked Scheduling

Traditional static resource-constrained scheduling techniques for cyclic problems are non-overlapped [64],[18]. These methods optimize the performance of a single repetition of the project graph and then repeat the schedule periodically.

Figure 2.4 shows the optimal blocked schedule for the simple problem described in Figure 2.3. The output of the solution approach is the schedule (or *block*) identified by the black arrows, which is then repeated every iteration period $\lambda = 27$. Hence, the throughput is one over the length of the schedule $\text{THP} = \frac{1}{\lambda} = 0.037$. The horizontal dotted line represents the resource capacity: note that this schedule leaves most of the resource idle. With blocked schedules, this may happen quite frequently.

Note that a periodic schedule (as from Definition 8) *dominates* a blocked schedules.

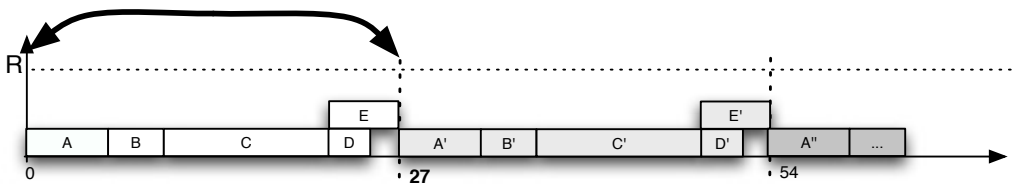


Figure 2.4: Blocked optimal schedule

#### 2.2.1.2 Unfolded Scheduling

Overlapped schedules exploit the repetitive nature of periodic schedules to achieve higher throughput. On this purpose, they have to take into account inter-repetition dependencies in addition to intra-repetition ones. The unfolding technique, presented in [83] and [84], consists in scheduling $u$ consecutive repetitions of the graph, where $u$ is referred to as *unfolding factor* (or as *blocking factor*). Then, the resulting schedule (say with total length $L$) is treated as a normal blocked schedule and repeated every $L$ time units. Note that the schedule length acts as a sort of period, but since $u$ repetitions are considered, the actual average inter-repetition distance is $\lambda = \frac{L}{u}$. In fact, $u$ executions are completed in $L$ time units.

Figures 2.5 and 2.6 show the optimal unfolded schedule for the simple problem depicted in Figure 2.3 with unfolding factor $u = 2$ and $u = 3$ respectively (with $u = 1$ we obtain the blocked schedule from Figure 2.4). The schedule is restarted after $L$ time units: for $u = 2$ the optimal $L$ is 35 and $\lambda = 17.5$, while for $u = 3$, $L = 47$ and $\lambda = 15.6$. The throughput THP is higher than in the blocked schedule, but the problem size (and consequently the search space) is bigger because it consists of $u$ different repetitions. Since the problem NP-complete, multiplying the number of activities by the unfolding factor leads to an exponential increase in the solution time.

Since the group of repetitions in the unfolded schedule is repeated in a blocked fashion, the unfolding approach may be incapable to optimally exploit inter-repetition overlaps to make the best use of the available resources.

Moreover, despite increasing $u$ from 2 to 3 led to a lower $\lambda$ value in our example, it is not true in general (as showed in [18]) that increasing

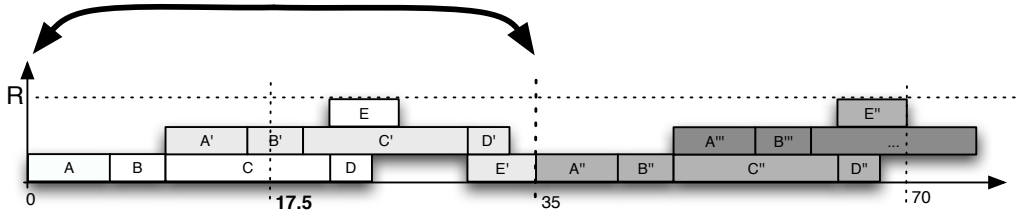


Figure 2.5: Unfolding optimal schedule with unfold factor 2

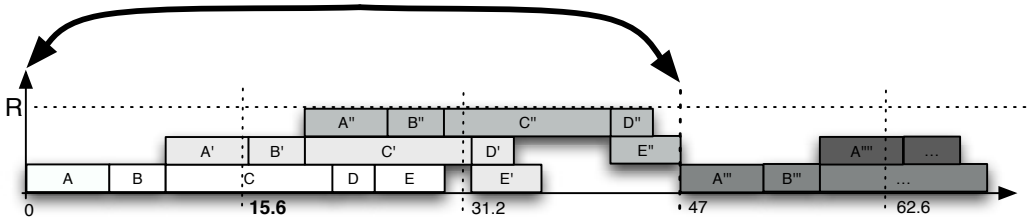


Figure 2.6: Unfolding optimal schedule with unfold factor 3

the unfolding factor leads to better schedules (note that the best unfolding factor $u$ cannot be trivially computed). Therefore an unfolded schedule do *not dominate* a periodic schedule.

On the other hand, it is known [48] that periodic schedules *are dominated* by K-periodic schedules[2] (i.e. periodic schedules for sequences of K iterations) in the presence of finite capacity resources. Note also that, as the unfolding technique takes into account $u$ iterations, there exists cases where the unfolded schedule is better than a (single-iteration) periodic schedule.

Hence, *no strict dominance* exists between periodic schedules and unfolded schedules.

An in-depth survey of traditional and unfolding cyclic scheduling techniques can be found in [14].

#### 2.2.1.3 Modulo Scheduling

The modulo scheduling method consists in finding a schedule for a single repetition, plus a modulus value $\lambda$ (usually lower than the whole schedule length). The schedule is repeated every $\lambda$ time units and the activities of consecutive repetitions, like in a pipelined system, overlap over time. This technique is the best in exploiting the available resources. Figure 2.7 shows the optimal modular schedule for the graph of Figure 2.3. The figure shows that, after an initial *transient phase*, the execution reaches a periodic phase where a repetition of the graph is completed every $\lambda = 10$ (corresponding to

[2]Note that an $u$-unfolded schedule is dominated too by a K-periodic schedule. In fact, considering an unfolding factor $u = K$, the resulting periodic schedule dominates the blocked (unfolded) one.

a throughput $\frac{1}{\lambda} = 0.1$).

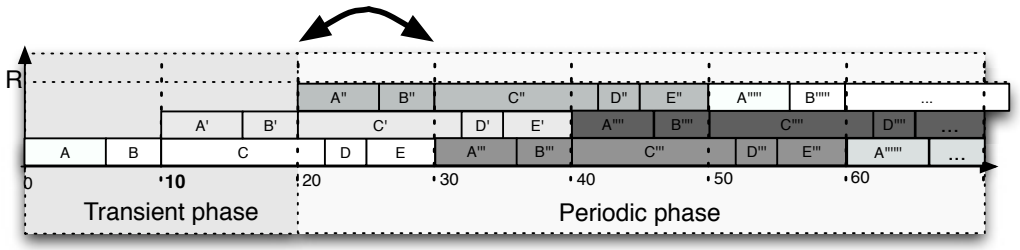


Figure 2.7: Modulo scheduling method optimal schedule

Note that a time window of length $\lambda$ (evidenced with the black arrows) contains the start time of exactly one execution of each activity. The activities may however appear with different execution numbers (i.e. $\omega$ values), following a well defined pattern. Such a collection of activity executions is called an *iteration* and is a fundamental concept for our method.

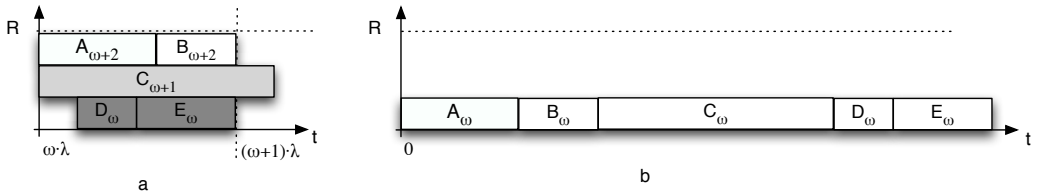


Figure 2.8: a) A single iteration. b) A single repetition.

Figure 2.8a depicts the schedule subpart corresponding to a specific iteration starting at a multiple of $\lambda$ (in the periodic phase). In the picture, with $x_\omega$ we refer to the $\omega$-th execution of activity $x$, i.e. to $(x, \omega)$. Note that an iteration contains a single execution of each activity and that not all the activities appear with the same $\omega$ values. For example, $(C, \omega + 1)$ and $(A, \omega + 2)$ are executing concurrently with $(D, \omega)$, i.e. the first execution of activity $D$ runs together with the second execution of C and the third execution of $A$.

One of the key ideas of this method is to focus on scheduling a single iteration (Figure 2.8a) instead of a single repetition with a large horizon (Figure 2.8b). In this context, it is convenient to restrict to iterations starting

at a multiple of the period $\lambda$, so that we can refer as the $\omega$-th iteration to the iteration starting at $\omega \cdot \lambda$. Further details will be given in Section 4.3.

### 2.2.2 Periodic Static-Order Schedule

Another common static scheduling technique is the static-order approach. This technique is often used in Embedded System Design [14], working with Digital Signal Processing (DSP) applications[3], and in problems where activities have variable duration times, see [69].

It consists in defining at design-time an order between the activities, delegating to the run-time the decision of the start times. A conventional policy is to consider an order involving executions $(i, 0)$ and $(j, 0)$ as an ordering decision between any other execution of the activities $i, j$. Therefore a static order schedule can be computed ordering the executions of a single repetition, namely the first repetition (i.e. $\omega = 0$).

Let $((i, \omega) \rightarrow (j, \omega))$ be an ordering decision between the executions $(i, \omega)$ and $(j, \omega)$. The ordering decision implies that the execution $(j, \omega)$ can start after $(i, \omega)$.

**Definition 11.** *A static-order schedule is a set $\mathbb{O}$ of ordering decision $((i, 0) \rightarrow (j, 0))$ between pairs of executions of the first repetition.*

Since a static-order schedule involves only execution of a single repetition, an ordering decision can be graphically expressed through a directed arc $\langle i, j, 0, 0 \rangle$ on the graph. As a consequence, a static-order schedule can be represented posting temporal dependencies in the problem graph. This methodology is used in the framework we devised for the Disjunctive RA&CSP. The proposed method models the effects of allocation and scheduling choices by means of graph modifications (see Chapter 3 for details).

Consider, for instance, activities $D$ and $E$ of the problem described in Figure 2.3 allocated on the same unary resource. Such

[3]DSP applications typically represent computations on an indefinitely long data sequence.

activities depend only from $C$, hence they could theoretically execute concurrently when $C$ ends. However they are competing for the same resource. Posting an ordering decision $((D, 0) \rightarrow (E, 0))$ between the activities solves the conflict creating a feasible schedule equivalent to the schedule represented in Figure 2.8b.

In this approach, the iteration period $\lambda$ corresponds to the iteration bound $IB^*$ of the modified graph (see [14]). Note that the iteration bound of the modified graph *is not smaller* than the iteration bound $IB$ of the original graph (i.e. $IB^* \geq IB$). In fact, adding arcs may create longer cycles and the iteration bound is the maximum of the cycle bounds.

#### Cyclic Static-Order Scheduling with Transient

As in cyclic problems activities execute an infinite number of times, any implementable static-order schedule has a periodic phase that is repeated indefinitely. Any practical static-order schedule should have a finite length. Otherwise, the schedule cannot be implemented. This periodic phase could be preceded by a transient phase (see the transient phase in the modulo scheduling technique, Section 2.2.1.3).

Static-order schedules that have a transient phase are called *periodic static-order schedules with transient*. This method usually computes better schedules[4], however, it needs the computation of two schedules: the transient phase schedule, executing only once, and the periodic schedule, repeated infinitely. Moreover the transient phase could be very long. In [97] the authors present an incomplete approach computing periodic static-order schedules with transient (more details in Section 3.3.2.2).

## 2.3 Problem Definition

Now we have all the notions to formally define the problems considered in this thesis.

[4]A schedule is better if it is able to achieve a higher throughput.

### 2.3.1 Disjunctive RA&CS Problem

The problem is defined as follows. Given

- a project graph $\mathbb{G} = \langle \mathbb{V}, \mathbb{A} \rangle$ with :
  - a set of activities $i \in \mathbb{V}$ having
    - fixed duration $d_i$
    - resource requirements $r_{i,k} \in \{0, 1\}$ for all resources $k \in \mathtt{R}$
  - a set of temporal dependencies $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle \in \mathbb{A}$ with $\theta_{(i,j)} = 0$
- a set $\mathtt{R}$ of unary resources.

a disjunctive RA&CS problem consists of finding a resource assignment for each activity and a static-order schedule such that all dependencies are consistent, no resource capacity is exceeded and the iteration period $\lambda$ is minimized (the throughput is maximized).

### 2.3.2 CRCS Problem

Given:

- a Project Graph $\mathbb{G} = \langle \mathbb{V}, \mathbb{A} \rangle$ with :
  - a set of activities $i \in \mathbb{V}$;
  - a set of temporal dependencies $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle \in \mathbb{A}$.
- A set $\mathtt{R}$ of limited capacity resources, where each resource $k \in \mathtt{R}$ has capacity $\mathrm{CAP}_k$.
- A fixed duration $d_i$ for each activity.
- A resource requirement $r_{i,k} \geq 0$ for each activity $i$ and resource $k$.

The CRCSP consists in finding a periodic schedule: i.e. an iteration period $\lambda$ and a feasible assignment of $start(i, 0)$ $\forall i \in \mathbb{V}$ such that all dependencies are consistent, no resource capacity is exceeded and the iteration period $\lambda$ is minimized.

# Chapter 3

# Solving the Disjunctive RA&CS Problem

In this chapter we propose an algorithmic framework for allocation and scheduling of DSP[1] applications on a target homogeneous multi-processor platform; the approach is complete, namely if a throughput requirement is specified, a feasible solution is guaranteed to be found if it exists; in general, the solver always finds the optimal solution if enough time is given. The method tackles the mapping[2] and scheduling problem as a whole, avoiding any sub-optimality due to decomposition.

Instances are modeled through Synchronous Data-Flow Graphs (SDFG). Section 3.2 presents a detailed description of Synchronous Data-Flow Graphs.

## 3.1 Embedded System Design

Smartphones, smartcameras, tablets, multimedia stations, the consumer electronic market is growing rapidly[3]. The universe of smart connected devices,

[1] Digital Signal Processing.

[2] In the Embedded System Design context the resource allocation problem is referred to as a mapping problem. The mapping problem usually consists in binding tasks to processors and memories.

[3] *Always On, Always Connected* at `http://www.accenture.com/SiteCollectionDocuments/PDF/Accenture_EHT_Research_2012_Consumer_Technology_Report.pdf` of Accenture (`http://www.accenture.com`)

including PCs, media tablets, and smartphones, saw shipments[4] of more than 916 mln units and revenues surpassing $489 bln dollars in 2011.

The transition in high-performance embedded computing from single CPU platforms with custom application-specific accelerators to programmable multi processor systems-on-chip (MPSoCs) is now a widely acknowledged fact [74, 54]. All leading hardware platform providers in high-volume applications areas such as networking, multimedia, high-definition digital TV and wireless base stations are now marketing MPSoC platforms with ten or more cores and are rapidly moving towards the hundred-cores landmark [1, 82, 9]. Large-scale parallel programming has therefore become a pivotal challenge well beyond the small-volume market of high-performance scientific computing. Virtually all key markets in data-intensive embedded computing are in desperate need of expressive programming abstractions and tools enabling programmers to take advantage of MPSoC architectures, while at the same time boosting productivity.

Stream computing based on a data-flow model of computation [63, 72] is viewed by many as one of the most promising programming paradigms for embedded multi-core computing. It matches well the data-processing dominated nature of many algorithms in the embedded computing domains of interest. It also offers convenient abstractions (synchronous data-flow graphs) that are at the same time understandable and manageable by programmers and amenable to automatic translation into efficient parallel executions on MPSoC target platforms. Our work addresses one of the key challenges in the development of programming tool-flow for stream computing, namely, the efficient mapping of synchronous data-flow graphs onto multi-core platforms. More in detail, our objective is to find allocations and schedules of SDFG nodes (also called activities or tasks) onto processors that meet throughput constraints or *maximize throughput*, which can be informally defined as the number of executions of a SDFG in a time unit. Meeting a throughput constraint is often the key requirement in many embedded application domains, such as digital television, multimedia streaming, etc.

[4]According to IDC (`http://idg.com/www/home.nsf`), *Embedded systems market to double by 2016* at `http://www.idc.com/getdoc.jsp?containerId=prUS23398412`

The problem of SDFG mapping onto multiple processors has been studied extensively in the past. However, the complex execution semantic of SDFGs on multiple processors has lead researchers to focus only on incomplete mapping algorithms based on decomposition [71, 97]. Allocation of activities onto processors is first obtained, using approximate cost functions such as workload balancing [71], and incomplete search algorithms. Then the throughput-feasible or throughput-maximal scheduling of activities on single processors is computed, using incomplete search search techniques such as list scheduling [63].

## 3.2 Background

### Synchronous Data-Flow Graphs

Synchronous Data-Flow Graphs (SDFGs) [65] are used to model periodic applications that must be bound to a Multi Processor System on Chip. They allow modeling of both pipelined streaming and cyclic dependencies between tasks. This model of computation represents data movements between activities through tokens[5] (i.e. dot on the arcs of the graph). To assess the performances of an application on a platform, one important parameter is the throughput. In the following we provide some preliminary notions on synchronous data flow graphs used in this thesis.

**Definition 12.** *An SDFG is a pair* $\langle \mathbb{V}, \mathbb{A} \rangle$ *consisting of a finite set* $\mathbb{V}$ *of activities (also called, nodes, tasks or actors) and a finite set* $\mathbb{A}$ *of* dependency arcs. *A dependency arc* $\vec{e} = \langle i, j, p, q, \delta \rangle$ *denotes a dependency of activity* $j$ *on* $i$*, with* $i, j \in \mathbb{V}$*. When* $i$ *executes, it produces* $p$ *tokens on* $\vec{e}$ *and when* $j$ *executes, it consumes* $q$ *tokens from* $\vec{e}$*. Arcs*[6] *may contain initial tokens* $\delta = tok(\vec{e}) = tok(i, j)$*.*

[5]The token in the SDF Model is equivalent to the repetition distances $\delta$ of the project graph presented in 2.

[6]Note that a dependency arc of a SDF graph could be physically represented as a FIFO Memory Buffer where data (i.e. tokens) are stored.

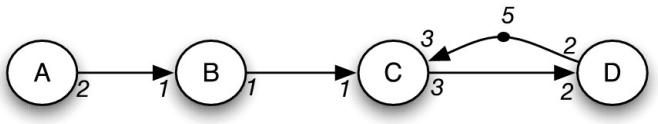


Figure 3.1: Synchronous Data-Flow Graph

An activity execution is defined in terms of firings. An essential property of SDFGs is that every time an activity fires it consumes a given and fixed amount of tokens from its input edges and produces a known and fixed amount of tokens on its output edges. These amounts are called *rates*. The rates determine how often activities have to fire w.r.t. each other such that the distribution of tokens over all edges is not changed. This property is captured by the repetition vector. Figure 3.1 represents a simple Synchronous Data-Flow graph with 4 nodes; the execution times are:
$A = 2$,
$B = 5$,
$C = 2$,
$D = 1$.

**Definition 13.** *A repetition vector of an SDFG* $= \langle \mathbb{V}, \mathbb{A} \rangle$ *is a function* $\gamma : \mathbb{V} \to \mathbb{N}$ *such that for every edge* $\vec{e} = (i, j, p, q, \delta) \in \mathbb{A}$ *from* $i \in \mathbb{V}$ *to* $j \in \mathbb{V}$, $p \cdot \gamma(i) = q \cdot \gamma(j)$. *A repetition vector* $\mathbb{Q}$ *is called non-trivial if* $\forall i \in \mathbb{V}$, $\gamma(i) > 0$.

The smallest non trivial repetition vector is usually referred to as *the* repetition vector. We say the SDFG completes an *sdf-iteration* whenever each activity $i$ has executed exactly $\gamma(i)$ times. We refer as *sdf-repetition* to each activity executing within an sdf-iteration. For instance, the repetition vector of the graph described in Figure 3.1 is $[1, 2, 2, 3]$

### 3.2.1 Throughput

Throughput is an important design constraint for embedded multimedia systems. The throughput of an SDFG refers to how often an activity produces tokens. To compute throughput, a notion of time must be associated with the execution of each activity (i.e., each activity has a duration $d_i$ also called *execution time*) and an execution scheme must be defined. We consider as execution scheme the **self timed execution** of activities: each activity executes as soon as all of its input data (i.e. tokens) are available (see [94] for details). In a real platform the self timed execution is implemented by assigning to each processor a sequence of activities to be fired in fixed order: the exact firing times are determined by synchronizing with other processors at run time.

Working with Synchronous Data-Flow models of computation, it becomes natural to adopt a scheduling strategy which defines only the allocation and let the run-time scheduler to decide the start times (i.e. the static-order scheduling technique, see Section 2.2.2).

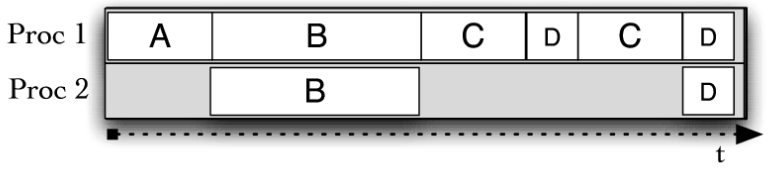


Figure 3.2: Single-iteration self-time execution

Considering the SDF graph in Figure 3.1, its single-iteration self-timed execution can be expressed by the Gantt chart of Figure 3.2. First activity $A$ is executed, it produces two tokens on $(A, B)$ since the *out*-rate of activity $A$ on the edge is 2. The tokens position is depicted in Figure 3.3a. The *in*-rate of activity $B$ on the same edge is 1; therefore $B$ can fire twice concurrently. After both executions of $B$ (see Figure 3.3b), the activity $C$ can start. Its execution consumes 1 token on $(B, C)$ and 3 on $(D, C)$

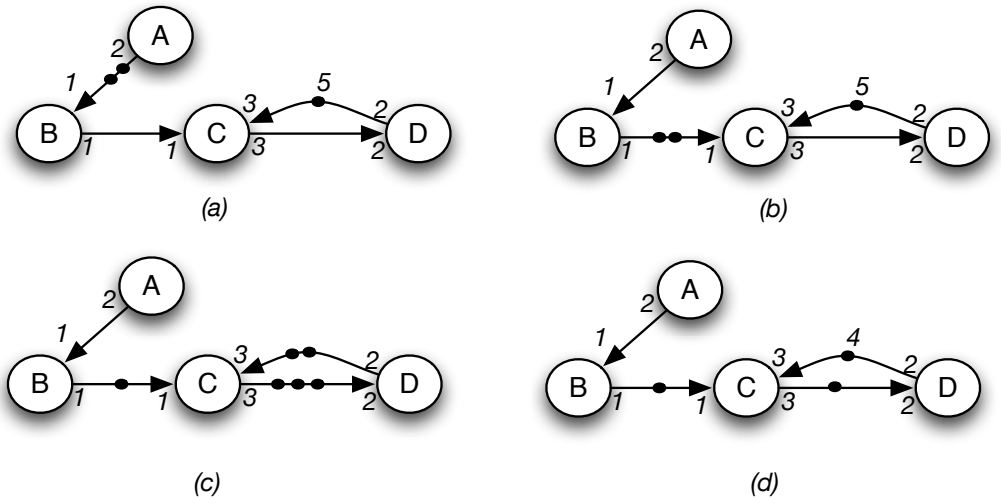


Figure 3.3: Synchronous Data-Flow Graph Execution Example

and produces 3 tokens on $(C, D)$ (see Figure 3.3c). Then, only the activity $D$ could fire, because the activity $C$ is constrained by the presence of only 2 tokens on $(D, C)$. Actor $D$ produces 2 tokens on $(D, C)$ (see Figure 3.3d) and enables the firing of $C$ whose execution enables the concurrent execution of two instances of $D$ that terminate the single-iteration self-timed execution of the graph.

### 3.2.2 Homogeneous Synchronous Data Flow Graphs

SDFGs in which all rates equal 1 are called Homogeneous Synchronous Data Flow Graphs (HSDFGs, [65]). Every SDFG $\mathrm{G} = \langle \mathbb{V}, \mathbb{A} \rangle$ can be converted to an equivalent HSDFG $\mathrm{GH} = \langle \overline{\mathbb{V}}, \overline{\mathbb{A}} \rangle$, by using the conversion algorithm in [14]. The transformation procedure is based on the repetition vector and produces an homogeneous graph that has a node for any sdf-repetition of each activity of the original SDF graph (i.e. $\gamma(i)$ nodes in $\overline{\mathbb{V}}$ for each activity $i \in \mathbb{V}$); as a consequence the homogeneous graph is usually larger than the related SDF.

In figure 3.4 we report the HSDFG corresponding to the SDFG in Figure 3.1. Note that, for example, activities $B_1$ and $B_2$ of the HSDFG corre-

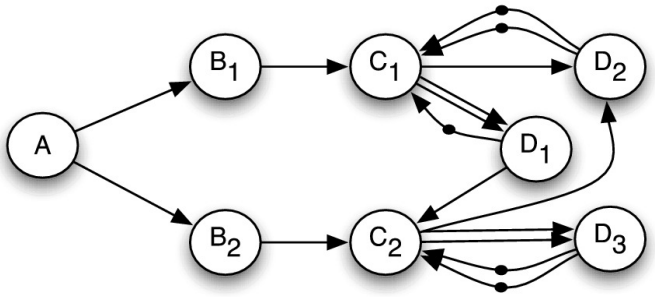


Figure 3.4: Homogeneous Synchronous Data-Flow Graph

spond to the activity $B$ of the SDFG that has a repetition vector $\gamma(B) = 2$. In the figure the (unary) rates are omitted.

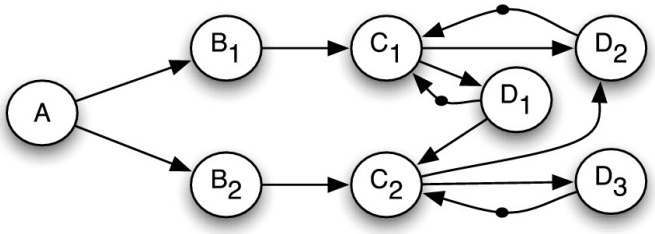


Figure 3.5: Filtered Homogeneous Synchronous Data-Flow Graph

Note that turning an SDFG into the equivalent homogeneous graph may produce multiple arcs between pairs of nodes (see edges from $D_2$ to $C_1$ in Figure 3.4). Therefore the homogeneous graph should be simplified before throughput computation removing multiple edges between two nodes. Figure 3.5 shows the filtered graph corresponding to the one in Figure 3.4.

## 3.3 Related Works

In this section we give an overview of the trends in Embedded System Design and presents the state-of-the-art of Data-Flow related approaches.

### 3.3.1 Application Domain

As the number of processors integrated on a single chip increases with the fast pace dictated by Moore's Law, multi-core systems-on-chip (MPSoCs) are becoming truly distributed systems at the micro-scale. A typical MPSoC [17, 56] features a number of computing tiles connected through a network-on-chip (NoC). A tile hosts a processor and a local memory hierarchy, and communicates with other tiles using communication services provided by the NoC interface. Processors are often highly optimized for domain-specific computation, with specialized instruction sets and support for vectorial data-parallel execution. While intra-tile parallelism is typically expressed through language intrinsics or automatically discovered by compilers, inter-tile communication is relatively expensive in time and power and it should be made explicit by the programmer. Thus, data-flow (streaming) models [14] , which express computations as collection of processes communicating through explicit channels with precisely defined production and consumption rules, match very well the nature of the underlying execution platforms [63].

From the application viewpoint, requirements for high performance and low power have increased at a breakneck speed in many embedded computing domains like wireless communication, imaging, audio and video processing, graphics, pushed by the demand for higher communication bandwidth, multimedia quality and realistic rendering. Applications in these areas are highly parallelizable and feature significant functional parallelism, which can effectively be expressed through a data-flow model of computation, where data is processed in (pipelined) sequences of computing stages with forks and loops to express alternatives and state.

As discussed above, technology and architectural evolution as well as application trends are motivating the use of data-flow programming in embed-

ded computing. For this reason increased research effort is being focused on developing methods and tools for efficiently mapping data-flow applications onto many-core MPSoC platforms [46]. The theoretical foundations of the data-flow model of computation were studied in the seventies and eighties [65], with the definition of several flavors of graph notations to formally and precisely express various classes of data-flow computational models, spanning the expressiveness vs. analyzability trade-off curve [14]. Synchronous data-flow (SDF) is one of the most widely used models (for details see Section 3.2), as it is sufficiently semantically rich to express practical computations, while being still analyzable with reasonable efficiency [64]. As of today, several commercial and academic programming environments are available for SDF application specification, analysis and mapping [46][14].

As most of the data-flow applications are subject to real-time constraints, a key problem that must be addressed by SDF mapping tools is throughput-constrained mapping (and/or throughput maximization). An informal definition of SDF execution throughput (see Section 3.5.1 for a formal definition) is the number of executions of an SDF graph in a unit of time. Applications usually come with throughput constraints, such as decoded frames per second, or processed polygons per second, and the key objective of a mapping tool is to find an allocation and scheduling of SDF nodes on computing tiles so that application throughput constraints are met. This is an NP-hard problem, and it is usually solved by sequential decomposition and incomplete search [71][97]. Additionally, even though SDF execution ultimately becomes periodic, the execution sequence within one period and the aperiodic initial transient can be very long. This greatly complicates throughput computation during the search of mapping and scheduling alternatives even for SDF graphs with a low number of nodes. Hence, complete search approaches were believed to be computationally intractable even for the simplest SDF instances.

### 3.3.2 Mapping and Scheduling Data-Flow Graphs

Data-Flow graphs are an extension of *computational graphs*, defined, for the first time, in 1966 by Karp and Miller [59]. Their studies focused on determinacy property and on termination conditions. The problem of mapping and scheduling task graphs has been widely studied (see, for instance, [80], [89] and [102]), but the limited descriptive power of task graphs as models of computation has lead to the development of graph models with a richer execution semantic.

The Synchronous Data-Flow Model of Computation (SDF MoC) has been proposed by Lee and Messerschmitt [65] to represent digital signal processing (DSP) applications. This Data-Flow MoC has been adopted in wide-ranging areas such as networking, multimedia, high-definition digital TV and wireless base stations; it can efficiently represent streaming applications such as mp3 playback [103], DAB channel decoding [15] and Software Defined Radio [73].

There exist many different scheduling techniques for SDFGs (see Section 2.2 for scheduling techniques details). Three of the most widely used classes of scheduling techniques for SDFGs are static-time scheduling, static-order scheduling and dynamic scheduling. Static-time scheduling techniques determine at design-time the start time of each actor firing on a processor. To implement such a schedule, the MP-SoC should have a global notion of time. This is hard to realize since MP-SoCs often contain multiple clock domains (i.e., many MP-SoCs use the GALS concept). Static-order scheduling techniques avoid this problem since they determine at design-time only the ordering in which actors are fired. The actual start times are determined at run-time based on the availability of tokens. The third class of scheduling techniques, dynamic schedulers, do not take any decision at design-time. Both the order in which actors are fired as well as their start times are determined at run-time. It is therefore not practical for a dynamic schedule to make globally optimal scheduling decisions. A dynamic schedule will be forced to take locally optimal decisions. Static-order schedules on the other hand can take more information into account as these are constructed at design-time. Therefore, the throughput of a static-order schedule will typically be better

than the throughput of a dynamic schedule [14]. A static-order schedule has also a better worst-case throughput than a dynamic schedule. Static-order schedules are therefore the most interesting class of schedules when mapping an SDFG onto an MP-SoC.

Much work has been published on scheduling of data-flow graphs with real-time requirements. Researchers have mostly focused on incomplete (also called heuristic) mapping algorithms for SDF allocation and scheduling (see [71, 97]). The motivation for the use of incomplete approaches is that both computing an optimal allocation and an optimal schedule are NP-hard [40].

Here we briefly describe state-of-the-art approaches to mapping and scheduling synchronous data-flow graphs that are classifiable onto two separate sets: complete and heuristic (incomplete) methods applied to Homogeneous SDFG, and heuristic methods applied directly on SDF graphs.

#### 3.3.2.1 HSDF Scheduling

The first class of approaches, pioneered by the group lead by E. Lee [94], and extensively explored by other researchers [14], can be summarized as follows. A SDFG specification is first checked for consistency, and its non-null iteration vector is computed. The SDFG is then transformed, using the algorithm, described in [14] into an Homogeneous SDF graph (HSDF[7]). The HSDFG is then mapped onto the target platform in two phases. First, an allocation of HSDFG nodes onto processors is computed, then a static-order schedule is found for each processor. The overall goal is to maximize throughput, given platform resource constraints. Unfortunately, throughput depends on *both* allocation and scheduling. However, the combination of possible mapping and scheduling decisions leads to an exponential blow-up of the solution space.

With the widespread diffusion of multi-core processors, scheduling and allocation of data-flow applications onto parallel computing platforms has received renewed interest. Kudlur et al. described in [61] an ILP that un-

[7]We recall that the Homogeneous SDF graph model is equivalent to the project graph model presented in Section 2.1. More details on the HSDF model in Section 3.2.

folds and partitions a stream application onto MPSoC architecture. Their approach consists in two steps: a fission and partitioning step, performed through ILP, to ensure work balancing, and then a stage assignment step wherein each activity is assigned to a pipeline stage for execution. An enhanced version of the same work was presented in [27] by Choi et al.

Chatha and co-authors have proposed two methods to support the compilation of streaming application on multi-core processors. The first, described in [24], uses fusion and fission operations to schedule streams onto (SPM based) multi-core processors while the second one, in [25], adopts a classic retiming technique. In both works the method is not complete, therefore the optimality is not guaranteed. They adopted the StreamIt language from MIT as the input specification (see [100] for details). StreamIt is an architecture-independent language with a synchronous data-flow semantic, supplied with an efficient compiler, described in [57] and in [93].

Ostler et al. devise, in [81], an ILP model for mapping streaming applications on multi-core platforms; the approach tackles acyclic applications, takes into account limited local memory capacity and allows throughput improvement via task fission. Communications are handled via double buffering, assuming exactly one DMA channel is dedicated to each processor. Within the specified assumptions the approach is optimal; it is important to observe that, since only acyclic SDFGs are considered, computing a feasible schedule is trivial once the mapping is specified.

Other approaches combine off-line/on-line scheduling techniques. For instance FlexStream, presented in [51], is a runtime adaptation system that dynamically re-maps an already partitioned stream graph according to the number of processors available for heterogeneous multi-core systems.

#### 3.3.2.2 SDF Scheduling

A different class of approaches [97] works directly upon SDF graphs using simulation techniques, without an explicit HSDFG transformation. This approach has the advantage to avoid the potential blow-up in the number of

nodes, with the disadvantage that if problem constraints are tight, incomplete approaches do not find any feasible solution. These approaches use a heuristic function to generate a promising allocation, and then compute the actual throughput by performing state-space exploration on the SDFG with allocation and scheduling information until a fixed point is reached.

Researchers from ST-Ericsson designed a scheduling strategy that allows a heterogeneous MPSoC to handle a dynamic mix of hard-real-time jobs which can start or stop independently. To solve this problem, a combination of Time Division-Multiplex (TDM) schedule and static-order of activities per processor is applied [73].

The incomplete approaches summarized above cannot give any proof of optimality, nor guarantee to find a feasible solution; actually, if the throughput requirement of the problem is tight, an incomplete solver is likely to fail. Our work aims at addressing this limitation, and proposes a complete search strategy which can compute max-throughput mappings for realistic-size instances. Our starting point is a HSDFG[8], which can be obtained from a SDFG by a pseudo-polynomial transformation [14]. We develop a CP-based solver which, given an architecture and an application described through a SDF graph, finds either the optimal or a feasible mapping and scheduling.

## 3.4 The Model

We devised a two-layer CP model: on one level the model features two sets of decision variables, respectively representing allocation and scheduling/ordering decisions; on the second level we have a set of graph description variables working directly on the graph by adding and removing arcs and tokens[9] as a consequence of the allocation and scheduling decisions. For this reason, the two models are linked via *channeling constraints*.

[8]As stated in Section 3.2, the Homogeneous SDFG is equivalent to the project graph presented in Chapter 2

[9]Note that the number of tokens corresponds to the $\delta$ value of the arcs, formally described in Section 2.1.2. In this section in order to simplify the formulation, an arc from $i$ to $j$ is formalized to as $(i, j)$ and $\delta_{i,j}$ its number of tokens.

As far as the first level is concerned, let $n$ be the number of activities in the input graph and let $p$ be the number of resources (i.e. processors in the platform), then the decision variables are:

$$\forall i = 0 \dots n-1 : \ \mathtt{P}_i \in [0..p-1] \tag{3.1}$$

$$\forall i = 0 \dots n-1 : \ \mathtt{Nx}_i \in [-1..n-1] \tag{3.2}$$

where $\mathtt{P}_i$ represents the resource allocated to activity $i$ and $\mathtt{Nx}_i$ represents the activity following activity $i$ if allocated on the same resource (the -1 value means that no activity follows).

$\mathtt{P}_i$ and $\mathtt{Nx}_i$ variables are subject to a set of constraints. First dependencies in the input graph cannot be violated: thus $i \prec j \Rightarrow \mathtt{Nx}_j \neq i$. Less intuitively, assuming that $i$ and $j$ (allocated on the same *unary* resource) cannot execute concurrently, the presence of an arc $(j, i)$ with $\delta_{j,i} = 1$ in the input graph implies $i$ to fire always *before* $j$, and therefore, $\mathtt{Nx}_j \neq i$.

Moreover, two nodes on the same resource, cannot have the same successor: $\mathtt{P}_i = \mathtt{P}_j \Rightarrow \mathtt{Nx}_i \neq \mathtt{Nx}_j$. Then, a node $i$ can be next of $j$ only if they are on the same resource: $\mathtt{P}_i \neq \mathtt{P}_j \Rightarrow \mathtt{Nx}_i \neq j$ and $\mathtt{Nx}_j \neq i$. The -1 value is given to the last node of each (non empty) resource:

$$\forall \ res : \ \sum_{i=0}^{n-1} (\mathtt{P}_i = res) > 0 \Rightarrow \sum_{i=0}^{n-1} [(\mathtt{P}_i = res) \times (\mathtt{Nx}_i = -1)] = 1 \tag{3.3}$$

Finally, the transitive closure on the activities running on a single resource is kept by posting an *ad hoc* constraint (based on the *nocycle* constraint [85]) on the related $\mathtt{Nx}$ variables.

Note that we consider the mapping platform as an ideal architecture without any communication cost or buffer requirement.

The second model, instead, considers the (dynamically changing) graph structure and defined decision variables on it. We define a matrix of binary variables $\mathtt{ARC}_{i,j} \in [0, 1]$ such that $\mathtt{ARC}_{i,j} = 1$ if and only if an arc from $i$ to $j$

exists. Existing arcs in the input graph result in some pre-filling of the ARC matrix, such that $\texttt{ARC}_{i,j} = 1$ for each arc $(i, j)$ in the original graph.

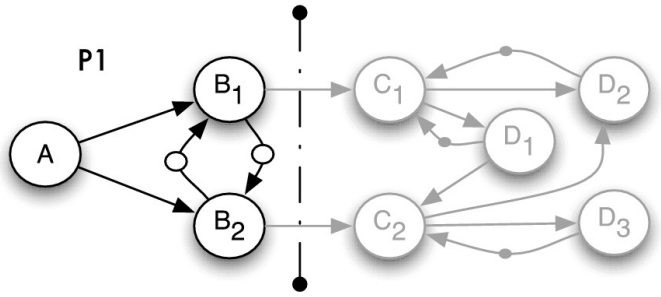


Figure 3.6: Concurrent task mapped on the same resource

Channeling constraints link the two models, i.e., allocation and scheduling decisions and graph description variables; first observe that token positioning is implicitly defined by the $\texttt{Nx}_i$ variables and is built on-line only at throughput computation time. As far as the $\texttt{P}_i$ variables are concerned, the relation with ARC variables depends on whether a path with no tokens exists in the original graph between two nodes $i, j$. As stated in Section 2.1.3 we write $i \prec j$ if such path exists; then, if $i \neq j$ and neither $i \prec j$ nor $j \prec i$ hold:

$$\texttt{P}_i = \texttt{P}_j \Rightarrow \texttt{ARC}_{i,j} + \texttt{ARC}_{ji} = 2 \tag{3.4}$$

Constraint (3.4) forces two arcs to be added, if two independent activities are allocated to the same resource (e.g. nodes $B_1$ and $B_2$ in Figure3.6).

If instead there is a path from $i$ to $j$ ($i \prec j$), then the following constraint is posted:

$$\left[(\texttt{P}_i = \texttt{P}_j) \wedge \sum_{k \prec i} (\texttt{P}_k = \texttt{P}_i) = 0 \wedge \sum_{j \prec k} (\texttt{P}_k = \texttt{P}_j) = 0\right] \Rightarrow \texttt{ARC}_{ji} = 1 \tag{3.5}$$

The above constraint completes dependency cycles: considering only activities on the same resource (first element in the constraint condition), if there is no activity before $i$ in the original graph (second element) and there is no

activity after $j$ in the original graph (third element), then close the loop, by adding an arc from $j$ to $i$. Figure 3.6 shows that, assuming an allocation of $A, B_1, C_1$ on the same resource, an arc with $\delta_{(C_1,A)} = 1$ is added from $C_1$ to $A$.

Finally, auto-cycles can be added to each node in a pre-processing step and are not considered here. Since we are dealing with a throughput bounded

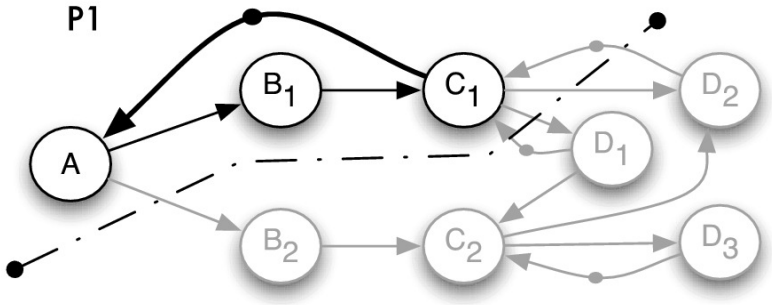


Figure 3.7: Pipelined task mapped on the same resource

application, we need a constraint computing the throughput depending on decisions taken during search. For this purpose we have defined a novel Throughput Constraint (see sec: 3.5.1) which is satisfied if and only if an allocation of `P` and `Nx` exists that defines an augmented graph with a throughput value higher than the current bound. The constraint is global and has the following signature:

$$thcst(\mathrm{TPUT}, [\mathtt{P}_{0..n-1}], [\mathtt{Nx}_{0..n-1}], [\mathtt{ARC}_{(0,0)..(n-1,n-1)}], d_{0..n-1})$$

where TPUT is a real valued variable representing the throughput, $[\mathtt{P}_{0..n-1}]$, $[\mathtt{Nx}_{0..n-1}]$ and $[\mathtt{ARC}_{(0,0)..(n-1,n-1)}]$ are defined as above, $d_{0..n-1}$ is a vector such that $d_i$ is the computation time of activity $i$.

Note that with this constraint, we can easily find also throughput maximal solutions (objective function $z = \max(\mathrm{TPUT})$), by iteratively solving a set of throughput bounded problems with increasing values of throughput.

### 3.4.1 Constraint Model

The complete constraint model is formalized as follows:

$$z = \max(\mathrm{TPUT}) \tag{3.6}$$

$$\sum_{i=0}^{n-1}(\mathtt{P}_i = res) > 0 \Rightarrow \sum_{i=0}^{n-1}[(\mathtt{P}_i = res) \times (\mathtt{Nx}_i = -1)] = 1 \quad \forall i\ , \forall\ res : 0..p-1 \tag{3.7}$$

$$[\neg(i \prec j) \wedge \neg(i \prec j) \wedge \mathtt{P}_i = \mathtt{P}_j] \Rightarrow \mathtt{ARC}_{i,j} + \mathtt{ARC}_{j,i} = 2 \quad \forall\ i, j\ , \quad i \neq j \tag{3.8}$$

$$\left[(i \prec j) \wedge (\mathtt{P}_i = \mathtt{P}_j) \wedge \sum_{k \prec i}(\mathtt{P}_k = \mathtt{P}_i) = 0 \wedge \sum_{j \prec k}(\mathtt{P}_k = \mathtt{P}_j) = 0\right] \Rightarrow \mathtt{ARC}_{j,i} = 1 \quad \forall\ i, j \tag{3.9}$$

$$i \prec j \Rightarrow \mathtt{Nx}_j \neq i \quad \forall\ i, j \tag{3.10}$$

$$[(\mathtt{P}_i = \mathtt{P}_j) \wedge (j, i) = 1 \wedge \delta_{j,i} > 1] \Rightarrow \mathtt{Nx}_j \neq i \quad \forall\ i, j \tag{3.11}$$

$$\mathtt{P}_i = \mathtt{P}_j \Rightarrow \mathtt{Nx}_i \neq \mathtt{Nx}_j \quad \forall\ i, j \tag{3.12}$$

$$\mathtt{P}_i \neq \mathtt{P}_j \Rightarrow \mathtt{Nx}_i \neq j \wedge \mathtt{Nx}_j \neq i \quad \forall\ i, j \tag{3.13}$$

$$thcst(\mathrm{TPUT}, [\mathtt{P}_{0..n-1}], [\mathtt{Nx}_{0..n-1}], [\mathtt{ARC}_{(0,0)..(n-1,n-1)}], W_{0..n-1}) \tag{3.14}$$

*Variables*

$\mathrm{TPUT} \in [0, \infty]$

$\mathtt{P}_i \in \{0..p-1\} \quad \forall i = 0 \ldots n-1$

$\mathtt{Nx}_i \in \{-1..n-1\} \quad \forall i = 0 \ldots n-1$

$$\mathtt{ARC}_{i,j} \in \{0,1\} \begin{cases} \mathtt{ARC}_{i,j} = 1 & \text{iff} \quad \text{exists } (i,j) \\ 0 & \text{otherwise} \end{cases}$$

#### 3.4.1.1 Communication Buffers and Latency

For the sake of simplicity, the model presented in this chapter is based on an ideal MPSoC architecture, where communication is considered as ideal (zero cost). However communication buffers and latencies can be modeled in different ways, depending on the target architecture. In this section we describe two approaches to model buffers and latencies for two widely adopted MPSoC architectural templates.

- Tightly-Coupled Shared-Memory Cluster Architecture (e.g. Platform

P2012 [12]): in this architecture all the processing units within a cluster share a fast multi-banked on-chip L1 data memory. The memory stores the buffers and the access and transfer cost (i.e. communication latency) is the same for each resource. In this case latencies time lags can be merged within the task execution times.
Let now $\omega$ be the bandwidth of the communication channel, $\widehat{L}$ be the latency of a single token communication. The latency $L$ of a communication depends on the bandwidth $\omega$ and the size of the transmission: i.e. the number of tokens $\delta$ the task produces.

$$L = \widehat{L} \cdot \frac{\delta}{\omega} \tag{3.15}$$

where $\omega$ has been normalized considering the size of a single token (e.g. when $\omega = 2$ the channel transmits two tokens concurrently). Hence the final execution time $\widehat{d_i}$ of a node $i$ should be $\widehat{d_i} = d_i + L_{in} + L_{out}$ where $L_{in}$ and $L_{out}$ are the sum of the latencies of the in-going and out-going communications, respectively. Furthermore, the memory capacity (L1 size) and the buffer requirements can be modeled[10] through a global cumulative constraint [7]. The constraint is satisfied iff, for each time instant, the sum of the buffer allocated does not exceed the total capacity of the memory.

- Non-Uniform Memory Access (NUMA) Architecture: the template in this scenario is based on a tile-based multiprocessor architecture (widely described in [33]) in which multiple tiles are connected by an interconnection network. Each tile contains a processor and a memory containing the communication buffers. The system has a Global Address Space, therefore the tasks and their communication buffers should be allocated as near as possible. Hence the model presented had to be drastically modified. In fact it should consider the buffer

[10] One of the advantages that the use of constraint programming (see Appendix A) has is that the definition of the model is loosely coupled with the search strategy adopted. Hence adding further constraints to existing models, not only is easily feasible but it could even help making, with the constraint propagation, search for a solution more efficient and more effective.

allocation problem and the impact of the allocation choices on the communication latencies. In this case latencies should be modeled through additional nodes with variable durations depending on the allocation of the buffers (e.g. see the approach in [90]) and each local memory capacity should be modeled though a cumulative constraint [7] (avoiding resource over-usage).

A trivial solution, in this scenarios could be to force the allocation of all the HSDFG nodes corresponding to repetitions of the original SDFG nodes on the same processor (thus in NUMA architectures buffers could be allocated locally). However the experiments show that, without this constrained hypothesis forcing the allocation, the search found much better solutions (see Section 3.7).

## 3.5 The Propagation

This section presents the filtering algorithm of the throughput constraint (see Section 3.5.1), its incremental improvement (see Section 3.5.2) and concludes describing several algorithmic optimizations (see Section 3.5.4)

### 3.5.1 Throughput Constraint

The relation between decision variables and the throughput value is captured in the proposed model by means of a novel global throughput constraint, whose signature is:

$$thcst(\mathrm{TPUT}, [\mathtt{P}_{0..n-1}], [\mathtt{Nx}_{0..n-1}], [\mathtt{ARC}_{(0,0)..(n-1,n-1)}], d_{0..n-1})$$

where TPUT is a real valued variable representing the throughput, $[\mathtt{P}_{0..n-1}]$, $[\mathtt{Nx}_{0..n-1}]$ and $[\mathtt{ARC}_{(0,0)..(n-1,n-1)}]$ are stated in Section 3.4, $W$ is a vector such that $d_i$ is the computation time of activity $i$.

We devised an algorithm consistently updating an upper bound on TPUT (this is sufficient for a throughput maximization problem).

Each time the graph is modified, by fixing an `ARC` variable or taking an ordering decision, the constraint receives a new description of the graph, and computes the throughput value over it.

During search the throughput variable is constrained to be within a lower and an upper bound. Initially the upper bound is set to the intrinsic iteration bound `IB` of the starting graph. This value always decreases during search. In fact, the application throughput depends on the inverse of the *longest* cycle whose value increases as search decisions are taken. On the other hand, the lower bound is set to the throughput requirement of the application, if any; in case we want to maximize the throughput value, the lower bound is updated with the best solution found so far. Since the optimal solution is found by iteratively improving feasible solutions, the lower bound increases during search.

At any time during the solution process, if the upper bound becomes lower than the lower bound, the search fails and backtracking is forced. In fact during the search the throughput bound values describe a monotonic decreasing function. Whenever allocation and ordering decisions are taken the graph is modified, adding arcs. Since the throughput depends on the *longest* cycle, its value can only decrease during the search.

As stated in Appendix A, global constraints comprise efficient filtering algorithms.

The filtering algorithm we propose extends the Maximum Cycle Mean (MCM) algorithm [53] and [32], which in turn is based on Karp's algorithm ([58]). The MCM algorithm is based on a recursive formula which computes, starting from a source node, the weight of each path (execution times of the considered nodes) of the graph. As soon as a cycle $c \in \mathbb{C}$ is found, the throughput $Th_c$ is computed. The final throughput value is the lowest found, corresponding to the weightiest cycle.

$$Th = \min_{c \in \mathbb{C}} Th_c \tag{3.16}$$

The algorithm is based on two three-dimensional matrices:

- $D_{(k,i,\vec{\delta})}$ that stores the weight of the path. In particular each element

$(k, i, \vec{\delta})$ is the maximum weight of a path of length $k$ from a node source $s$ to $i$; the number of tokens in the path is described with $\vec{\delta}$. If $D_{(3,2,1)} = 1.7$ means that at level $k = 3$ (i.e., three nodes far from the source $s$) there exists a path that connects $s$ to $j = 2$ with one token over its arcs, $\vec{\delta} = 1$; if no such path exists, then $D_{(k,i,\vec{\delta})} = -\infty$.

- $\Pi_{(k,i,\vec{\delta})}$ that saves the location of the predecessor of task $i$ at level $k$. In particular, such location consists of two coordinates: the index of the task and its token number; note that the predecessor level is $k-1$. For instance $\Pi_{(k,i,\vec{\delta})} = (3, 2)$ means that the activity $i$ at level $k$ has node 3 as predecessor (referred to as $idx(\Pi_{(k,i,\vec{\delta})})$); the number of tokens on the path from the source (referred to as $tok(\Pi_{(k,i,\vec{\delta})})$) is 2.

If $n$ is the number of the activities and $\textsc{Tot}_\Delta = \sum_{\forall (i,j) \in \mathbb{A}} \delta_{i,j}$ the number of tokens of the original graph, both $D$ and $\Pi$ are $(n + 1) \times n \times (\textsc{Tot}_\Delta + n)$ matrices.

The algorithm is divided in three phases:

#### 3.5.1.1 Step 1: building the input graph

The input for the throughput algorithm is a "minimal" graph built by adding arcs to the original project based on the current state of the model. More precisely, an arc is assumed to exist between activities $i$ and $j$ iff $\texttt{ARC}_{i,j} = 1$; unbound `ARC` variables are therefore treated as if they were set to 0. In the following we will often write $\texttt{ARC}_{i,j} = 1$ to mean an arc $(i, j)$ exists. Note that the computation of a lower bound for the throughput would require to fix values for unbound `ARC` variables as well.

Let $V_{i,j}[0, 1]$ (*Vertex matrix*) be a matrix which defines for each couple of activities $i, j$ the presence of an arc ($V_{i,j} = 1$ if $\texttt{ARC}_{i,j} = 1$ exists, 0 otherwise).

#### 3.5.1.2 Step 2: Token positioning

Next we construct a *dependency graph* DG with the same activities as the original project graph $\mathbb{G}$, and such that an arc $(i, j)$ exists in DG iff either an arc $(i, j)$ without tokens exists in $\mathbb{G}$ (detected since $\texttt{ARC}_{i,j} = 1$ and $\delta_{i,j} = 0$) or

$\texttt{Nx}_i = j$. Note that a DG graph is a Direct Acyclic Graph (DAG) augmented with the scheduling information of the partial solution.

A token matrix $\Delta$ is then built, according to the following rules:

$$\texttt{ARC}_{i,j} = 0 \;\Rightarrow\; \Delta_{i,j} = 0 \tag{3.17}$$

$$\texttt{ARC}_{i,j} = 1 \;\Rightarrow\; \begin{cases} \Delta_{i,j} = 0 \text{ if } i \prec^{DG} j \\ \Delta_{i,j} = \delta_{i,j} \text{ otherwise} \end{cases} \tag{3.18}$$

where we write $i \prec^{DG} j$ if there is path from $i$ to $j$ in DG. The rules above ensure the number of tokens is over-estimated, until all `Nx` and `P` are fixed. In the actual implementation, the dependency check is performed without building any graph, while the token matrix is actually stored in the constraint.

By considering the graph described in Figure 3.8 and an hypothetical allocation of activities $B_1, B_2$ on the same resource (see 3.9), the modified graph is showed in Figure 3.10. Assuming that in the DG graph both nodes are independent the resulting associated values of the token matrix[11] are $\Delta_{B_1,B_2} = \Delta_{B_2,B_1} = 1$. This is clearly an over-estimation of the number of tokens. Whenever an ordering decision is taken, for example $B_1 \prec B_2$, the token matrix is changed with the following values: $\Delta_{B_1,B_2} = 0$, $\Delta_{B_2,B_1} = 1$.

[11]Note that $\Delta_{i,j}$ represents the number of tokens of the arc of the modified graph, while $\delta_{i,j}$ represents the original value. $\Delta_{i,j}$ value is changed during search.

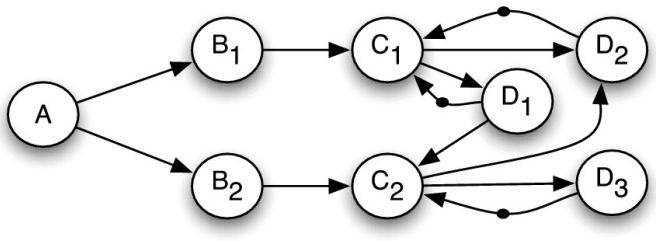


Figure 3.8: Filtered Homogeneous Synchronous Data-Flow Graph

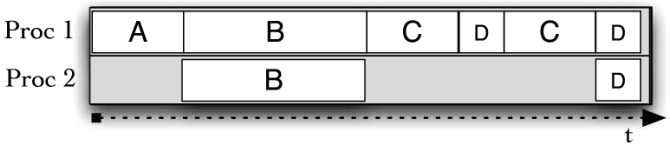


Figure 3.9: Single-iteration self-time execution

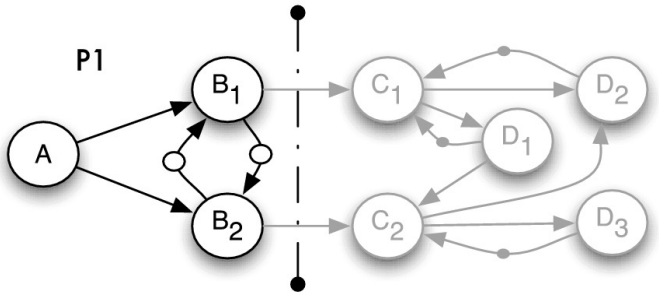


Figure 3.10: Concurrent task mapped on the same resource

#### 3.5.1.3 Step 3: Throughput computation

As stated in Section 2.1.3, for a project graph, its ideal[12] throughput equals the inverse of a quantity known as the iteration bound of the graph and denoted as `IB`

In [53] it is shown how to compute the iteration bound as the *maximum cycle mean* of an opportunely derived *delay graph*; Karp's algorithm [58] is used for the computation. In general *cycle mean* algorithms cannot be used to compute the throughput directly on a project graph. In fact, it is necessary to transform the graph into a weighted directed graph. Unfortunately it has been experimentally proven that this transformation is very time-consuming [44]. Here, we show that the transformation can be avoided by using proper data structure; this enables a *maximum cycle mean* algorithm to be used to compute the iteration bound directly on a graph. This is done by exploiting the third dimension (token dimension) of the matrices $D$ and $\Pi$ of the data structure, in the sense that they can store paths with different number of

[12]Without resource constraints.

tokens. Karp's algorithm works on a set of two-dimensional matrices; in fact, the $MCM$ algorithm considers a single token on each arc. We introduce a third matrix dimension to keep track of the number of tokens on the paths.

The basic idea is that, according to Karp's theorem, the critical loop constraining the iteration bound can be found by analysing cycles on the worst case $k$-length paths (e.g. the longest ones) starting from an arbitrary source. Since no cycle can involve more than $n$ nodes, considering $k$-length paths with $k$ up to $n$ is sufficient. Starting from a source node, we traverse the graph, storing for each node the critical path in the Matrices $D$ and $\Pi$. The critical path is the path with maximum cycle ratio; namely, assuming the same number of tokens, the path with greater execution time. Each time a cycle is detected, the throughput bound is updated. In order to simplify the notation in the algorithms the throughput variable TPUT is substituted with the period bound $\lambda' = \frac{1}{\text{TPUT}}$.

The pseudo code for the throughput computation is reported in Algorithm 1, where $A^+(i)$ denotes the set of direct successors of $i$. $Q^-$ is the set of nodes visited while $Q_k^+$ store, for each level $k$ the set of nodes to visit and their token level. Once the table is initialized, a source node $s$ is chosen. The experiments show that choosing a proper source node is non-trivial. We face the problem by reordering the activities with a heuristic function. The function is based on scores computed using the following expression:

$$score_i = \sum_{0 \leq j \leq task} Dep_{j,i} \tag{3.19}$$

where $Dep_{j,i}$ is 1 if there exists a path without tokens that connects $i$ to $j$, 0 otherwise. This structure can be easily computed from matrices $V$ and $\Delta$. Note that the choice has no influence on the correctness of the method, but a strong impact on its performance, hence choosing an arbitrary node is not recommended.

Next, the procedure is initialized by setting $D_{(0,s,0)}$ to 0 (line 4,5) and adding $s$ to the list of nodes to visit $Q_0^+$ (line 2). For each node $i$ in $Q$ each

**Algorithm 1:** Throughput computation - build $D$ table

**Data**: Let $s$ be the source node
**Data**: Let all $D_{(k,i,\vec{\delta})} = -\infty$, $\Pi_{(k,i,\vec{\delta})} = NIL$

```
1  begin
2      Q_0^+ = {(s, 0)}
3      Q^- = ∅
4      D_(0,s,0) = 0
5      Π_(0,s,0) = −1
6      for path level k ∈ n do
7          forall the (i, δ⃗) ∈ Q^+ do
8              forall the j ∈ A^+(i) do
9                  cycle = false
10                 δ^nx = δ⃗ + Δ_{i,j}
11                 currPos = (k, i, δ⃗)
12                 nextPos = (k + 1, j, δ^nx)
13                 Ex_j = D_nextPos
14                 Ex_i = D_currPos + d_j
15                 if Ex_i > Ex_j then
16                     Q^- = Q^- ∪ {i}
17                     D_nextPos = Ex_i
18                     Π_nextPos = (i, δ⃗)
19                     if Ex_i > λ′ then
20                         Find loops on level k
21                 if not cycle then
22                     Q^+_{k+1} = Q^+_{k+1} ∪ {(j, δ^nx)}
```

successor $j$ is considered (lines 7,8), and, if necessary, the corresponding cells in $D$ and $\Pi$ are updated to store the $k$-length path from $s$ to $j$ (lines 15 to 18). Once a cell is updated, if the weight of the path is higher than the current bound $\lambda'$, loops are detected as described in Algorithm 2. If the node $j$ does not close a cycle (line 21), it is added to the $Q^+_{k+1}$ queue and then we move to the next $k$ value. A single iteration of the algorithm is sufficient to compute the throughput of a strictly connected graph; otherwise, the process is repeated starting from the first never touched node, until no such node exists.

The loop finding procedure (Algorithm 2) is started when a cell in $D$ at

a specific level (let this be $k$) is updated. The algorithm moves backward along the predecessor chain ($\Pi(k+1, j, \delta^{nx})$ is the predecessor of current node) until a second occurrence of the starting node $j$ is detected ($a' = j$ in line 5) and a cycle is found. If this loop constrains the iteration period more than the last one found so far (line 11), this is set as critical cycle. The algorithm also stops when the start of $D$ is reached (in line 4).

**Algorithm 2:** Throughput computation - finding loops

**Data**: Let $i$ the node considered and $\vec{\delta}$ its tokens lvl
**Data**: Let $j$ the successor and $\delta^{nx} = \vec{\delta} + \Delta_{i,j}$ its tokens lvl
**Data**: Let $D_{nextPos}$ the cell updated, $nextPos = (k+1, j, \delta^{nx})$

```
1  begin
2      define a' = i
3      define δ' = δ^nx;
4      for path level z = k to 1 do
5          if j == Π_{z,a',δ'} then
6              define Π' = Π_{z,a',δ'}
7              define backPos = (z − 1, idx(Π'), tok(Π'))
8              define Ex_Thp = D_nextPos − D_backPos
9              define Δ_Thp = δ^nx − tok(Π')
10             cycle = true
11             if Ex_Thp / Δ_Thp > λ' then
12                 λ' = Ex_Thp / Δ_Thp
13             return;
14         define temp = a'
15         a' = idx(Π_{z,temp,δ'})
16         δ' = tok(Π_{z,temp,δ'})
```

#### 3.5.1.4 Example

Figures 3.12, 3.13, 3.14 and 3.15 represent matrices $D$ and $\Pi$ with regard to the sub-graph composed by activities $C$ and $D$ (5 nodes) of Figure3.5 with execution time respectively 2 and 1. The sub-graph is reported in Figure 3.11A. Assuming the source node is $C_1$, Figures 3.12 and 3.13 report respectively the sub-matrices $D_{i,j,0}$ and $\Pi_{i,j,0}$ (0 tokens), while 3.14 and 3.15

refer to the sub-matrices $D_{i,j,1}$ and $\Pi_{i,j,1}$ (with one token). Node $C_1$ has two out-going arcs: $(C_1, D_1)$ and $(C_1, D_2)$ stored in the matrix (Figure 3.12 and consequently in Figure 3.13) in cell $D_{1,2,0}$ and $D_{1,3,0}$. Let us now consider level $k = 1$: the only non-negative entries are the ones for $D_1$ and $D_2$. $D_1$ has an out-going arc that enters in $C_1$, that is stored in the cell $D_{2,0,1}$ (see Figure 3.14 and 3.15); at run-time when $C_1$ entry is processed, the "find loop" procedure detects the cycle $C_1 \rightarrow D_1 \rightarrow C_1$. The path is deduced by using the information stored in matrix $\Pi$: the cell $\Pi_{2,0,1} = 2, 0$ (activity $C_1$) refers to cell $D_{1,2,0} = 0, 0$ (activity $D_1$) that points to cell $D_{0,0,0}$ (activity $C_1$ again). By finding a loop, the algorithm infers a new bound over the throughput, namely one over the sum of the execution times: $\frac{1}{2+1} = 0,333$; the value is computed based on the starting and ending cell: $\frac{tok(D_{2,0,1}) - tok(D_{0,0,0})}{D_{2,0,1} - D_{0,0,0}}$.

Then the algorithm can proceed by considering arcs $(D_1, C_2)$ and $(D_2, C_1)$: the former updates the cell $D_{2,2,0} = 3$ while the latter updates the cell $D_{2,0,1}$ with value 3. However, since the current value of the same cell is 3, no change is performed. This means that there exist two different paths (namely $C_1 \rightarrow D_1 \rightarrow C_1$ and $C_1 \rightarrow D_2 \rightarrow C_1$) with the same length (2 steps) that connect the source activity with the same end node; since they have the same weight (computation time), they are equivalent, and only one is stored. The algorithm, then, computes all the remaining paths of the sub-graph considered and finds, as expected, the iteration bound; this corresponds to the cycle $(C_1 \rightarrow D_1 \rightarrow C_2 \rightarrow D_2 \rightarrow C_1)$ that refers to the throughput of the graph, that is $\frac{1}{6}$ $(\frac{tok(D_{4,0,1}) - tok(D_{0,0,0})}{D_{4,0,1} - D_{0,0,0}})$.

Note that by finding new longer loops, the upper bound always decreases;

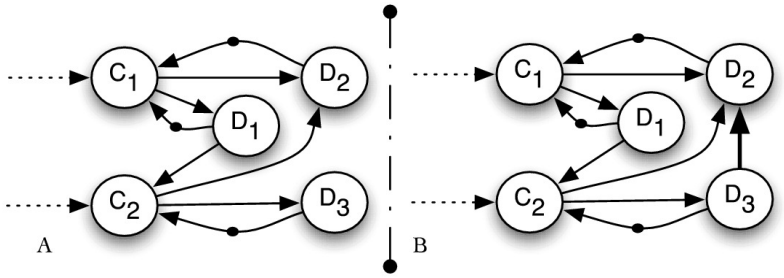


Figure 3.11: Sub-graphs related to Figure3.4

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| $D_{i,j,0}$ | 0 | 1 | 2 | 3 | 4 |
| 0 | **0** | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | **2** | 2 | -∞ |
| 2 | -∞ | **3** | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | **5** | 5 |
| 4 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 5 | -∞ | -∞ | -∞ | -∞ | -∞ |

Figure 3.12: Matrix $D_{i,j,0}$

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| Π | 0 | 1 | 2 | 3 | 4 |
| 0 | **-1** | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | **0,0** | 0,0 | -∞ |
| 2 | -∞ | **2,0** | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | **1,0** | 1,0 |
| 4 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 5 | -∞ | -∞ | -∞ | -∞ | -∞ |

Figure 3.13: Matrix $\Pi_{i,j,0}$

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| $D_{i,j,1}$ | 0 | 1 | 2 | 3 | 4 |
| 0 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 2 | 3 | -∞ | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 4 | **6** | 6 | -∞ | -∞ | -∞ |
| 5 | -∞ | -∞ | -∞ | -∞ | -∞ |

Figure 3.14: Matrix $D_{i,j,1}$

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| Π | 0 | 1 | 2 | 3 | 4 |
| 0 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 2 | 2,0 | -∞ | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 4 | **3,0** | 4,0 | -∞ | -∞ | -∞ |
| 5 | -∞ | -∞ | -∞ | -∞ | -∞ |

Figure 3.15: Matrix $\Pi_{i,j,1}$

hence, if at any step a cycle is found such that the resulting throughput is lower than the minimum value of the TPUT variable, then the constraint fails. Moreover, it is easy to prove that no more than 1 token can be collected by traversing a sequence of nodes on a single resource: the filtering algorithm exploits this property to improve the computed bound at early stages of the search, where the number of tokens is strongly overestimated (see Section 3.5.4).

Although the throughput computation is rather efficient, experimental tests show that its computational time takes more than the 70% of the total search time. In fact, every time the constraint is considered, it has to recompute the throughput on the entire modified graph. Therefore, we propose an incremental version of the constraint that avoids the recomputation of the throughput starting from scratch.

### 3.5.2 Incremental Algorithm

In this section we describe an incremental algorithm that enables to achieve over one order of magnitude speed up w.r.t. the non incremental version (described in Section 3.5.1), therefore increasing scalability and enabling the solution of harder and larger problems.

Note that the state of the constraint, referred to as $\Upsilon$, is defined by the data structures $\Upsilon \equiv < D, \Pi, V, \Delta >$. During search such data structures are modified at each search node and restored on backtracking. In detail, the data structure contains:

- Matrix $D_{k,i,\vec{\delta}}$: it stores the maximum weight of the $k$-arc path with $\vec{\delta}$ tokens from a source node $s$ to activity $i$.
- Matrix $\Pi_{k,i,\vec{\delta}}$: it stores the predecessor of the corresponding element of $D$.
- Matrix $V_{i,j}[0,1]$ (*Vertex matrix*) which defines for each couple of activities the presence of an arc ($V_{i,j} = 1$ if the arc $(i,j)$ exists, 0 otherwise)
- Matrix $\Delta_{i,j}[0, \inf]$(*Token matrix*) which defines the number of tokens on the arc of the path that connects $i$ to $j$.
  Clearly, $\Delta_{i,j} > 0$ only if $V_{i,j} = 1$, that is one or more tokens can exist between two nodes if and only if there is a corresponding arc.
- Period value of the longest cycle.

At the root node, the data structures are initialized from the original graph, getting state $\Upsilon_0$. The iteration bound of the graph is computed and used to shrink the throughput variable domain. At every search node, the state $\Upsilon$ and the throughput value are updated on the basis of graph modifications.

In particular, during search the graph is modified either by

- Adding arcs (arc append operation)
- Adding tokens (token append operation)
- Removing tokens (token remove operation)

Edges are removed only in backtracking. Therefore they are not considered as possible graph modification. Edges and tokens are added (tokens are also removed) in the graph for ordering the execution between activities allocated on the same processing element, as explained in Section 3.4.

At each invocation of the constraint, a new state $\Upsilon_{new}$ is computed starting from the previous one $\Upsilon_{old}$; the procedure requires one to know the current (modified) graph $G$, described by its *Vertex* and *Token matrices*. The update procedure consists of two main phases:

- **Gathering changes**: in this phase the current graph structure is compared to the previous one. Differences are stored in a proper data structure called UPDATES consisting of a set of dynamic queues UPDATES($k$) (one for each level in the $D$ matrix but the last one). Each queue stores triples $(i, j, \vec{\delta})$, where $i$ and $j$ are respectively the source and the destination nodes of the arc $(i, j)$ to be recomputed and $\vec{\delta}$ is the number of tokens collected along the path to $i$. Note that, joining triples $(i, j, \vec{\delta})$ and the index $k$ of the structure $UPDATES$, we compute the coordinates, in $D$ and $\Pi$, of the source and destination cells: in fact, $(k, i, \vec{\delta})$ refers to the starting node while $(k+1, j, \vec{\delta}+\Delta_{i,j})$ is the destination node.

- **Updating the values**: in this phase, arcs in the $UPDATES(k)$ queues are processed and the corresponding elements of matrices $D$ and $\Pi$ are re-computed, possibly identifying new cycles.

In the following, we describe in detail the algorithmic steps performed in each phase for the three possible types of graph modifications (arc append, token append, token remove) and the update phase.

#### 3.5.2.1 Gathering changes for an Arc Append Operation

Let $\vec{e} = (i, j)$ be an arc added from node $i$ to $j$ (see Algorithm 3) . Intuitively, adding an arc creates new paths containing node $i$; such paths may possibly cover (in terms of weight) existing ones and thus update the $D$ matrix cells

**Algorithm 3:** Arc Append Operation

```
   Data: Let e⃗ = (i, j) be an arc appended from node i to j
   Data: Let n be the number of the graph nodes
   Data: Let Γ be the sum of the original graph tokens
1  begin
2      for path level k in [1..n] and token level δ⃗ in [1..Γ + n] do
3          if D_{k,i,δ⃗} ≥ 0 then
4              UPDATES(k) → push(i, j, δ⃗);
```

referring to $i$. In this step we want to collect all matrix cells that need to be modified.

We remind that a path crossing node $i$ at level $(k, i, \vec{\delta})$, necessarily has $D_{(k,i,\vec{\delta})} \geq 0$, since $D_{(k,i,\vec{\delta})}$ is the maximum weight of a path of length $k$ from a source node $s$ to $i$. Therefore, we should identify in the matrix $D$ all the elements with $D_{k,i,\vec{\delta}} \geq 0$ (line 3) and insert the triple $(i, j, \vec{\delta})$ into $UPDATES(k)$ (line 4); this will trigger a re-computation of cells $D_{(k+1),j,\vec{\delta}+\Delta_{i,j}}$ in the update phase.

#### 3.5.2.2 Gathering changes for a Token Append Operation

Let $\vec{e} = (i, j)$ be the arc where we add a token. If the modification involves the insertion of both one arc and one token, the token modification is not considered and the only procedure run is that for the arc append. Otherwise, if the arc already exists (see Algorithm 4), the added token results in the modification of an *existent* path; the modified path may (a) cover other paths in $D$ and (b) uncover previously covered ones.

Detecting situation (a) requires to process the arc $\vec{e}$ in exactly the same fashion as Section 3.5.2.1. The only difference is that, for each cell $D_{k,i,\vec{\delta}} \geq 0$ (line 3) the triple $(i, j, \vec{\delta})$ (line 4) will trigger a re-computation of $D_{(k+1),j,(\vec{\delta}+1)}$ in the update phase.

Detecting whether the modified path uncovers existing ones (situation (b)) deserves a more detailed explanation. In particular, each cell in $D_{k,i,\vec{\delta}} \geq 0$ corresponds to a path with $\vec{\delta}$ accumulated tokens and including node $i$.

**Algorithm 4:** Token Append Operation

**Data**: Let $\vec{e} = (i, j)$ be an arc appended from node $i$ to $j$
**Data**: Let $n$ be the number of the graph nodes
**Data**: Let $\Gamma$ be the sum of the original graph tokens

```
1  begin
2      for level k in 1..n and level δ⃗ in 1..Γ + n do
3          if D_{k,i,δ⃗} ≥ 0 then
4              UPDATES(k) → push(i, j, δ⃗);
5              if idx(Π_{k+1,j,δ⃗}) = i then
6                  for activity i' in n do
7                      if (i' ≠ i) && (i' ≠ j) && (V_{i',j} = 1) then
8                          for token level δ' ≤ δ⃗ do
9                              if (D_{k,i',δ'} ≥ 0) && (δ' + Δ_{i',j} = δ⃗) then
10                                 UPDATES(k) → push(i', j, δ');
```

The addition of the new token uncovers other paths in $D$ if, at the next level $k + 1$:

1. there is a node $j$ having $i$ as predecessor
2. the node $i$ is the predecessor of $j$ on a path with $\vec{\delta}$ accumulated tokens, as the arc $\vec{e} = (i, j)$ previously had no token.

Formally, a re-computation of the cell $D_{(k+1),j,\vec{\delta}}$ (referring to the $j$ node) is required if $\Pi_{(k+1),j,\vec{\delta}} = i$ (line 5).

Since node $j$ has *lost* its former predecessor $i$, performing the update requires to consider all paths ending in $j$ at level $k + 1$. In practice this is done by re-considering all arcs from nodes $i'$ to $j$ (lines 6-8), such that at level $k$ it holds $D_{k,i',\delta'} \geq 0$ (i.e. they are part of a path at level $k$). Hence, we have to append into $UPDATES(k)$ all triples $(i', j, \delta')$ such that $D_{k,i',\delta'} > 0$ (for every $\delta' \leq \vec{\delta}$) (lines 9,10); this will trigger the re-computation of $D_{(k+1),j,\vec{\delta}}$ in the update phase.

**Algorithm 5:** Token Remove Operation

**Data**: Let $n$ be the number of the nodes
**Data**: Let $\Gamma$ be the sum of the original tokens
**Data**: Let $\vec{e} = (i, j)$ be an arc appended from node $i$ to $j$

```
1  begin
2      for level k in 1..n do
3          if D_{k,i,δ⃗} ≥ 0 then
4              UPDATES(k) → push(i, j, δ⃗);
5              if Π_{(k+1),j,(δ⃗+1)} = i then
6                  for activity i' in n do
7                      if (i' ≠ i) && (i' ≠ j) && (V_{i',j} = 1) then
8                          for token level δ' ≤ δ⃗ do
9                              if (D_{k,i',δ'} ≥ 0) && (δ' + Δ_{i',j} = δ⃗) then
10                                 UPDATES(k) → push(i', j, δ');
```

#### 3.5.2.3 Gathering changes for a Token Remove Operation

This is the dual of the previous case (see Algorithm 5). Simply at point (a) one has to recompute cell $D_{(k+1),j,\vec{\delta}}$ instead of $D_{(k+1),j,(\vec{\delta}+1)}$. At point (b), if $\Pi_{(k+1),j,(\vec{\delta}+1)} = i$ (note the $\vec{\delta}+1$ index), then cell $D_{(k+1),j,(\vec{\delta}+1)}$ needs to be recomputed (line 9); this requires to reconsider arcs $(u', j)$ for each $D_{k,u',\delta'} \geq 0$ (with $\delta' \leq \vec{\delta}$) (lines 10-14).

### 3.5.3 Updating the Values of $D_{k,i,\vec{\delta}}$

In this phase the algorithm processes the $D$ matrix, by increasing values of the $k$ index. At each level $k$, all triples in $UPDATES(k)$ are extracted; based on the $(i, j, \vec{\delta})$ values in the triple, the proper cell of the $D$ matrix is reconsidered (namely $D_{(k+1),j,(\vec{\delta}+\delta_{i,j})}$). If the computed value is higher that the current value of the cell, the $D$ and $\Pi$ matrices are updated if

$$D_{(k+1),j,(\vec{\delta}+\delta_{i,j})} < D_{k,i,\vec{\delta}} + d_j \tag{3.20}$$

**Algorithm 6:** Updating $D_{k,i,\vec{\delta}}$ Phase

**Data**: Let $n$ be the number of the nodes
**Data**: Let $d_i$ be the execution time of activities i.
**Data**: Let $\lambda'$ be the best bound on the period

```
1  begin
2      forall the path level k in n do
3          forall the triple (i, j, δ⃗) in UPDATES(k) do
4              if D_(k+1,j,δ⃗+Δ_i,j) < D_(k,i,δ⃗) + d_j then
5                  Q⁻ = Q⁻ ∪ {i}
6                  D_(k+1,j,δ⃗+Δ_i,j) = D_(k,i,δ⃗) + d_j
7                  Π_(k+1,j,δ⃗+Δ_i,j) = (i, δ⃗)
8                  if D_(k,i,δ⃗) + d_j > λ' then
9                      Find loops on level k
10             if k < n − 1 then
11                 forall the j' ∈ A⁺(j) do
12                     UPDATES(k) → push(i, j, δ⃗);
```

where $d_j$ is the execution time of activities $j$ and $\Delta_{i,j}$ is the number of tokens of the arc $(i, j)$.

Next, the performed update has to be propagated recursively: this is done by inserting into $UPDATES(k+1)$ a triple $(j, j', \vec{\delta} + \delta_{i,j})$ for each outgoing arc having $j$ as source (successors). During this phase, new and weightier cycles can be found. The weightiest one is the critical path that impacts the throughput value of the graph.

For instance consider the sub-graph shown in Figure 3.11A. The current state of the matrices $D$ and $\Pi$ is described in Figures 3.12, 3.13, 3.14 and 3.15. Assume now that the solver modifies the graph adding the arc ($D_3$,$D_2$) as reported in Figure 3.11B. When *gathering changes*, the incremental algorithm detects that the activities $D_3$ (the source of the modification) has been considered only in the cell $D_{(3,4,0)}$: as consequence the triple $(4, 3, 0)$,

that stands for $(D_3, D_2, 0)$, is pushed in $UPDATES(3)$ .
Next, the triple is extracted and evaluated in the *updating phase*: note that the arc $(D_3,D_2)$ now "points" to the cell $D_{(4,3,0)} = -\infty$. Then, inequality 3.20 is checked ($-\infty < D_{(3,4,0)} + d_{D_3} = 5 + 1 = 6$, with $d_{D_3}$ execution time of $D_3$) and the cells $D_{(4,3,0)}$, and $\Pi_{(4,3,0)}$ are updated with values $D_{(4,3,0)} = 6$ and $\Pi_{(4,3,0)} = (4, 0)$. Since some cell has been updated, the algorithm has to propagate the changes. This is done by pushing into $UPDATES(4)$ the triple that refers to the successor $C_1$ of the node $D_2$: the triple is $(3, 0, 0)$. When this triple is pulled from the vector $UPDATES(4)$, inequality (3.20) is evaluated and the cells $D_{(5,0,1)} = 7$, and $\Pi_{(5,0,1)} = (3, 0)$ are updated.

Moreover, the "loop find" procedure finds a new critical cycle that impacts on the throughput upper bound. The new cycle is $C_1 \rightarrow D_1 \rightarrow C_2 \rightarrow D_3 \rightarrow D_2 \rightarrow C_1$) that refers to the throughput of the graph that corresponds to throughput value $\frac{1}{7}$ ($\frac{tok(D_{5,0,1}) - tok(D_{0,0,0})}{D_{5,0,1} - D_{0,0,0}}$). The modified matrices $D$ and $\Pi$ are reported in Figures 3.16, 3.17, 3.18 and 3.19.

The theoretical worst-case complexity of the incremental algorithm is $O(n^4)$, the same as the non-incremental version. However, in practice the number of performed operations is much lower, as pointed out by the experimental results (see Section 3.6). In fact, the average complexity of the incremental algorithm depends on the number of graph modification, and this value is rarely high.

### 3.5.4 Further Optimizations

In this section we describe several improvements to the global throughput constraint described in Section 3.5.1 that speed up the throughput computation. Optimization concerns three aspects: first, the non-strictly connected components are removed as they do not contribute to the throughput computation, the cycles are partitioned into multi-resource and single resource

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| $D_{i,j,0}$ | 0 | 1 | 2 | 3 | 4 |
| 0 | 0 | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | 2 | 2 | -∞ |
| 2 | -∞ | 3 | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | 5 | **5** |
| 4 | -∞ | -∞ | **6** | -∞ | -∞ |
| 5 | -∞ | -∞ | -∞ | -∞ | -∞ |

Figure 3.16: Matrix $D_{i,j,0}$

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| Π | 0 | 1 | 2 | 3 | 4 |
| 0 | -1 | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | 0,0 | 0,0 | -∞ |
| 2 | -∞ | 2,0 | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | 1,0 | **1,0** |
| 4 | -∞ | -∞ | **4,0** | -∞ | -∞ |
| 5 | -∞ | -∞ | -∞ | -∞ | -∞ |

Figure 3.17: Matrix $\Pi_{i,j,0}$

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| $D_{i,j,1}$ | 0 | 1 | 2 | 3 | 4 |
| 0 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 2 | 3 | -∞ | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 4 | 6 | 6 | -∞ | -∞ | -∞ |
| 5 | **7** | -∞ | -∞ | -∞ | -∞ |

Figure 3.18: Matrix $D_{i,j,1}$

| | C1 | C2 | D1 | D2 | D3 |
|---|---|---|---|---|---|
| Π | 0 | 1 | 2 | 3 | 4 |
| 0 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 1 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 2 | 2,0 | -∞ | -∞ | -∞ | -∞ |
| 3 | -∞ | -∞ | -∞ | -∞ | -∞ |
| 4 | 3,0 | 4,0 | -∞ | -∞ | -∞ |
| 5 | **3,0** | -∞ | -∞ | -∞ | -∞ |

Figure 3.19: Matrix $\Pi_{i,j,1}$

cycles and considered separately. In the following we detail the optimizations performed.

#### 3.5.4.1 Removing the non-strictly connected components

Since the throughput value is cycle dependent, nodes not belonging to any cycle are useless. A filtering algorithm has been implemented to recursively remove (temporally) the non-strictly connected components from the graph. The result is a graph composed by a set of strictly connected sub-parts.

#### 3.5.4.2 Single-Resource Execution Time Bound

A first very trivial bound on the throughput value can be computed by considering cycles on each resource. During search, as activities are allocated, arcs and tokens are added to the graph to guarantee the non overlapping execution of the nodes over the resources (i.e. processors). This is done by setting a cyclic path that orders the activities execution over each resource.

Let us call $\text{Ex}_k$ the sum of the execution times of all activities allocated on resource $k$. $\text{Ex}_k$ is the inverse of the maximal throughput ($\text{Thp}_k$) that the resource $k$ can achieve:

$$\text{Ex}_k = \frac{1}{\text{Thp}_k} = \sum_{i \in \mathbb{V}_k} d_i \tag{3.21}$$

where $\mathbb{V}_k$ is the set of activities allocated on $k$ and $d_i$ is the execution time of the activities $i$.

$$\text{Thp}_{min} = \frac{1}{\max_{k \in \mathbb{R}} (\text{Ex}_k)} \tag{3.22}$$

is a throughput upper bound that must be higher than the current lower bound otherwise the search is stopped and the solver backtracks.

#### 3.5.4.3 Single-Resource Cycle Pruning

The key idea is that a resource $k$ can be part of a *multi-resource* cycle if and only if its activities have at least one input and one output arc that connect them to activities onto other resources. We can now remove every remaining *single-resource* cycle, since its impact over the throughput has been considered by computing $\text{Thp}_{min}$. The result is a reduced graph which consists of activities allocated on resources that communicate with each other.

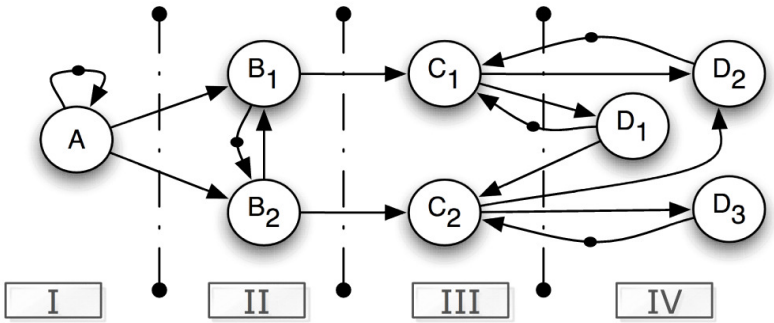


Figure 3.20: A graph allocation example

These optimizations filter the graph by removing nodes that do not contribute to the throughput computation.

Consider for example the graph reported in Figure3.20; it has eight activities allocated onto four different resources ($I...IV$). It is composed only by strictly connected components, so the first optimization (Section 3.5.4.1) is not employed. Then, assuming that the computation of $\mathrm{Ex}_k^{max}$ does not force a backtrack on the search process (see Section 3.5.4.2), the latter optimization is executed (see Section 3.5.4.3).

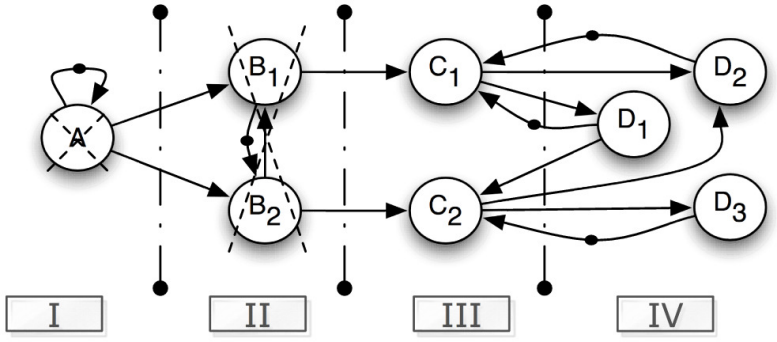


Figure 3.21: A graph allocated and optimized

Since the activity $A$, allocated on resource $I$, has only out-going arcs, it cannot be part of a multi-resource cycle. Thus it is removed from the graph. As a consequence, activities allocated on $II$ ($B_1$ and $B_2$) "loose" their in-going arcs. For this reason they are recursively removed.

The algorithm, in this example, computes the bound over the sub-graph composed by activities $C_1, C_2, D_1, D_2, D_3$ reducing the overall computation time of the throughput constraint (see Figure 3.21).

## 3.6 The Search

CP problems are generally solved via tree search. Constraint propagation is used to narrow the search space, but many branching choices still have to be explored during search. Hence, the efficiency of CP solvers heavily depends on good heuristics to prioritize branching decisions.

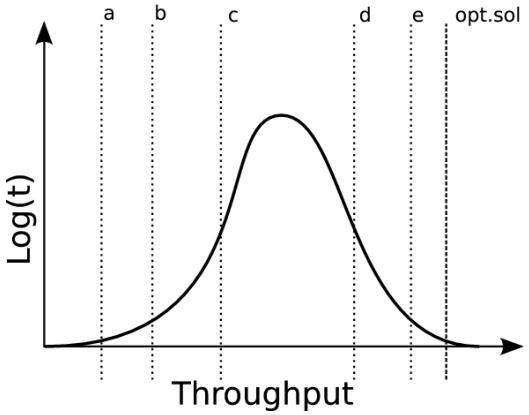


Figure 3.22: Transition Phase peak complexity during the search process

A pictorial intuition of this fact is given in Figure 3.22: the solid line represents the time spent by the search process (vertical axis with logarithmic scale) to find a feasible solution given a specific throughput requirement (horizontal axis).

When solving an optimization problem, whenever a feasible solution is found, the throughput value (higher than the current threshold) is set as new threshold and the solver keeps on searching until no other feasible solutions are present in the search space. The last solution found is the optimal one.

Note that the curve presents a sharp complexity peak in the transition phase between a loosely and a tightly constrained problem (between $c$ and $d$ dotted lines). This means that searching a feasible solution with threshold $c$ is harder w.r.t. a tighter one like $e$ or a lesser one like $b$. A loose threshold (dotted line $a$) leaves a lot of feasible solutions in the search space, hence finding a feasible solution is a relatively easy. If the threshold is tight (dotted line $e$) the propagation process becomes very effective, boosting the solution process. The main purpose of a search heuristic is to quickly find a solution that ensures a tight bound to drastically reduce the search space.

### 3.6.1 Variable Selection Heuristics

Experimental tests evidenced that branching over resource allocation variables have far-reaching implications over the throughput values, therefore

our heuristic function evaluates these variables first. Focusing on the more "decisive variables" first is more likely to lead to good solutions.

The heuristic function we propose is divided into two components:

- the variable selection heuristic intuitively gives priority to actors whose execution has more impact on the throughput value. This is achieved by giving higher rank (low value) to tasks with longer execution time and also giving priority to actors whose execution enables the execution of other nodes. The node $i$ chosen by the heuristic is the one with minimal value of the following expression:

$$\frac{\alpha \cdot d^{max}}{d_i} + \frac{\beta \cdot dep_i}{dep^{max}} \tag{3.23}$$

  where $d^{max}$ corresponds to the maximal node execution time, $d_i$ is the execution time of the node $i$. $dep_i$ corresponds to the number of nodes which precede actor $i$, and $dep^{max}$ the maximum over these values (A node with a low $\frac{dep_i}{dep^{max}}$ means that it's execution depends on few other nodes, therefore it could execute earlier than a node with an higher $\frac{dep_i}{dep^{max}}$, i.e. whose execution depends on more nodes).

  The heuristic function combines two distinct components, with relative weight set by two coefficients. The coefficients $\alpha$ and $\beta$ have been defined experimentally, and their values are respectively 0.68 and 0.32 ($\alpha = 1 - \beta$).

- The value (resource) selection heuristic beside balancing the load, tends to allocate on the same resource actors that are tightly linked by precedence constraints. This function tries to reduce the number of dependencies between tasks on different resources.
  This is achieved by selecting first the resource $k$ that minimizes the following expression:

$$\frac{\rho \cdot \mathrm{Ex}_k}{\mathrm{Ex}^{max}} + \frac{\sigma \cdot con_k}{con^{max}} \tag{3.24}$$

where $\mathrm{Ex}_k$ corresponds to the actual resource workload, i.e., the total execution time of the actors allocated on it. $\mathrm{Ex}^{max}$ is the highest workload over all resources. The value $con_k$ is the total execution times of the nodes that are *non-dependent* on $k$, and $con^{max}$ is the highest of these numbers. The coefficients $\rho$ and $\sigma$ are 0.79 and 0.21 respectively ($\rho = 1 - \varsigma$).

Note that the coefficients of the heuristic functions have been experimentally tuned: 1000 heterogeneous instances were solved with 20 different combination of coefficient values. The values of the best average solution quality were chosen.

The experimental results show (see Section 3.7) that the described heuristics obtain one order of magnitude speed up w.r.t. search using lexicographic ordering. Finally, symmetry due to homogeneous processors are broken at search time; namely, whenever an allocation decision has to be taken, if there is more than one free processor, the one with the lowest index is chosen.

## 3.7 Experimental Results

We have extensively evaluated the approach presented in this chapter for assessing three aspects:

1. the performance of the incremental throughput algorithm in comparison with the non-incremental version;

2. the scalability of the allocation and scheduling framework;

3. the quality of the solutions found.

The synthetic instances were built by means of the `sdf3` (see [98]) task-graph generator, designed to produce graphs with realistic structure and

parameters[13]. The instances were solved using a workstation with a 3.3GHz Core 2 Duo processor and 8GB of RAM. The system described so far was implemented on top of ILOG Solver 6.3.

### 3.7.1 Incremental Algorithm Evaluation

The following section proves the effectiveness of the proposed incremental algorithm (see Section 3.5.1) by comparing its computational time with respect to its non-incremental version on a set of 4500 instances. We have generated three sets of realistic task graph instances featuring 10, 12 and 15 nodes. Each set includes cyclic, acyclic and strictly connected graphs. For this experiments we assume that two homogeneous resources are available.

| Type | Node | SrcNInc | SrcInc | SrcSpUP | CstNInc | CstInc | CstSpUP |
|---|---|---|---|---|---|---|---|
| Cyclic | 10 | 2.022 | 0.64 | **2.174** | 1.443 | 0.11 | **11.88** |
| | 12 | 35.86 | 2.72 | **12.18** | 27.86 | 0.88 | **30.66** |
| | 15 | 3504.43 | 37.64 | **92.11** | 2980.28 | 9.43 | **315.04** |
| Strictly Connected | 10 | 3.063 | 0.89 | **2.421** | 2.264 | 0.15 | **14.12** |
| | 12 | 58.29 | 4.32 | **12.49** | 46.99 | 1.3 | **35.15** |
| | 15 | 4231.02 | 52.64 | **79.18** | 3546.98 | 10.92 | **323.81** |
| Acyclic | 10 | 3.88 | 1.213 | **2.19** | 2.77 | 0.16 | **16.31** |
| | 12 | 105.48 | 14.23 | **6.41** | 82.87 | 1.94 | **41.72** |
| | 15 | 5968.58 | 143.58 | **40.57** | 5106.05 | 18.89 | **269.31** |

Table 3.1: Search and Constraint execution times and speed-up

In Table 3.1 the first two columns (SrcNInc,SrcInc) refer to the Total Search Time for finding the solution with maximum throughput with the non-incremental and incremental algorithm version. We can see that the solver with the incremental algorithm runs up to 90 times faster (see the speed-up column, SrcSpUP).

The values in Table 3.1 represent the average over 500 instances. The remaining three columns (CstNInc, CstInc, CstSpUP) report respectively the computational times of the throughput filtering algorithm and the cor-

[13]The generator tends to produce tasks with high execution time variance (therefore representing the difference between the loading/storing task w.r.t. the faster executing ones) and with an average number of out-going arcs that ranges from 1.1 to 1.3. These coefficients produce SDF graphs that, transformed into HSDFG, will resemble to applications with high data-parallelism.

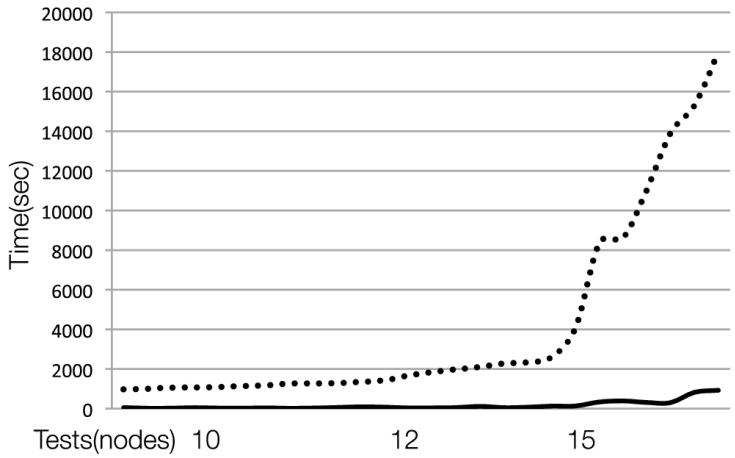


Figure 3.23: Graphical representation of the Speed-UP

responding speed-up. The speed-up column shows that the incremental filtering algorithm gains over one order of magnitude speed-up w.r.t. its non incremental version. Moreover the speed-up tends to increase with the dimension of the problem instance. The acyclic graphs are the most though to solve, as they feature relatively fewer arcs compared to the cyclic and to the strictly connected ones; this results in a higher number of possible scheduling choices and a larger search tree.

The problem faced is NP-hard and clearly the computational time grows up exponentially in the instance dimension. However, the reduced time for constraint computations in the incremental solver increases scalability and enables the solution of harder and larger problems.

| Node | % Non-Incr | % Incr |
|---|---|---|
| 10 | 72.24 | 15.35 |
| 12 | 79.01 | 15.99 |
| 15 | 82.91 | 19.32 |

Table 3.2: Relative algorithms computation time

This is clear in Figure 3.23, where the two reported lines represent the total time for the throughput constraint; the $x$ axis has an entry for each

instance. The dotted line refers to the non-incremental solver, the solid line to the incremental one. Instances are sorted according to the number of nodes. We can notice that the throughput computation time for the new incremental algorithm grows much more slowly.

The table in Figure 3.2 shows the relative amount of time that the algorithm computation absorbs during the search. It is evident that the new algorithm is definitely faster and lighter. Its impact on the search time is lower than 20% of the total time while the non-incremental version time exceeds 70%.

### 3.7.2 Overall Solver Experimental Evaluation

We have evaluated the scalability of our approach on various sets of synthetic instances, designed to match structure and features of realistic applications. We considered both cyclic and acyclic graphs. In particular the approach proposed tends to be more effective on cyclic graphs[14]; in fact, if a graph contains cycles, it has an implicit throughput upper bound defined by the longest loop in the graph. In contrast, acyclic graphs have no implicit bound and expose the highest parallelism: this makes them the most challenging instances.

For these experiments we assume that four resources are available. The generated graphs have been divided according to the number of nodes (from 10 to 18).

Table 3.3 presents the median and maximum computing time for cyclic (2nd and 3rd column) and acyclic (4th and 5th) instances. A time limit of 1200 seconds was set on all the experiments. As expected, the average running time grows exponentially with the size of the instances. However, the solution time is reasonable for graphs up to 20 nodes which is a realistic size for many real world applications.

This work was published in [21] and [18]. To the best of the authors knowledge, this was the first complete approach that handles cyclic/acyclic

[14]Working only with acyclic data-flow, the described approach looses in efficiency, and the *CROSS* approach, presented in Chapter 4 should be considered.

| Node | CMedian | CMax | AMedian | AMax |
|---|---|---|---|---|
| 10 | 0.12 | 11.84 | 0.53 | 13.44 |
| 12 | 0.96 | 36.84 | 1.98 | 47.77 |
| 14 | 12.67 | 146.75 | 27.15 | 275.56 |
| 16 | 63.33 | 446.24 | 96.08 | 659.32 |
| 18 | 187.64 | 837.32 | 269.43 | 1134.37 |

Table 3.3: Search execution times

synchronous data-flow graphs; this made harder any comparison with existing methods, as they were all incomplete. Incomplete approaches feature higher scalability, but provide sub-optimal solutions. We also performed a comparison with the simulation based procedure described in [97], on graphs with manageable size for our approach. Despite the incomplete approach is much faster then our tree search procedure, the solution provided by the incomplete method was found to be on average 20% worse than the optimal one.

Note that when the number of nodes becomes larger, the search could be stopped after a certain time limit (or a given number of feasible solutions), thus obtaining an incomplete approach. Differently from other incomplete approaches, our use of tree search and constraint propagation enables to find feasible solutions of tightly-constrained problems. We can compute a feasible solution of thousand-node graphs in terms of seconds.

Therefore we compare our method on five real benchmark with a state-of-the-art incomplete approach: the heuristic *Swing Modulo Scheduling* (SMS) approach, used by the GCC compiler [45]. The focus of these experiments is to assess the effectiveness on practically significant embedded multimedia benchmarks. Instances are derived from real application[15] such as: Sobel, JPEG2000, Motion JPEG, MPEG and MPEG-2.

Results are presented in Table 3.4. The first three columns report the name of the application, the number of tasks and the number of arcs, respec-

[15]These instances were developed as benchmarking work for the *Mapping Applications to MPSoCs 2009* workshop (`http://www.artist-embedded.org/artist/program,1755.html`). Source codes can be found at `http://www.artist-embedded.org/artist/benchmarks.html`.

tively. The following two columns (4 and 5) refer to the optimal solution computation time (solver runs until the optimality was proved) and optimality gap[16] of the solution found by the SMS approach. Note that the SMS method found all the solutions within 5 seconds. Each following column (from 4 to 8) refers to a different search time limit (5,10,30,60,300 seconds respectively) and presents the optimal gap. Note that all the instances were optimally solved within 106 seconds. The easiest instances, *Sobel* and *JPEG2000* were solved within a second. *Motion JPEG* solution computed within the first five seconds is the optimal solution (it is proved after 105.56 seconds) while SMS solution features a 8.12% gap. For both *MPEG* and *MPEG-2* applications, the SMS approach initially found a better solution; however note that our approach compute the optimal within 75.91 and 53.31 seconds respectively.

| Name | Nodes | Arcs | OPT | SMS | 5s | 10s | 30s | 60s | 300s |
|---|---|---|---|---|---|---|---|---|---|
| Sobel | 5 | 15 | **0.001** | 0% | **0%** | - | - | - | - |
| JPEG2000 | 8 | 10 | **0.07** | 0% | **0%** | - | - | - | - |
| Motion JPEG | 12 | 15 | **105.56** | 8.12% | 0% | 0% | 0% | 0% | **0%** |
| MPEG | 12 | 14 | **75.91** | 6.45% | 10.84% | 10.84% | 10.84% | 10.84% | **0%** |
| MPEG-2 | 12 | 14 | **53.31** | 9.46% | 10.34% | 10.34% | 0% | **0%** | - |

Table 3.4: SMS comparison on real benchmarks

This experimentation shows that in real contexts the solver can compute good quality solutions in terms of seconds (and the optimal solution within few minutes).

### 3.7.3 Solution quality evaluation

We finally designed a third set of experiments, to evaluate the search heuristics; tests were performed on a new large (1000 graphs) set of synthetic instances (see also [18]). Given an initial SDF graph, we perform mapping under two different assumptions: one allocates and schedules the derived HSDFG actors independently, while the second forces the allocation of all the

[16]The optimality gap represents the distance of the solution from the optimal one and it is computed in the following mode: $\text{Gap}(\%) = 100 * \frac{Opt-Sol}{Opt}$

HSDFG nodes corresponding to repetitions of the original SDFG nodes on the same processor. We refer to the first type of allocation as *unconstrained* and to the second type as *constrained*; the latter is typically obtained by approaches working directly on the SDFG, without a preliminary transformation into HSDFG [71, 97, 98]. In Table 3.5 we provide comparisons among

| Istance Size | *Const.* sol. | First sol. | First *Const.* sol. |
|---|---|---|---|
| 10 Nodes | 82,66% | 78,48% | 75,86% |
| 12-15 Nodes | 77,44% | 78,18% | 66,72% |

Table 3.5: Optimality gaps of incomplete searches.

the throughput achievable by complete search with the *unconstrained* and *constrained* approaches. This allows to assess the solution quality loss due to the use of a more restrictive assumption. We give the optimality gap of the the *constrained* solution (*Const.* sol.), the first feasible *unconstrained* solution (First sol.) and the first feasible *constrained* solution (First *Const.* sol.). The optimality gap widens as the number of nodes increases: on medium-size instances the optimal *unconstrained* throughput is about 20% higher than the *constrained* solution. This clearly demonstrates that the additional degrees of freedom enabled by mapping multiple actor iterations on different processors help in finding higher throughput solutions.

The third and fourth columns in the table refer to incomplete versions of the search procedure, which could be used to find fast, but sub-optimal solutions. In the third column we report the optimality gap obtained by stopping the search after the first solution found by tree search driven by our heuristic functions described in Section 3.6. The optimality gap is around 22%, which is significant but not enormous. This implies that the solver finds a reasonably good solution in a very short time, regardless of the exponential search effort required to reach the actual optimum. Thus, our strategy is quite effective even when used as a fast, incomplete search.

The first feasible *constrained* solution provides an estimate of the quality one could expect from the solution provided by an incomplete algorithm which map directly the SDFG. As expected, it has the largest optimality

gap, with a 30% loss in throughput. This result gives a clear indication that our algorithm provides a significant quality improvement with respect to previously presented incomplete algorithms.

# Chapter 4

# Solving the CRCS Problem

Cyclic scheduling problems[1] consist in ordering a set of activities executed indefinitely over time in a periodic fashion, subject to precedence and resource constraints. This class of problems has many applications in manufacturing, embedded systems and compiler design, production and chemical systems. In this chapter we present a Constraint Programming framework for cyclic resource constrained scheduling problems, based on modular arithmetic: in particular, we introduce a modular precedence constraint and a global cumulative constraint along with their filtering algorithms.

We discuss two possible formulations. The first one (referred to as *CROSS*) faithfully models a cyclic scheduling problem and makes use of both our novel constraints. The second formulation (referred to as *CROSS*$^*$) introduces a restrictive assumption to enable the use of classical resources constraints, but may incur a loss of solution quality. Many traditional approaches to cyclic scheduling operate by fixing the period value and then solving a linear problem in a generate-and-test fashion. Conversely, our technique is based on a non-linear model and tackles the problem as a whole: the period value is inferred from the scheduling decisions. The approaches have been tested on a number of non-trivial synthetic instances and on a set of realistic industrial instances. The methods proved to be effective in finding high quality

[1] As stated in Section 1 in this chapter we tackle the cyclic resource-constrained scheduling problems.

solutions in a very short amount of time.

The following section, Section 4.1, describes all related works highlighting the differences w.r.t. our approaches. Section 4.2 formally describes the modular algebra adopted in the approach. In section 4.3 and 4.4 we describe respectively the model with the constraints and their filtering algorithms. Section 4.5 proposes two search strategies, a specialized version of the solver and formally defines a dominance rule used to narrow the search space. Experimental results conclude the chapter.

## 4.1 Modulo Cyclic Scheduling in Research

The cyclic scheduling literature mainly arises in industrial and computing contexts. While there is a considerable body of work on cyclic scheduling in the OR literature [2], the problem has not received much attention from the AI community ([35] is one of the few related papers).

### 4.1.1 OR Approaches

An advanced ILP formulation for the modulo scheduling approach has been proposed in [36] by Dupont de Dinechin and is based on a time-indexed model. This kind of approach has some difficulties with periodic problems, since the schedule length (which determines the number of problem variables) may be fairly big compared to the period. In an attempt to circumvent this issue, a second formulation was proposed in [37] by Eichenberger and Davidson (by exploiting a decomposition of start times) at the cost of a reduction in the quality of the LP bound. An excellent overview of state-of-the-art formulations is given in [3], where the authors present also a new model based on the Danzig-Wolfe Decomposition. In [4] the authors propose

an hybrid approach using a retiming[2] techniques to build an ILP formulation of reduced size.

Good overviews of complete methods can be found in [48] and in [49]. All the mentioned approaches are based on iteratively solving resource constrained subproblems obtained by fixing the period value. To the best of our knowledge, this is a common trait of all the state-of-the-art approaches in the OR field. The main reason is that fixing $\lambda$ allows to solve the Resource Constrained Cyclic Scheduling Problem via an integer linear program, while modeling $\lambda$ as an explicit decision variable yields non-linear models.

### 4.1.2 A Constraint Programming Approach

In [35] the authors present a formulation for solving cyclic job shop scheduling as a Constraint Satisfaction Problem. They also describe how their formulation could be generalized to face problems with cumulative resources. The approach they propose is again based on fixing the period and solving the derived subproblems. The obvious drawback is that a resource constrained scheduling problem needs to be repeatedly solved for different $\lambda$ values to obtain the feasible/optimal solution.

Our method is also based on Constraint Programming, but it does not require to fix a $\lambda$ value, thanks to the use of a global constraint to model resource restrictions.

### 4.1.3 Incomplete Approaches

Several heuristic approaches have been proposed to find the smallest possible $\lambda$. These usually are instruction scheduling techniques that are used by many

[2]The retiming approach considered was the Decomposed Software Pipelining method presented by Darte and Huard in [30], based on the ideas of Gasperoni and Schwiegelshohn described in [41]. In this method, a cyclic scheduling problem ignoring resource constraints is first considered and a so-called *legal retiming* of the activities is issued. Second, a standard acyclic problem, taking this retiming as input, is solved through list scheduling techniques.

current compilers[3].

The *Iterative Modulo Scheduling* algorithm is presented in [86]: the main feature of the algorithm is its iterative nature in the sense that each activity can be scheduled and unscheduled (therefore considering backtrack operations) several times before a suitable slot is found.

Another interesting heuristic approach (called SCAN) is presented in [16]. SCAN is built on some of the main ideas behind Iterative Modulo Scheduling, but it is based on an ILP model.

In [52] Huff presented the slack modulo scheduling algorithm. Key of this method is the use of a bidirectional scheduling approach[4] (top-down and bottom-up) and the use of slacks. The slack of an unbound activity is a measure of the freedom that the activity would have if it was scheduled in the partial solution.

The state of the art for incomplete methods is probably *Swing Modulo Scheduling*, described in [66, 67]. The approach is implemented in the optimization chain of the gcc compiler [45]. The method produces effective schedules with a low computational cost (see [28] for a well structured comparison between most of these incomplete modulo scheduling approaches). Key for the efficiency is a valid variable selection function considering both the period bound imposed by the current iteration and the criticality of the path to which the activity to be scheduled belongs to.

Another state of the art incomplete method is presented by Benabid and Hanen in [11]. The approach is based on the ideas of retiming described in [23, 30, 41] extending them to propose a guaranteed heuristic for unitary resource-constrained modulo scheduling problems (i.e. instruction scheduling problems).

[3]Heuristic-based Modulo Scheduling is a family of Software Pipelining techniques that produce effective schedules with a relatively small compilation time. These Modulo Scheduling techniques take as input the application to be scheduled represented by its data dependence graph and a description of the architecture and produce a schedule for the application.

[4]The method can take advantage of the use of bidirectional scheduling method as the period $\lambda$ is fixed.

Those heuristic approaches compute a schedule for a single repetition of the application. The schedule is characterized by its length and by an *initiation interval*, which is the same as the iteration bound and determines the throughput. However, the schedule length can be extremely large, with implications on the size of the model in case of ILP based approaches (e.g. SCAN). Our model is considerably more compact, since we schedule a single iteration as opposed to a repetition, so that we restrict to a time window with length $\lambda$.

## 4.2 Modular Representation for Cyclic Schedules

In this section, we recall some modular arithmetic notions that are the foundations of the cyclic scheduling solver we propose.

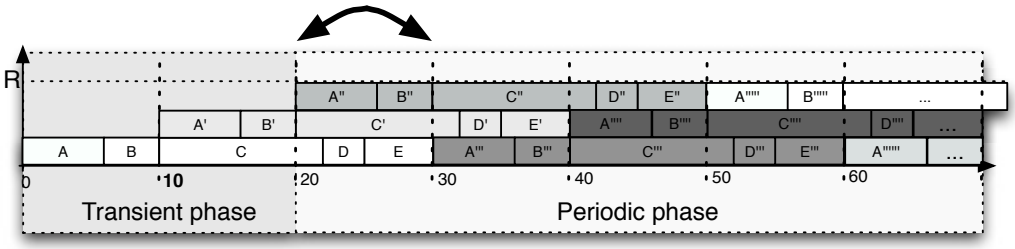


Figure 4.1: Modulo scheduling method optimal schedule

The main underlying idea is to focus on a $\lambda$-width time window in the periodic phase (see Figure 4.1). First, we present a start/end time decomposition similar to that in [37]. The start time of execution 0 of activity $i$ (i.e. $start(i, 0)$) can be expressed as:

$$start(i, 0) = s_i + \beta_i \cdot \lambda \tag{4.1}$$

where $s_i$ is a value in the half-open interval $[0, \lambda[$ and $\beta_i$ is an integer number. In practice, $\beta_i$ identifies the *iteration* when activity $i$ is first scheduled (see Section 2.2.1.3) and $s_i$ is its relative start time within the corresponding

$\lambda$-width time window, i.e. its *modular start time*. Analogously, the end time $end(i, 0)$ can be decomposed into a *modular end time* $e_i$ and an iteration number $\eta_i$.

$$end(i, 0) = e_i + \eta_i \cdot \lambda \tag{4.2}$$

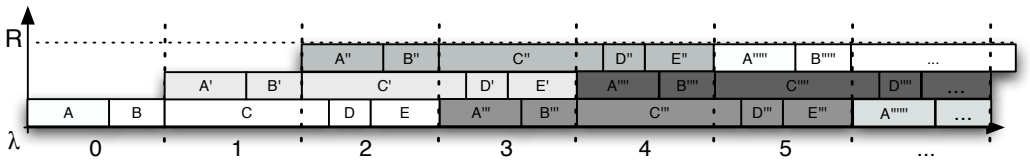


Figure 4.2: Modulo scheduling method optimal schedule

Figure 4.2 shows the optimal modulo scheduling related to the graph from Figure 2.3 described in Section 2.2.1. Note that activity $C$ is longer than the period $\lambda$, hence the activity begins in a period and ends in the following one. For instance, the third execution of $C$ (i.e. $C''$) starts at instant $start(i, 3) = 30$ and terminates at 42. Using the modular representation the activity has $\mathtt{s}_i = 0$, $\beta_i = 3$ and $\mathtt{e}_i = 2$, $\eta_i = 4$. Hence it starts at $start(i, 3) = 0 + 3 \cdot \lambda$ and ends at $2 + 4 \cdot \lambda$. Since $\lambda = 10$ we have $\mathtt{s}_i + \beta_i \cdot \lambda = 30$ and $\mathtt{e}_i + \eta_i \cdot \lambda = 42$.

Note that there is a strong correlation between iteration numbers and schedule length. In particular, a larger difference between the highest and smallest $\beta_i$ in an iteration corresponds to a larger schedule length. Moreover, start and end times are constrained by the relation $end(i, 0) = start(i, 0) + d_i$, hence we have $e_i + \eta_i \cdot \lambda = s_i + \beta_i \cdot \lambda + d_i$ and hence:

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

Moreover, since $e_i - s_i$ is stricly less than $\lambda$, we have $\eta_i - \beta_i = \left\lfloor \frac{d_i}{\lambda} \right\rfloor$, which means that $\eta_i$ is unambiguously determined once $\beta_i$ and $\lambda$ are known. We can also rewrite the temporal dependencies using the modular formulation. In particular, the relation:

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

is rewritten as:

$$start(j,0) + \omega \cdot \lambda \geq start(i,0) + (\omega - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)}$$
$$s_j + \beta_j \cdot \lambda + \omega \cdot \lambda \geq s_i + \beta_i \cdot \lambda + (\omega - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)}$$

performing the usual eliminations we have the following inequality that no longer depends on $\omega$:

$$s_j + \beta_j \cdot \lambda \geq s_i + (\beta_i - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)} \tag{4.3}$$

This is a very important result and it is equivalent to say that in a periodic schedule, if the (modular) precedence constraints are satisfied for iteration 0, then they are satisfied for every other iteration.

### 4.2.1 Resource Modeling

Before introducing our constraint model it is necessary to get a better insight into the resource usage profile of a periodic schedule. 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. The most unusual peculiarity of this case is that the value of $\lambda$ may modify the resource usage over time, as depicted in Figure 4.3. The picture presents the profiles corresponding to schedules with the same start times, but with 4 different period values:

- case a) represents a classic non-overlapped schedule.
- In b) the modular start time `s` is equal to the modular end time `e`, since the period is equal to the length of the activity.
- In case c) the period is shorter than the activity, hence the schedule is overlapped and the resource profile exhibits a "pulse" (called also resource peak) in the middle.
- In d) the period is slightly larger than the half of the activity duration.

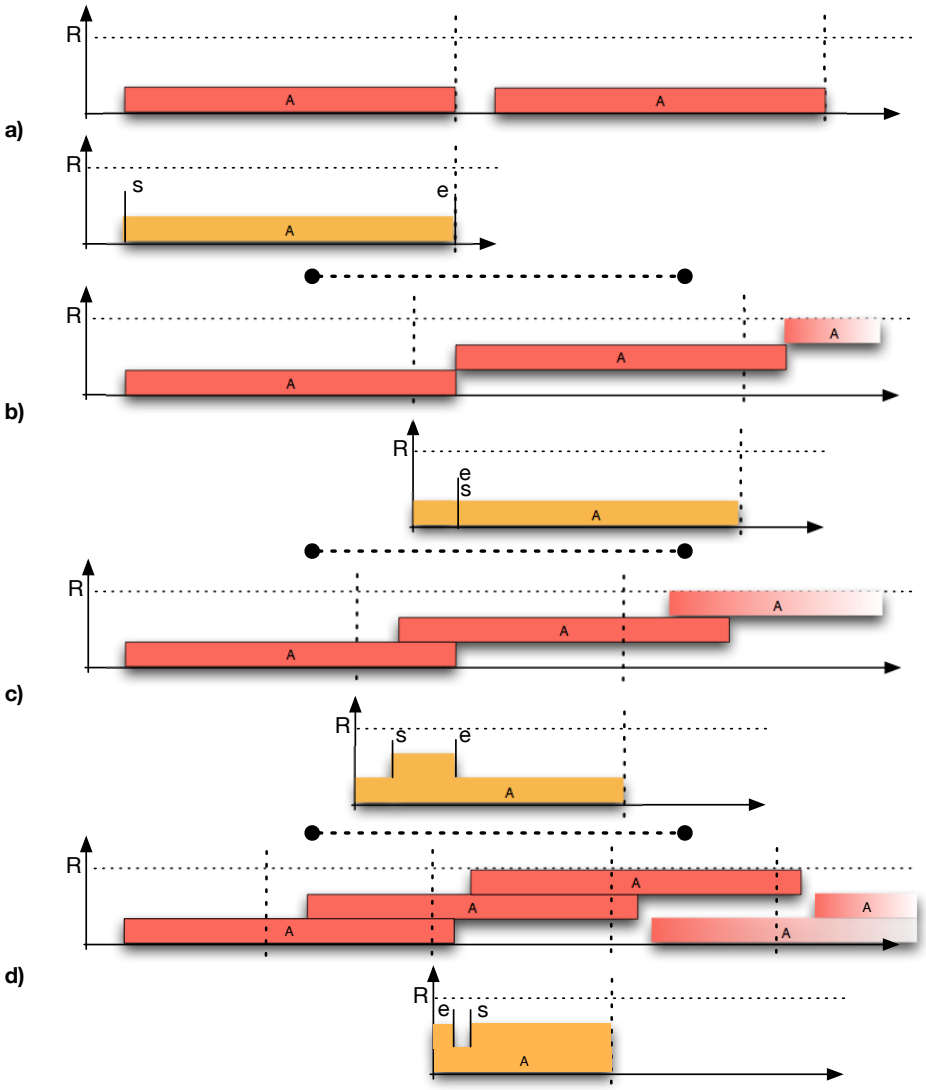


Figure 4.3: Resource profile modifications

In this profile the modular end time $\mathtt{e}$ precedes the modular start time $\mathtt{s}$, which is a bit counterintuitive.

Note that changes in the modulus affect not only the resource usage but also the the modular end time $\mathtt{e}_i$.

*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 $i$ are *always* executing. An extra iteration should be added if the time instant under analysis falls between the modular start and the modular end (i.e. the resource peak window).

In general, the amount of resource $k$ required by activity $i$ in a time window having width $\lambda$ is given by the following expression:

$$rq_{i,k}(s_i, d_i, t, \lambda) = \begin{cases} r_{i,k} \cdot \left( \left\lfloor \dfrac{d_i}{\lambda} \right\rfloor + 1 \right) & \text{if } \mathtt{s}_i \leq t < \mathtt{e}_i \text{ or } \mathtt{e}_i < \mathtt{s}_i \leq t \text{ or } t < \mathtt{e}_i < \mathtt{s}_i \\ r_{i,k} \cdot \left\lfloor \dfrac{d_i}{\lambda} \right\rfloor & \text{otherwise} \end{cases} \tag{4.4}$$

In other words, the resource usage is given by a constant factor $r_{i,k} \cdot \lfloor \frac{d_i}{\lambda} \rfloor$, plus an additional $r_{i,k}$ in case the considered time point lies "between" the modular start and the modular end (the quotes are used since we may have $\mathtt{e}_i < \mathtt{s}_i$).

In Figure 4.3, the constant usage factor is always 1 except for a), where it is 0. As a particular case, if $\beta_i = \eta_i$ the constant usage factor is zero and $rq_{i,k}(\mathtt{s}_i, d_i, t, \lambda)$ becomes a classical resource usage function. This suggests that forcing the end times to be within the modulus allows the use of classical resource constraints (see Section 4.3). A wider discussion on cyclic resource profile can be found in [68].

## 4.3 The Model

In this section we propose a complete constraint-based approach for the cyclic scheduling problem, based on modular arithmetic. Our CP model features three classes of variables, representing the (modular) start times, the corresponding iteration numbers and the modulus. The modular start time variables have domain $[0, \texttt{MAX\_TIME}[$, the iteration numbers are in $\{-\|\mathbb{V}\|..+\|\mathbb{V}\|\}$ and the modulus is in $]\texttt{IB}, \texttt{MAX\_TIME}]$[5]. We recall that $\|\mathbb{V}\|$ is the number of nodes in the graph. The value MAX_TIME instead is given by the sum of the durations of the activities and the sum of the time lags of the edges.

For sake of brevity, in the rest of the chapter we will use the notation $\mathtt{s}_i$, $\mathtt{e}_i$, $\beta_i$, $\eta_i$ and $\lambda$ to refer to variables rather than values. Moreover, we will speak of "start/end times", implicitly meaning their modular counterparts: non modular start/end times are never used in the model. In the rest of the chapter we also assume that:

- $EST_i$ is the earliest start time of $i$: the minimum in the domain of $\mathtt{s}_i$
- $LST_i$ is the latest start time of $i$: the maximum in the domain of $\mathtt{s}_i$
- $EET_i$ is the earliest end time of $i$: the minimum in the domain of $\mathtt{e}_i$
- $LET_i$ is the latest end time of $i$: the maximum in the domain of $\mathtt{e}_i$

Hence the domain of a start time variable $\mathtt{s}_i$ is actually $[EST_i..LST_i]$, with $LST_i < \lambda$. Similarly, the domain of $\mathtt{e}_i$ is $[EET_i..LET_i]$). Furthermore, we use the notation $\overline{x}$ and $\underline{x}$ to refer to the highest and the lowest values (i.e. the upper and the lower bounds) of the domain of a generic variable $x$. Obviously the lower/upper bound of a time variable is the same as its earliest/latest time, e.g. $\underline{\mathtt{s}}_i = EST_i$ and $\overline{\mathtt{s}}_i = LST_i$. Finally, as stated in Section 4.2, the $\mathtt{e}_i$ and $\eta_i$ values are equal to $\mathtt{s}_i + (d_i \mod \lambda)$ and $\beta_i + \lfloor \frac{d_i}{\lambda} \rfloor$ respectively. Therefore the model is actually based only on the $\mathtt{s}_i$ and $\beta_i$ variables.

[5]As stated in Section 2.1.3.1 the iteration bound IB is a valid lower bound for the modulus.

These variables are subject to temporal and resource constraints. In order to model a temporal dependency $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle$ we have implemented a modular precedence constraint (ModPC), with the following signature:

$$\texttt{ModPC}(\mathtt{e_i}, \mathtt{s_j}, \eta_\mathtt{i}, \beta_\mathtt{j}, \lambda, \theta_{\mathtt{i,j}}, \delta_{\mathtt{i,j}}) \tag{4.5}$$

where $\mathtt{e}_i$, $\mathtt{s}_j$, $\eta_i$, $\beta_j$, $\lambda$ are the variables representing the end time of activity $i$, the start time of activity $j$, their respective iteration numbers and the modulus. The parameters $\theta_{i,j}$ and $\delta_{i,j}$ are assumed to be fixed values. The filtering algorithm for the constraint is described in Section 4.4.1.

In the CRCSP all activities are subject to resource constraints. Unlike in traditional scheduling, the resource profile depends on the period value (see Section 4.2.1). For this reason we devised a global cumulative constraint based on a modular time table, that ensures a consistent resource usage (the GCCC, discussed in Section 4.4.3). The signature is as follows:

$$\texttt{GCCC}([\mathtt{s}_i], [d_i], [r_i], \textsc{cap}, \lambda) \tag{4.6}$$

where $[\mathtt{s}_i]$ is a vector of start time variables, $[d_i]$ is the vector of corresponding durations and $[r_i]$ are the requirements for the modeled resource. The CAP value is the resource capacity and $\lambda$ is the period variable.

As an alternative, it possible to use traditional resource constraints by making the assumption that $\beta_i = \eta_i$, i.e. that the end and the start time of an execution $(i, \omega)$ must be within to the same period. Formally:

$$\mathtt{s}_i \leq \mathtt{e}_i \leq \lambda \quad \forall i \in \mathbb{V}$$

Since no activity is scheduled across different periods, the classical cumulative constraint can be used. As a main drawback, we may incur a quality loss due to the presence of an unnecessary restriction, which can be substantial for large resource capacities and large activity durations. In such situations, if we make no special assumption and we allow $\eta_i \geq \beta_i$ and

$\mathtt{e}_i < \mathtt{s}_i$, we may obtain much better schedules. We refer as *CROSS* to the solution approach making use of both the modular precedence constraint and the GCCC, with no restricting assumption. Conversely, in the *CROSS*$^*$ approach we force the end times to be within the modulus and replace the GCCC with traditional cumulative constraints. The two approaches differ also for the adopted search strategy, discussed in Section 4.5.

### 4.3.1 Buffer Constraints

In a real world context, a precedence constraint often implies an exchange of intermediate products between activities that should be stored in buffers. For example, in the context of an embedded system (as described in Chapter 3) two activities may exchange data packets that should be stored in memory.

Every time the activity $i$ ends, its *product* is accumulated in a *buffer* and whenever the activity $j$ starts, a *product* in the buffer is consumed. It is common to have on each buffer a size limit, which can be modeled through the following constraint:

$$\beta_j - \beta_i + (\mathtt{e}_i \leq \mathtt{s}_j) \leq \mathtt{B}_{(i,j)} - \delta_{(i,j)} \tag{4.7}$$

where $\mathtt{B}_{(i,j)}$ is the size limit and the reified constraint $(\mathtt{e}_i \leq \mathtt{s}_j)$ evaluates to one if the condition is satisfied. Inequality (4.7) limits the number of executions of activity $i$ (*the producer*) before the first execution of $j$ (*the consumer*): this ensures that the buffer capacity is always respected. Obviously $\mathtt{B}_{(i,j)} \geq \delta_{(i,j)}$, otherwise the problem is infeasible. In fact, the value $\delta_{(i,j)}$ can be thought as the number of *products* already accumulated in the *buffer* $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle$ when the project starts for the first time (e.g. the initial tokens number in Synchronous Data-Flow models).

### 4.3.2 Constraint Model

The complete constraint model is formalized as follows:

$$z = \min(\lambda) \tag{4.8}$$

$$\texttt{ModPC}(\mathtt{s_i}, \mathtt{s_j}, \beta_\mathtt{i}, \beta_\mathtt{j}, \lambda, \theta_\mathtt{i,j}, \delta_\mathtt{i,j}) \quad \forall \text{ arc } \langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle \in \mathbb{A} \tag{4.9}$$

$$\texttt{GCCC}([\mathtt{s}_i], [d_i], [r_{i,k}], \text{CAP}_k, \lambda) \quad \forall \text{ resource } k \in \text{R} \tag{4.10}$$

$$\beta_j - \beta_i + (\mathtt{e}_i \leq \mathtt{s}_j) \leq \mathtt{B}_{(i,j)} - \delta_{(i,j)} \quad \forall \text{ arc } \langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle \in \mathbb{A} \tag{4.11}$$

$$\beta_j \leq \max_{i \in \text{PREC}_j} \left( \beta_i - \delta_{(i,j)} + \left\lceil \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda} \right\rceil \right) \quad \forall \text{ activity } j \in \mathbb{V} \tag{4.12}$$

$$\beta_i \geq \min_{j \in \text{NEXT}_i} \left( \beta_j + \delta_{(i,j)} - \left\lfloor \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda} \right\rfloor \right) \quad \forall \text{ activity } i \in \mathbb{V} \tag{4.13}$$

*Variables*

$\mathtt{s}_i \in [0, \texttt{MAX\_TIME}[ \quad \forall\, i \ \in \mathbb{V}$

$\beta_i \in \{-\|\mathbb{V}\|.. + \|\mathbb{V}\|\} \quad \forall\, i \ \in \mathbb{V}$

$\lambda \in ]\texttt{IB}, \texttt{MAX\_TIME}]$

Note that the constraints (4.12) and (4.13) are used as dominance rules, and are explained in Section 4.5.3.

## 4.4 The Propagation

This section presents the filtering algorithms of the Modular Precedence Constraint, the Buffer Constraint and the Global Cyclic Cumulative Constraint, used respectively to model temporal dependencies, buffer and resource constraints.

### 4.4.1 Modular Precedence Constraint ModPC

The Modular Precedence Constraint (ModPC) constraint has the following signature:

$$\texttt{ModPC}(\mathtt{e_i}, \mathtt{s_j}, \eta_\mathtt{i}, \beta_\mathtt{j}, \lambda, \theta_{(\mathtt{i,j})}, \delta_{(\mathtt{i,j})})$$

where $\mathtt{e}_i, \mathtt{s}_j, \eta_i, \beta_j, \lambda$ are variables representing respectively the end time of activity $i$, the start time of activity $j$, their respective iteration numbers and the modulus, and $\theta_{(i,j)}$, $\delta_{(i,j)}$ are constant values representing the minimum time lag and the iteration distance associated to the arc. For sake of simplicity, in the following we will omit the subscript when referring to $\theta$[6] and $\delta$.

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

- The filtering rules for the iteration variables, which updates the bounds of the $\eta_i$ and $\beta_j$ variables so that a proper distance (in terms of number of iterations) exists between activity $i$ and $j$.
- The filtering rules for the start time variables, which modify the start times of the involved activities to avoid infeasible overlaps.
- The filtering rules for the modulus variable, which compute a lower bound on the modulus.

The algorithm is executed whenever the domain bounds of any involve variable change. Filtering a single precedence relation achieves bound consistency and takes constant time.

[6] We recall that the value $\theta_{(i,j)}$ must be non-negative, i.e. $\theta \geq 0$.

#### 4.4.1.1 Filtering the Iteration Variables

With reference to the temporal model proposed in Section 4.3 we can rewrite the Inequality (4.3) as:

$$\mathtt{s_j} + \beta_j \cdot \lambda \geq \mathtt{e_i} + \theta + (\eta_i - \delta) \cdot \lambda \tag{4.14}$$

Starting from the equation above, we have

$$\eta_i - \beta_j - \delta \leq \frac{\mathtt{s_j} - \mathtt{e_i} - \theta}{\lambda} \tag{4.15}$$

with $-\lambda < \mathtt{s_j} - \mathtt{e_i} \leq \lambda$ and $\theta \geq 0$. Equation (4.15) can be used to obtain bounds over the $\beta_j$ (and $\eta_i$) variables, in particular:

$$\eta_i \leq \overline{\beta_j} + \delta + \left\lfloor \frac{\overline{\mathtt{s_j}} - \underline{\mathtt{e_i}} - \theta}{\overline{\lambda}} \right\rfloor \tag{4.16}$$

$$\beta_j \geq \underline{\eta_i} - \delta - \left\lfloor \frac{\overline{\mathtt{s_j}} - \underline{\mathtt{e_i}} - \theta}{\overline{\lambda}} \right\rfloor \tag{4.17}$$

As an example, suppose that during search two activities $i$ and $j$ connected by a temporal dependency $\langle i, j, 0, 0\rangle$ are overlapping and have $\mathtt{s_j} = 0, \mathtt{e_i} = 3$: then the Inequality (4.17) appears as follows:

$$\beta_j \geq \underline{\eta_i} - \left\lfloor \frac{-3}{\overline{\lambda}} \right\rfloor \tag{4.18}$$

which implies that $\beta_j > \eta_i$. In fact, two *connected* activities can overlap iff they have different iteration values: in particular $\beta_{sinkNode} > \eta_{sourceNode}$.

#### 4.4.1.2 Filtering the Start Time Variables

Let $\dot{\delta}_\omega$ be $\eta_i - \beta_j - \delta$ and hence $\underline{\dot{\delta}_\omega}$ be $\underline{\eta_i} - \overline{\beta_j} - \delta$. The Constraint (4.14) can now be written as:

$$\mathtt{s_j} - \mathtt{e_i} - \theta \geq \dot{\delta}_\omega \cdot \lambda \tag{4.19}$$

Note that we must have $\dot{\delta}_\omega \leq 0$: in fact $\dot{\delta}_\omega > 0$ would imply that $\beta_j < \eta_i - \delta$ which is, by definition, impossible. From $\dot{\delta}_\omega \leq 0$, we can deduce the two inequalities, that lead to bounds on the start/end variables:

$$\mathtt{s_j} \geq \mathtt{e_i} + \theta + \dot{\delta}_\omega \cdot \lambda \geq \underline{\mathtt{e_i}} + \theta + \underline{\dot{\delta}_\omega} \cdot \overline{\lambda} \tag{4.20}$$

$$\mathtt{e_i} \leq \mathtt{s_j} - \theta - \dot{\delta}_\omega \cdot \lambda \leq \overline{\mathtt{s_j}} - \theta - \underline{\dot{\delta}_\omega} \cdot \overline{\lambda} \tag{4.21}$$

Note that if $\dot{\delta}_\omega = 0$ the modular constraint boils down to a classical precedence constraint, i.e. $\mathtt{s_j} \geq \mathtt{e_i} + \theta$. In fact, two *connected* activities with the same iteration value cannot overlap and the Inequality (4.20) "pushes" the destination activity $j$ after the end time of $i$ plus the time lag $\theta$.

#### 4.4.1.3 Filtering the Modulus Variable

The domain of the modulus variable can be pruned only when $\dot{\delta}_\omega$ is strictly negative, i.e. $\dot{\delta}_\omega < 0$. In this situation, we can derive from Equation (4.19) the following inequality, resulting in a lower bound on the modulus variable:

$$\lambda \geq \frac{\mathtt{e_i} - \mathtt{s_j} + \theta}{-\dot{\delta}_\omega} = \left\lceil \frac{\underline{\mathtt{e_i}} - \overline{\mathtt{s_j}} + \theta}{\overline{\beta_j} - \underline{\eta_i} + \delta} \right\rceil \tag{4.22}$$

### 4.4.2 Filtering for the Buffer Constraints

In the following we report Inequality (4.7) from Section 4.3.1.

$$\beta_j - \beta_i + (\mathtt{e}_i \leq \mathtt{s}_j) \leq \mathtt{B}_{(i,j)} - \delta_{(i,j)} \tag{4.23}$$

From this formulation we can derive two expressions to compute bounds over the $\beta$ variables:

$$\beta_j \ \leq \overline{\beta}_i - (\underline{\mathtt{e}}_i \leq \overline{\mathtt{s}}_j) + (\mathtt{B}_{(i,j)} - \delta_{(i,j)}) \tag{4.24}$$

$$\beta_i \ \geq \ -\underline{\beta}_j + (\underline{\mathtt{e}}_i \leq \overline{\mathtt{s}}_j) - (\mathtt{B}_{(i,j)} - \delta_{(i,j)}) \tag{4.25}$$

where $\mathtt{B}_{(i,j)}$ is the size limit of the *buffer* and the reified constraint $(\underline{\mathtt{e}}_i \leq \overline{\mathtt{s}}_j)$ equals one if the condition is satisfied. In other words Inequality (4.7) limits the iteration distance between the activities involved.

### 4.4.3 The Global Cyclic Cumulative Constraint GCCC

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

$$\sum_{i \in \mathbb{V}} rq_{i,k}(\mathtt{s}_i, d_i, t, \lambda) \leq \mathrm{CAP}_k \quad \forall t \in [0, ..\lambda[$$

Since the GCCC refers to a single resource, for the sake of readability we remove the $k$ index from the requirement functions. Hence $r_{i,k}$ becomes $r_i$ and $\mathrm{CAP}_k$ becomes CAP. The constraint is inspired by the timetable filtering for the cumulative constraint [6]. The function $rq_i(\mathtt{s}_i, d_i, t, \lambda)$ (see Section 4.2.1) can be used to compute the resource consumption of the activity $i$ at time $t$. Similarly to timetable filtering, our algorithm works with the compulsory parts of activities.

Activity $i$ has a compulsory part if and only if there exist a time span where the activity is necessarily executing. This happens if $EST_i + d_i > LST_i$. To take into account the resource usage corresponding to compulsory parts, we introduce a generalized version of the $rq_i(\mathtt{s}_i, d_i, t, \lambda)$ function from Section 4.2.1:

$$\widehat{rq}_{i,k}(\mathtt{s}_i, d_i, t, \lambda) = \begin{cases} r_{i,k} \cdot \left( \left\lfloor \dfrac{d_i}{\lambda} \right\rfloor + 1 \right) & \text{if } \begin{cases} LST_i \leq t < EET_i \text{ or} \\ EET_i < LST_i \leq t \text{ or} \\ t < EET_i < LST_i \end{cases} \\ r_{i,k} \cdot \left\lfloor \dfrac{d_i}{\lambda} \right\rfloor & \text{otherwise} \end{cases} \tag{4.26}$$

Where $\mathtt{s}_i$ is a start time *variable*, rather than a value as in the non generalized for of the function. Basically, the generalized $\widehat{rq}_i(\mathtt{s}_i, d_i, t, \lambda)$ represents the resource usage of an unbound activity, corresponding in practice to that of

its compulsory part. If the $\mathtt{s_i}$ variable is bound, then $\widehat{rq}_{i,k}(\mathtt{s}_i, d_i, t, \lambda) = rq_{i,k}(s_i, d_i, t, \lambda)$ The shape of the function is the same as before, namely a constant factor plus a "pulse". Note that if the activity duration is longer than the period (i.e. $d_i \geq \lambda$), then there exists a compulsory part at least as wide as the period itself.

The GCCC constraint guarantees that:

1. the start time of each activity is not lower than the minimum instant where enough resources are available.

2. The modulus is not lower than the minimum value such that the cumulative usage due to the compulsory parts does not exceed the resource capacity.

The filtering algorithm exploits incremental computation and consists of three procedures:

- **Trigger**: this procedure is executed whenever any variable bound changes. The aim of this algorithm is to update the time tabling data structure.

- **Core**: this algorithm is executed at the end of all trigger procedures and it is structured in two independent phases:

    1. Start Time Propagation, that propagates the lower bound of the start time variables.

    2. Modulus Propagation, that computes the minimum $\lambda$ needed to guarantee feasibility.

- **Coherence**: the procedure is executed whenever the modulus upper bound changes. The procedure modifies the data structure to guarantee the coherence with the new $\lambda$ bound.

In the rest of this section we focus on the two phases of the **Core** procedure.

#### 4.4.3.1 Core Phase 1: Start Time Filtering Algorithm

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

$$\mathtt{s}_i \geq \min_{t \in [0,\lambda[} : \sum_{j \in V \setminus \{i\}} rq_j(s_j, d_j, t', \lambda) \ \leq \ \mathrm{CAP} - rq_i(s_i, d_i, t', \lambda) \ \ \forall t' \in [t, t + d_i^*]$$

where $d_i^* = d_i \bmod \lambda$ is the length of the pulse and is referred to as *modular duration*[7]. Similarly to the timetable approach, we adopt a data structure to store the minimum resource usage, given the current scheduling decisions. For every $t \in [0, \lambda)$, this is given by the following expression:

$$\sum_{i \in V} \widehat{rq}_i(\mathtt{s}_i, d_i, t, \lambda)$$

Note that changes in the expression value occur at the $LST_i$ and $EET_i$ of all the activities: this is a direct consequence of the $\widehat{rq}_i$ definition in Equation (4.26).

Intuitively the algorithm proceeds as follows: for each unbound activity $i$ the algorithm scans the resource profile, starting from $EST_i$, to search for a schedulability window. A schedulability window is a time slice large enough and with enough resources to allow the activity execution. The process stops when a window is found or the search goes beyond the Latest Start Time ($LST_i$). Since the solver is based on modular arithmetic, the procedure follows a modular time wheel and the $LST_i$ and $EET_i$ values (i.e. the time points when a profile change may occur) are stored in a circular queue. Recall that, since $LST_i$ and $EET_i$ are modular values, they are guaranteed to be lower than $\overline{\lambda}$. The filtering algorithm has an asymptotic complexity of $O(n^2)$.

**Data Structure** As stated in Section 4.3, each activity $i \in \mathbb{V}$ has four relevant time points: two of them are related to the start time, namely $EST_i$ (or $\underline{\mathtt{s}}_i$) and $LST_i$ (or $\overline{\mathtt{s}}_i$), and two related to the end time, i.e. $EET_i$, $LET_i$.

[7] In case $d_i$ is not integer, then $d_i^* = d_i - \lambda \cdot \lfloor \frac{d_i}{\lambda} \rfloor$

The constraint relies on an a circular queue $\Omega[0, (\|\mathbb{V}\| * 2)]$ where each activity $i \in \mathbb{V}$ is represented via two queue items, respectively corresponding to the $LST_i$ and its $EET_i$. Each item $\Omega[idx]$ stores three values:

- $\Omega[idx].activity$: the activity corresponding to item $\Omega[idx]$.
- $\Omega[idx].time$: the time value, either the $LST_i$ or the $EET_i$.
- $\Omega[idx].res$: the total resource usage at instant $\Omega[idx].time$; formally

$$\Omega[idx].res = \sum_{i \in \mathbb{V}} \widehat{rq}_i(\mathtt{s}_i, d_i, \Omega[idx].time, \lambda)$$

The items are sorted by increasing $\Omega[idx].time$.

**The Algorithm** The pseudo-code is reported in Algorithm 7 where $\mathbb{S}$ is the set of unscheduled activities. Lines 3-7 contain the algorithm initialization: the variable *canStart* represents the candidate start time of the schedulability window and initially it assumes the value of the Earliest Start Time of the selected activity $i$. The $feasible$ flag is used to remember if enough resource is available at the current *canStart* value. The $t_0$ and $idx_0$ values are respectively the time instant and the $\Omega$ index currently being processed. Initially, they are respectively equal to $EST_i$ and to the largest index in $\Omega$ corresponding to an item with $\Omega[idx].time \leq t_0$. Note that since profile changes always correspond to queue items, the resource usage at $t_0$ is the same as at $\Omega[idx_0].time$. The *startidx* variable stores the first index in $\Omega$ examined by the algorithm, i.e. the first $idx_0$. The *stop* flag is used for loop termination.

The algorithm searches for a schedulability window by scanning the queue $\Omega$, item by item. The window is assumed to start at the current *canStart* value and to end at $t_0$. Note that, with the exception of the first while-loop iteration (where $\Omega[idx_0].time$ may be strictly less than $t_0$), the current time point under examination (i.e. $t_0$) always matches the time value of the queue index under examination (i.e. $\Omega[idx_0].time$).

**Algorithm 7:** Core 1: Start Times Filtering Algorithm

**Data**: Let $\mathbb{S}$ be the set of activities not already scheduled

```
1  begin
2      forall the unbound activities i ∈ 𝕊 do
3          feasible = false
4          canStart = EST_i
5          t_0 = EST_i, idx_0 = largest idx such that Ω[idx].time ≤ EST_i
6          startidx = idx_0
7          stop = false
8          while ¬stop do
               // Resource availability check
9              avRes = Ω[idx_0].res − r̂q_i(s_i, d_i, t_0, λ̄)
10             if avRes + rq_i(canStart, d_i, t_0, λ̄) ≤ CAP then
11                 if ¬feasible then
12                     feasible = true
13                     canStart = t_0
14             else feasible = false
               // Offset determination
15             if idx_0 < startidx then offset = λ̄
16             else offset = 0
               // Pruning
17             if feasible ∧ canStart + d_i ≤ offset + t_0 then
18                 EST_i ← canStart
19                 stop = true
20             else if ¬feasible ∧ offset + t_0 > LST_i then
21                 fail()
               // Move forward
22             idx_0 = idx_0 + 1, t_0 = Ω[idx_0].time
```

From line 9 to 14 the algorithms checks the resource availability at $t_0$. The value $avRes$ computed at line 9 is the resource usage, adjusted by subtracting the contribution of the compulsory part of activity $i$. Line 10 checks if scheduling $i$ at time $canStart$ would cause an over-usage at time $t_0$. Note that $rq_i(canStart, d_i, t_0, \overline{\lambda})$ corresponds to the resource consumption of activity $i$ at time $t_0$, assuming it is scheduled at time $canStart$. Depending on

the result of the check, the $feasible$ flag is updated. A transition between $feasible = false$ to $feasible = true$ means that a new candidate start time for the schedulability window has been found and that the *canStart* value must be updated.

Since $\Omega$ is circular, at some point the process may reach the end of the queue and re-start from the first element: the filtering algorithm makes use of an offset value to distinguish between items encountered before and after crossing the end of the queue. In particular, for the latter ones it holds $idx_0 < startidx$ and the offset is set to $\overline{\lambda}$ (lines 10-11). We recall that $\overline{\lambda}$ (i.e. the maximum value in the domain of $\lambda$) is the modulus value corresponding to the minimum resource usage profile.

At lines 17-19 the algorithm checks if the current schedulability window, from *canStart* to $t_0$: 1) has enough available resource and 2) it is long enough to contain activity $i$. The time window length is adjusted by adding the value *offset*. If both the conditions hold, then a valid schedulability window has been found and the start time is filtered (line 18). If this is not the case, then the algorithm checks if the current $t_0$ (adjusted with the offset value) has become larger than $LST_i$ (line 20-21): in this case there is no way to schedule activity $i$ and the algorithm fails. If neither of the two situations occurs, at line 22 the process moves to the next item in $\Omega$ by updating $t_0$ and $idx_0$.

**Time Complexity** The filtering algorithm has two nested cycles, respectively over the set of non-scheduled activities $\mathbb{S}$ and over the items of the circular queue $\Omega$. The corresponding maximum numbers of iterations are $n$ and $2 \cdot n$. Hence the asymptotic complexity is $O(n^2)$.

#### 4.4.3.2 Core 2: Modulus Filtering Algorithm

In cyclic scheduling, it is possible to reduce the cumulative usage at time $t$ by increasing the modulus. As a consequence, unlike in classical scheduling, the compulsory parts in the current schedule may enforce a non-trivial lower bound on the feasible $\lambda$. The goal of lambda filtering is to find the mini-

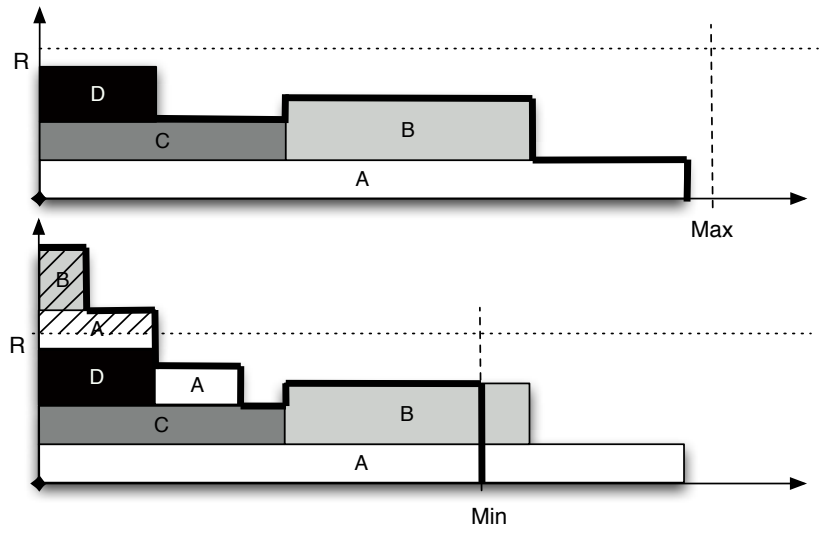


Figure 4.4: Resource Profile of a partial allocation with minimum and maximum modulus.

mum instant where sufficient resources are available for the current schedule. Formally:

$$\lambda \geq \min_{\lambda' \in [0,\overline{\lambda}[} : \sum_{i \in V} \widehat{rq}_i(\mathtt{s_i}, d_i, t, \lambda') \leq \text{CAP} \quad \forall t \in [0, \lambda'[$$

The algorithm makes again use of a circular queue $\Omega$. However, the value $\Omega[idx].time$ of the items corresponding to the Earliest End Time is computed assuming that $\lambda = \underline{\lambda}$. In other words, they are the Earliest End Time for the scenario where the period takes its minimum value, corresponding to the most constrained resource profile.

Figure 4.4 shows two different resource usage profiles for the same activity start times, but with different period value. The former corresponds to the maximum $\lambda$ value (i.e. $\overline{\lambda}$) and the latter corresponds to the minimum value (i.e. $\underline{\lambda}$). Note that with $\underline{\lambda}$ the activities $A$ and $B$ now cross the modulus, increasing the resource consumption at time 0. This causes a resource over-usage, represented by the shaded area. The effect of period $\lambda$ on the usage profile is a direct consequence of how the resource usage function

$rq(s_i, d_i, t, \lambda)$ is defined (see Section 4.2.1). The goal of the modulus filtering algorithm is to get rid of the over-usage by increasing the lower bound $\underline{\lambda}$. This is done in an iterative fashion, by repeatedly computing the over-usage integral and pushing the $\lambda$ lower bound.

**Algorithm 8:** Core 2: Modulus Filtering Algorithm

**Data**: *Let En be the cumulative over-usage amount*

```
1  begin
2      repeat
           // Initialize iteration
3          En = 0
4          Update and reorder the modular end times in Ω, given the new
           λ
5          lastres = 0, lasttime = 0
6          for idx = 0; idx < ‖Ω‖; idx = idx + 1 do
               // Update resource usage
7              let j be the activity corresponding to Ω[idx]
8              if Ω[idx] corresponds to LST_i then
9                  Ω[idx].res = lastres + r_j
10             else
11                 Ω[idx].res = lasttime − r_j
               // Update cumulative resource overusage
12             if lastres > CAP then
13                 En = En + (lastres − R) · (Ω[idx].time − lasttime)
               // Update last resource availability and time
                  point
14             lastres = Ω[idx].res, lasttime = Ω[idx].time
           // Prune the period lower bound
15         λ ← λ + En/CAP
16     until En = 0
```

**The Filtering Algorithm** The pseudo-code for the filtering procedure is reported in Algorithm 8. It is an iterative process, repeated until the resource over-usage becomes 0.

The cumulative resource overusage at each iteration is referred to as $En$. At the beginning of each iteration (lines 3-5), the algorithm updates and

reorders the data structure $\Omega$: this is necessary since $\underline{\lambda}$ is changed at each iteration, causing a modification of all the modular end times. Moreover, the procedure resets the values *lastres* and *lasttime*, respectively referring to the last processed resource consumption and to the corresponding time point.

Then the items in the queue are processed one by one. At lines 8-11 the resource consumption is increased or decreased depending on whether the item corresponds to a start or an end time. At line 12, the procedure checks if the resource consumption of the last processed time point (i.e. *lastres*) exceeds the resource. In this case, the cumulative resource over-usage on the time window $[lasttime, \Omega(idx).time[$ is summed to the current $En$ quantity. At line 14 the algorithm updates the *lastres* and *lasttime* values, before starting to process the next item. The period lower bound is updated at the end of each iteration (line 15), by dividing the cumulative overusage by the capacity of the resource and summing the quantity to the current $\underline{\lambda}$.

**Time Complexity** The algorithm inner loop has two main steps:

- Updating and sorting the modular end time for items in $\Omega$, with asymptotic complexity $O(n \cdot \log_n)$.
- Computing the cumulative overusage. This takes up to $2 \cdot n$ steps, and thus has asymptotic complexity $O(n)$

It is difficult to obtain a tight bound on the number of iterations of the main loop (line 2-16). If $\lambda$ is an integer, then $\overline{\lambda} - \underline{\lambda}$ provides a trivial bound and the overall asymptotic complexity is $O((\overline{\lambda} - \underline{\lambda}) \cdot n \cdot \log_n)$. In practice, however, the number of iterations of the main loop is very small.

## 4.5 The Search

In this section, we introduce two search strategies tailored for two models of the problem. The first (called *Path-based method*) works for the restricted case where the end time are assumed to be within the modulus, $\beta_i = \eta_i \quad \forall i \in$

$\mathbb{V}$. The second (called *random restart method*) is used in the generic version and it is based on restarts, although in this case they are employed to ensure correctness rather than to speed up search. As discussed in Section 4.2, by constraining the end times to be within the modulus, we can model the resource constraints with a traditional cumulative constraints.

In this section we present both the search strategies, together with their variable and value selection heuristics. Finally we also present a dominance rule on $\beta$ variables used to narrow the search space.

### 4.5.1 Path-based method

This is the search strategy employed in *CROSS*$^*$. Since our approach is designed to build periodic schedules, the start time value of each activity can be decided with respect to another, arbitrarily chosen, reference activity. The $\mathtt{s_i}$ and $\beta_i$ variables of this reference activity (called *The reference node*) can be fixed to zero. Such a restriction does not compromise the method completeness, as long as the $\beta_i$ variable domains are large enough. We always choose as a reference activity a node with no in-going arc having $\delta_{(i,j)} = 0$. Formally, for a node $src$ to be a candidate, it must hold $\nexists\ \langle i, src, \theta_{(i,src)}, \delta_{(i,src)}\rangle \in \mathbb{A}$ such that $\delta_{(i,src)} = 0$. Note that there always exists at least one node with this property, otherwise the graph is in deadlock and the problem is infeasible. A formal proof can be found in [14].

The reference activity (let its index be $src$) is always immediately scheduled, by posting $\mathtt{s}_{src} = 0, \beta_{src} = 0$, with no backtrack possible. Then, at any search node the next activity $j$ to be used for branching is chosen among those connected by an arc $(i, j)$ to one of the activities $i$ already scheduled. Those candidates are ranked according to the number of outgoing arc (the more, the better). This selection criterion seems to lead to better filtering of the Modular Precedence Constraint and for the dominance rule described in Section 4.5.3. In case of ties, the algorithm prefers activities with longer duration, and finally with smaller index.

Then we assign a value to the start time and to the iteration number of the selected activity, i.e. to variables $\mathtt{s}_j$ and $\beta_j$. This is done in two successive

search steps, rather than by posting the conjunction of the two assignments as a single constraint.

In particular, the algorithm attempts to schedule the activity at its earliest start time, i.e. it assigns $\mathtt{s}_j$ to its minimum value. On backtrack, the activity is postponed, i.e. marked as non selectable until its earliest start time is modified by propagation. This is analogous to what it done in the classical schedule-or-postpone strategy for the RCPSP [62] and can be done since in *CROSS*$^*$ we enforce resource restrictions by means of traditional cumulative constraints.

The $\beta_j$ variables are assigned via labeling, with value order given by increasing $|v|$ (where $v$ is the value to be assigned). In case of ties, the positive $v$ is chosen. This criterion is based on the existing correlation between iteration numbers and schedule length (see Section 4.2) and is designed to produce schedules as short as possible. A deeper discussion on the values that the iteration variables could assume can be found in Section 4.5.3). On backtrack, the previously assigned value is simply removed from the domain of $\beta_j$.

Unlike other modulo scheduling approaches, our solution method does not require to a-priori fix the period. When all the activities have been scheduled, we simply assign to the $\lambda$ variable the minimum in its domain. This value is guaranteed to be feasible by the propagation of precedence and resource constraints. Optimization is performed in a traditional CP fashion by posting a permanent upper bound on $\lambda$ whenever a feasible solution is found.

### 4.5.2 Random Restart method

If the activities are allowed to cross different iterations, like in our generic *CROSS* approach, the resource usage becomes dependent on $\lambda$. As a major consequence, by scheduling activities at their earliest (modular) start time, updated by resource constraint propagation, we may miss an optimal solution. Therefore, we had to devise a different search strategy for the non restricted *CROSS* approach.

The structure of the search strategy is the same as before: an activity $j$ is selected, then a value is assigned to its start time and iteration variables (i.e. $\mathtt{s_j}$ and $\beta_j$) in two successive search nodes. Once all the activities have been scheduled, the $\lambda$ variable is fixed to $\underline{u}$. The iteration numbers are assigned as in *CROSS*$^*$, giving priority to values $v$ in the domain having the minimum absolute value $|v|$ and posting $\beta_j \neq v$ on backtrack.

As a first difference, we do not fix a reference activity. At every search node, the activity $j$ to be scheduled is chosen at random among those connected to at least one already scheduled activity $i$. At the root node (where no scheduling decision has been taken) the choice is simply made at random.

Second, the selected activity is not scheduled at $EST_i$. Instead, the algorithm keeps a list of candidate start times, corresponding to the values $\mathtt{s_i} + (\mathtt{d_i} \mod \overline{\lambda})$ for all the already scheduled activities. The value 0 is always added to the list in case it is not present. Then the selected activity $j$ is scheduled at the smallest candidate start time $t$ falling in the interval $[EST_i, LST_i]$. If there is no such $t$, the algorithm backtracks without even opening a choice point. Otherwise, the activity $j$ is scheduled at time $t$ and on backtrack the constraint $\mathtt{s_i} > t$ is posted.

The modifications described so far are not yet sufficient to guarantee the method completeness, because the assigned start times (equal to $\mathtt{s_i} + (\mathtt{d_i} \mod \overline{\lambda})$ for some activity $i$) depend on the $\overline{\lambda}$ value, which is updated by the bounding constraint whenever an improving solution is found. To address this issue, we simply restart[8] the search process every time we find an new solution: this ensures that consistent start times are computed and the optimal solution is not missed.

### 4.5.3 Dominance Rules

One important observation is that assigning different iteration values to activities connected by an arc $(i, j)$ allows to place activity $j$ before activity $i$ in the $\lambda$-long window, apparently violating the precedence constraint. In

[8]Note that in this case restarts are needed to ensure the method completeness, rather than to speed up the search as it is usually the case in CP.

particular, this requires to $\beta_j$ to be greater than $\beta_i$ by a minimum amount: in further increase obtains the same effect. This observation serves as the basis to devise a dominance rule that filters redundant values for the $\beta$ domains.

Suppose we want to schedule two activities $i$ and $j$, connected by a temporal dependency $\langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle$. From (4.3)[9] we derive:

$$\beta_j \geq \beta_i - \delta_{(i,j)} + \left\lceil \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda} \right\rceil \tag{4.27}$$

and let $\hat{\delta}_\beta = \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda}$. Let's focus on the successor $j$. We analyze the dependency in Equation (4.27) in two different cases:

1. activity $j$ follows activity $i$: in this case $\mathtt{s}_j \geq \mathtt{s}_i + d_i + \theta_{(i,j)}$. Since $\mathtt{s}_j < \lambda$, $\hat{\delta}_\beta$ is in the interval $]-1, 0]$. Hence, $\lceil \hat{\delta}_\beta \rceil = 0$ and then

$$\beta_j \geq \beta_i - \delta_{(i,j)} \tag{4.28}$$

2. the two activities overlap in the $\lambda$-long window, or activity $j$ precedes activity $i$: in this case $\mathtt{s}_j < \mathtt{s}_i + d_i + \theta_{(i,j)}$ and $\hat{\delta}_\beta$ is strictly positive. Therefore, the lowest possible value that $\lceil \hat{\delta}_\beta \rceil$ can assume is 1, which implies

$$\beta_j > \beta_i - \delta_{(i,j)} \tag{4.29}$$

Note that $\beta_j$ must be at least equal to $\beta_i - \delta_{(i,j)}$, and must be strictly greater if we want activity $j$ to overlap with or precede activity $i$ in the time window. In particular, the smallest value that allows the apparent violation of the precedence constraint is $\beta_i - \delta_{(i,j)} + \lceil \Delta_\beta \rceil$: any higher value provides no benefit from a schedule building perspective. This information can be used to devise a dominance constraint that removes dominated (but feasible) values from the domain of $\beta$ variables. In detail, we can post:

$$\beta_j \leq \max_{i \in \text{PREC}_j} \left( \beta_i - \delta_{(i,j)} + \left\lceil \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda} \right\rceil \right) \tag{4.30}$$

[9] $\mathtt{s}_j + \beta_j \cdot \lambda \geq \mathtt{s}_i + (\beta_i - \delta_{(i,j)}) \cdot \lambda + d_i + \theta_{(i,j)}$

where $\textsc{Prec}_j = \{i \in \mathbb{V} \mid \langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle \in \mathbb{A}\}$ is the set of the predecessors of $j$. Note that, as a combined consequence of the modular precedence constraints (i.e. Equation (4.27)) and of the dominance rule, $\beta_j$ is strongly constrained by the $\beta_i$ of the predecessor activities: this is the reason why scheduling activities according to their topological order may result in a much better propagation (see Section 4.5.1).
We can obtain a second dominance rule by reasoning in a similar fashion for the successors of an activity $i$:

$$\beta_i \geq \min_{j \in \textsc{Next}_i} \left( \beta_j + \delta_{(i,j)} - \left\lfloor \frac{\mathtt{s}_i + d_i - \mathtt{s}_j + \theta_{(i,j)}}{\lambda} \right\rfloor \right) \tag{4.31}$$

where $\textsc{Next}_i = \{j \in \mathbb{V} \mid \langle i, j, \theta_{(i,j)}, \delta_{(i,j)} \rangle \in \mathbb{A}\}$ is the set of the successors of $i$.

> To make the rules clearer, we make a simple example. Assume we have a graph with three activities $A, B, C$ connected with two arcs: $\langle A, B, 0, 0 \rangle$ and $\langle B, C, 0, 0 \rangle$. Assume that each variable is initially unbound, with $s_i \in [0, \lambda[$ and $\beta_i \in \{-3.. + 3\}$ for every activity. If we schedule activity $A$ by posting $s_A = 0$ and $\beta_A = 0$, then the $\beta_i$ domain for all other activities becomes $\{0.. + 3\}$. The application of the dominance rule (4.30) results in a further reduction of the domains, in particular $\beta_B \in \{0, +1\}$ and $\beta_C \in \{0.. + 2\}$. Note that the values $\beta_B = 2$ and $\beta_B = 3$ are feasible but redundant. At this point, if the activity $B$ is scheduled so that it overlaps with $A$ (i.e. $\mathtt{s}_B < \mathtt{s}_A + d_A$), its iteration variable is forced to be $\beta_B = 1$.

From an implementation perspective, the direct formulation of the rules may be difficult to model and heavy to propagate. It is therefore convenient to modify the right hand expression in Inequality 4.30 with an upper bound,

obtaining the following, updated rule:

$$\beta_j \leq \max_{i \in \text{PREC}_j} \left( \beta_i - \delta_{(i,j)} + \left\lceil \frac{\lambda + d_i + \theta_{(i,j)}}{\lambda} \right\rceil \right) \tag{4.32}$$

$$\text{i.e.} \quad \beta_j \leq \max_{i \in \text{PREC}_j} \left( \beta_i - \delta_{(i,j)} + 1 + \left\lceil \frac{d_i + \theta_{(i,j)}}{\lambda} \right\rceil \right) \tag{4.33}$$

For a fixed $\overline{\lambda}$ value, the expression $\left\lceil \frac{d_i+\theta_{(i,j)}}{\lambda} \right\rceil$ can be replaced with a fixed value to obtain a (much simpler to compute) upper bound on $\beta_j$. This technique is employed in the context of the *CROSS* search strategy, which works by restarting the search whenever the $\lambda$ upper bound is modified.

## 4.6 Experimental Results

We have evaluated the effectiveness and scalability of both the approaches discussed in Section 4.3 (*CROSS* and *CROSS*$^*$) on various sets of instances. Moreover, we have compared them against a state-of-the-art ILP method and a state-of-the-art heuristic.

In the first part of this section we will focus on the *CROSS*$^*$ approach: we show that the solver is able to compute good quality solutions in terms of seconds in the context of a real world problem. In the second part of the section we present a comparison between *CROSS* and *CROSS*$^*$, plus a blocked and an unfolded approach. More in detail, this section contains:

1. An experimentation on industrial instances: the main purpose of this experimental evaluation is to show the our *CROSS*$^*$ approach to cyclic scheduling is viable in a practical setting. In particular, the experimentation will focus on instruction scheduling benchmarks, namely an industrial set of 36 instances for the ST200 processor by STmicroelectronics (also employed in [3]). We compare *CROSS*$^*$ with two state-of-the-art approaches: the complete ILP method presented in [3] and *Swing Modulo Scheduling* (SMS), a heuristic used by the GCC compiler [45].

2. An evaluation of the solution quality obtained by the *CROSS*$^*$ approach: the second group of tests is performed on a set of synthetic instances and compares the best solution obtained by *CROSS*$^*$ (within 300 seconds) with a lower bound on the instance period (namely, the *iteration bound* from Section 2.1.3.1).

3. A comparison between *CROSS*/*CROSS*$^*$ and blocked/unfolded scheduling: this experimentation is performed on a set of synthetic instances, for which we compare the solution quality of both the approaches with respect to classic blocked and unfolded scheduling.

4. A comparison between *CROSS* and *CROSS*$^*$, w.r.t. the Throughput / Resource trade-off: the fourth group of results compares our two approaches in terms of Throughput / Resource-usage trade-off.

All our solvers have been implemented in IBM ILOG Solver and Scheduler 6.7. All the experiments are performed on an Intel Core 2 Duo 3.3GHz with 8GB of RAM. A 300 seconds time limit was set on each run. The synthetic instances were built by means of an internally developed task-graph generator, designed to produce graphs with realistic structure and parameters.

### 4.6.1 Evaluation of *CROSS*$^*$ on Industrial Instances

The first set of 36 instances is obtained from a compiler for a VLIW (Very-Long Instruction Word) architecture. In this setting, cyclic scheduling problems arise when optimizing inner loops at instruction level: in this context the activities represent instructions, the precedence constraints model data and control dependencies and the resources are the hardware units required to execute the instructions. In the considered instances, all the resources have unary capacity and all the instructions have unary duration. With the objective to evaluate the approaches on a more diversified setting, in [3] the authors have obtained an additional set of modified instances, by replacing the original resource capacities and consumptions with randomly generated integer values.

The smallest instance features 10 nodes and 42 arcs, while the largest one has 214 nodes and 1063 arcs. In [3] the authors present two ILP formulations for the resource-constrained modulo scheduling problem (RCMSP), which is equivalent to the CRCSP discussed in this chapter, with the (easy to enforce) exception that the period must be integer. As described in [3], both the 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 with the solution found by our method within a 300 sec time limit[10]. We also compare our approach with an incomplete method, namely a state of the art heuristic approach called *Swing Modulo Scheduling* (SMS), presented in [66] and used by the gcc compiler [45].

As we mentioned, our benchmarks consists of two subsets of instances: the original problems extracted by the compiler and the modified ones (which tend to be more challenging). The results of the experiments performed with both the sets are reported respectively in Table 4.1 and Table 4.2. In the tables the first three columns describe the instance (name, number of nodes and arcs), the third shows the run-time of the ILP approach in [3], the fourth and the fifth respectively report the solution time and the quality of our solutions, evaluated by means of the gap w.r.t. the optimal solution[11]. The last two columns present the solution time and the quality gap for the *SMS* heuristic approach. For some instances (see the missing time information) the ILP approach was not able to find a solution: in such cases we report the best solution found by $CROSS^*$, which is used as a reference for computing the *SMS* gap.

On the *industrial* set, $CROSS^*$ is able to find the optimal solution within one second for all but one instance (`adpcm-st231.2`, whose optimality gap is 2.44%). We also found a solution for the `gsm-st231.18` instance that

[10]In a preliminary experimentation the solver was found to typically provide the best solution in less than a second. Hence a 300 seconds time limit should be widely sufficient for the method to converge to best solution it can find.

[11]For both $CROSS^*$ and SMS, the quality gap is computed with the following formulation: $100 * (Sol - ILP\_Opt)/ILP\_Opt$, where $ILP\_Opt \leq Sol$.

| | | | | *CROSS* | | SMS | |
|---|---|---|---|---|---|---|---|
| Instances | nodes | arcs | ILP time(sec) | time(sec) | **Gap(%)** | time(sec) | **Gap(%)** |
| adpcm-st231.1 | 86 | 405 | 14400.00 | 0.03 | 0% | 0.41 | 19.23% |
| adpcm-st231.2 | 142 | 722 | 582362.00 | 0.18 | 2.44% | 1.74 | 0% |
| gsm-st231.1 | 30 | 190 | 0.05 | 0.02 | 0% | 0.02 | 0% |
| gsm-st231.2 | 101 | 462 | 79362.00 | 0.03 | 0% | 0.61 | 0% |
| gsm-st231.5 | 44 | 192 | 0.05 | 0.01 | 0% | 0.06 | 13.33% |
| gsm-st231.6 | 30 | 130 | 17.00 | 0.01 | 0% | 0.02 | 31.25% |
| gsm-st231.7 | 44 | 192 | 0.05 | 0.01 | 0% | 0.07 | 41.66% |
| gsm-st231.8 | 14 | 66 | 0.05 | 0.01 | 0% | 0.01 | 31.25% |
| gsm-st231.9 | 34 | 154 | 0.05 | 0.01 | 0% | 0.02 | 0% |
| gsm-st231.10 | 10 | 42 | 0.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.11 | 26 | 137 | 0.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.12 | 15 | 70 | 0.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.13 | 46 | 210 | 1856.00 | 0.01 | 0% | 0.09 | 0% |
| gsm-st231.14 | 39 | 176 | 301.25 | 0.98 | 0% | 0.05 | 17.39% |
| gsm-st231.15 | 15 | 70 | 0.05 | 0.01 | 0% | 0.01 | 28.57% |
| gsm-st231.16 | 65 | 323 | 7520.00 | 0.01 | 0% | 0.2 | 0% |
| gsm-st231.17 | 38 | 173 | 0.05 | 0.01 | 0% | 0.17 | 23.81% |
| gsm-st231.18 | 214 | 1063 | - | 0.05 | - | 58.36 | 3.80%* |
| gsm-st231.19 | 19 | 86 | 0.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.20 | 23 | 102 | 0.05 | 0.01 | 0% | 0.02 | 0% |
| gsm-st231.21 | 33 | 154 | 0.05 | 0.01 | 0% | 0.02 | 45.45% |
| gsm-st231.22 | 31 | 146 | 0.05 | 0.01 | 0% | 0.03 | 0% |
| gsm-st231.25 | 60 | 273 | 3652.00 | 0.01 | 0% | 0.15 | 0% |
| gsm-st231.29 | 44 | 192 | 12.60 | 0.01 | 0% | 0.06 | 23.81% |
| gsm-st231.30 | 30 | 130 | 12.00 | 0.01 | 0% | 0.02 | 0% |
| gsm-st231.31 | 44 | 192 | 47.00 | 0.01 | 0% | 0.05 | 41.67% |
| gsm-st231.32 | 32 | 138 | 0.05 | 0.01 | 0% | 0.02 | 31.25% |
| gsm-st231.33 | 59 | 266 | 2365 | 0.06 | 0% | 0.13 | 11.76% |
| gsm-st231.34 | 10 | 42 | 0.05 | 0.01 | 0% | 0.01 | 6.25% |
| gsm-st231.35 | 18 | 80 | 0.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.36 | 31 | 143 | 27.00 | 0.01 | 0% | 0.03 | 14.29% |
| gsm-st231.39 | 26 | 118 | 0.05 | 0.01 | 0% | 0.02 | 0% |
| gsm-st231.40 | 21 | 103 | 0.05 | 0.01 | 0% | 0.02 | 0% |
| gsm-st231.41 | 60 | 315 | 2356.00 | 0.01 | 0% | 1.37 | 0% |
| gsm-st231.42 | 23 | 102 | 0.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.43 | 26 | 115 | 0.05 | 0.76 | 0% | 0.01 | 21.73% |

Table 4.1: Run-Times/Gaps of Industrial instances

was previously unsolved. The *SMS* heuristic method obtains an average optimality gap of 14.58%. Its average run time is less than few seconds, with a maximum of 58.36 seconds, corresponding to instance `gsm-st231.18` . For the previously unsolved instance (i.e. `gsm-st231.18`), the gap between *SMS* and *CROSS*$^*$ is 3.8%.

| | | | | *CROSS* | | SMS | |
|---|---|---|---|---|---|---|---|
| Instances | nodes | arcs | ILP time(sec) | time(sec) | **Gap(%)** | time(sec) | **Gap(%)** |
| adpcm-st231.1M | 86 | 405 | - | 0.04 | - | 6.61 | 40.8%* |
| adpcm-st231.2M | 142 | 722 | - | 0.09 | - | 68.55 | 55%* |
| gsm-st231.1M | 30 | 190 | 250.00 | 0.02 | 10.70% | 0.04 | 10.7% |
| gsm-st231.2M | 101 | 462 | - | 0.04 | 1.50%* | 8.36 | - |
| gsm-st231.5M | 44 | 192 | 280 .00 | 0.01 | 0% | 0.06 | 5.26% |
| gsm-st231.6M | 30 | 130 | 152.00 | 0.01 | 0% | 0.08 | 0% |
| gsm-st231.7M | 44 | 192 | 92.00 | 0.01 | 0% | 0.24 | 2.38% |
| gsm-st231.8M | 14 | 66 | 0.27 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.9M | 34 | 154 | 0.56 | 201.56 | 0% | 0.03 | 8.57% |
| gsm-st231.10M | 10 | 42 | 0.1 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.11M | 26 | 137 | 0.37 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.12M | 15 | 70 | 12.65 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.13M | 46 | 210 | 985.03 | 0.02 | 0% | 0.22 | 0% |
| gsm-st231.14M | 39 | 176 | 220.00 | 0.01 | 2.94% | 0.13 | 0% |
| gsm-st231.15M | 15 | 70 | 12.36 | 0.01 | 0% | 0.01 | 8.33% |
| gsm-st231.16M | 65 | 323 | - | 0.03 | 2.24%* | 2.77 | - |
| gsm-st231.17M | 38 | 173 | 90.00 | 0.01 | 0% | 0.17 | 0% |
| gsm-st231.18M | 214 | 1063 | - | 0.2 | 3.49%* | 429.36 | - |
| gsm-st231.19M | 19 | 86 | 38.23 | 0.01 | 0% | 0.02 | 6.25% |
| gsm-st231.20M | 23 | 102 | 123.00 | 0.01 | 3.23% | 0.03 | 4.76% |
| gsm-st231.21M | 33 | 154 | 42.06 | 0.01 | 0% | 0.07 | 3.24% |
| gsm-st231.22M | 31 | 146 | 80.36 | 0.03 | 0% | 0.06 | 0% |
| gsm-st231.25M | 60 | 273 | (604800) | 0.01 | - | 1.27 | 1.75% |
| gsm-st231.29M | 44 | 192 | 210.00 | 0.01 | 0% | 0.3 | 0% |
| gsm-st231.30M | 30 | 130 | 58.00 | 0.01 | 0% | 0.06 | 3.84% |
| gsm-st231.31M | 44 | 192 | 142.00 | 0.01 | 0% | 0.25 | 2.5% |
| gsm-st231.32M | 32 | 138 | 0.25 | 0.01 | 0 | 0.03 | 0% |
| gsm-st231.33M | 59 | 266 | (604800) | 56.00 | - | 1.14 | 0% |
| gsm-st231.34M | 10 | 42 | 5.05 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.35M | 18 | 80 | 52.00 | 0.01 | 0% | 0.01 | 0% |
| gsm-st231.36M | 31 | 143 | 230.00 | 0.01 | 0% | 0.06 | 7.69% |
| gsm-st231.39M | 26 | 118 | 95.00 | 0.01 | 0% | 0.04 | 4.55% |
| gsm-st231.40M | 21 | 103 | 15.00 | 0.01 | 0% | 0.03 | 5.56% |
| gsm-st231.41M | 60 | 315 | - | 0.02 | - | 0.61 | 6.15%* |
| gsm-st231.42M | 23 | 102 | 12.00 | 0.01 | 0% | 0.02 | 14.29% |
| gsm-st231.43M | 26 | 115 | 15.00 | 0.1 | 0% | 0.03 | 9.1% |

Table 4.2: Run-Times/Gaps of Modified instances

Table 4.2 report the experimental results on the *modified* set of instances. Our approach finds the optimal solution within a second for all of the instances but two (for which the average gap is 0.61%). *SMS* achieves an average gap of 3.03%: again, the highest computation time corresponds to instance `gsm-st231.18`, but it is considerably larger compared to that of the

*industrial* set (i.e. 429.36 seconds).

Finally, note that for the instances `gsm-st231.25` and `gsm-st231.33` no optimal solution is available, since in [3] the authors were only able to find a suboptimal schedule within the time limit of 604800 seconds (no details are given on how the dual process they propose converges to sub-optimal solutions). On the instance `gsm-st231.25` both the modular and the SMS approaches find the same solution as [3]. On `gsm-st231.33` the best period value found in [3] has value 52 and the *SMS* solver finds the same solution; our method however finds in one second a solution with value 47 and within 56 seconds a solution of value 46.

The results of our experimentation show that *CROSS*$^*$ is able to find very good solutions in a very short time (a few seconds). The method appears however much less effective in proving optimality: despite the optimal solution was found on most instances, an optimality proof was achieved only for 12.5% of them. We are currently investigating how to improve the efficiency of the proof of optimality. Note however that the optimality gap is so small that the method is very appealing even if used a heuristic.

### 4.6.2 Evaluating *CROSS*$^*$ solution quality on synthetic benchmarks

The second set of experiments targets a task scheduling problem over a multi-processor platform. The benchmark in this case contains 1200 synthetic instances with 20 to 100 activities, cycles in the graph and high concurrency (i.e. the precedence constraints allow many activities to run in parallel.). In a preliminary experimentation, this kind of graph structure appears to be the toughest to solve for our approach. The set of processor on the platform is represented in the generated instances as a cumulative resource with capacity 6 (representing the number of parallel threads). All activities have unary resource consumption.

Table 4.3 shows the average, best and worst gap over time between the

| 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 4.3: Solution quality

*CROSS*$^*$ solution and an a lower bound, computed as:

$$lb = \left\lceil \max\left(\texttt{IB}, \frac{\sum_{i \in \texttt{V}} d_i}{\text{CAP}}\right)\right\rceil$$

that is the maximum between the iteration bound `IB` (see Section 2.1.3.1) and the ratio between the sum of the execution times and the total capacity.

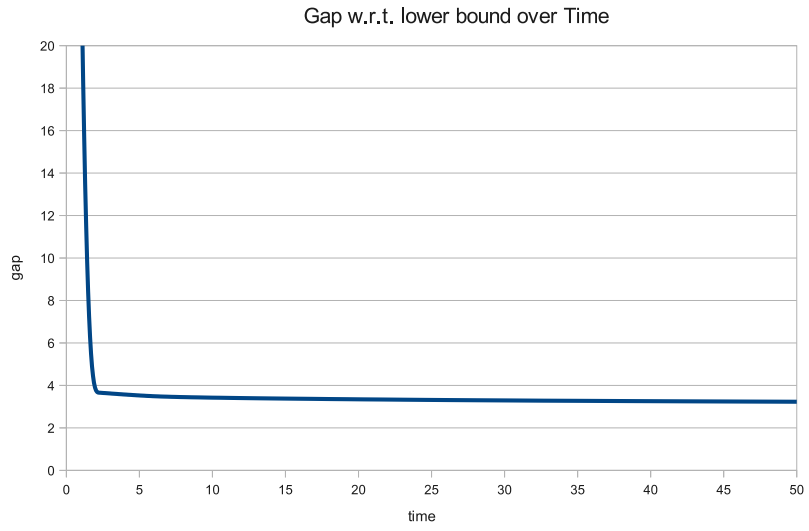


Figure 4.5: Graphical representation of the optimality gap improvement over time

As one can see in the table, *CROSS*$^*$ finds a solution which is about 3.7% distant from the lower bound value within one second. After 300 seconds the gap has decreased to 2.9%. Hence, even on this new instance set the solver

is able to find a high quality solution very quickly. After that, improvements are obtained much more slowly. Figure 4.5 provides a more detailed view of the progress of the gap value. Once again, we can conclude that our approach converges very quickly to period values close to the lower bound. The actual optimum lies somewhere in-between the two values and therefore is even closer.

### 4.6.3 *CROSS* and *CROSS*$^*$ vs Blocked and Unfolding Scheduling

Our third experimentation contains in first place a comparison between our approach, producing overlapped schedules, and the blocked approach that takes into account a single iteration. Since the problem is periodic and the schedule is iterated indefinitely over time, the latter method pays a penalty in the quality of the schedule obtained. A technique often employed to address this limitation and allow some inter-iteration parallelism is *unfolding* (see Section 4.1). Unfolding often leads to improved blocked schedules, at the cost of an increase of the instance size. In detail, we report the results for two separate sets of experiments, respectively evaluating the *CROSS*$^*$ and *CROSS* method.

**First Set of Experiments:** The first experimentation is performed on a set of 220 instances, divided into three classes: small instances, with (14 to 24 activities), medium-size instances (25 to 44 activities) and big instances (45 to 65 activities). We compared our approach with the *blocked* one and with unfolded scheduling, using seven different unfolding factors (referred to as *UnfoldX*, where $X$ is the number of unrolled iterations).

Table 4.4 shows the average gap between the above mentioned configurations and *CROSS*$^*$. The last column presents the average optimality gap over the whole experimental set. As expected, the worst gap is relative to the blocked schedule. Less obviously, the *UnfoldX* configurations tend to have an oscillatory behavior. Figure 4.6 depicts the relation between the gap (`Y` axis) and the blocking factor (`X` axis) for a selected instance with 30

activities. With an unfolding factor $u = 11$ the solver finds a solution with the same period as the overlapped one. Increasing the unfolded factor to 12 deteriorates, rather than improving, the solution quality. In general, it is false that increasing $u$ necessarily results in better solutions, making the determination of the optimal unfolding factor a non-trivial problem.

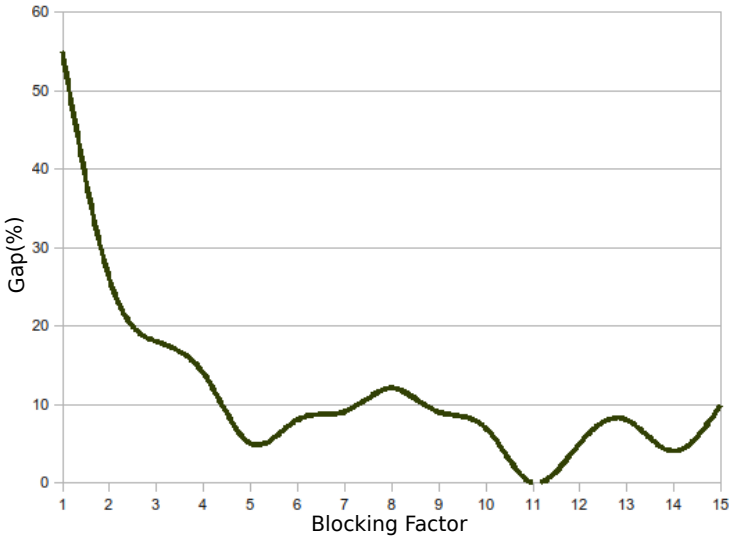


Figure 4.6: Optimality gap over blocking factor

Furthermore, there are cases where finding a schedule with the same quality of a periodic approach is not possible at all: such an example is provided in [84]. On the other hand, we recall that periodic schedules are dominated by K-periodic schedules, i.e. periodic schedules for a sequence of consecutive iterations. K-periodic schedules are known to dominate 1-periodic schedule in the presence of finite capacity resources [48]. Therefore, since the unfolding technique produces a *restricted* class of K-periodic schedules, there may be cases where and unfolded schedule is strictly better than any possible 1-periodic schedule. In summary: no strict dominance exists between the two approaches. Our experimentation shows however that, on the considered benchmarks, periodic schedules tend to be much better in practice.

**Second Set of Experiments:** The second experimentation is performed on a group of 200 synthetically generated project graphs with 5 to 25 activi-

| | *CROSS*$^*$ 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 4.4: *CROSS*$^*$ Unfolding set results

ties. The resource capacity is fixed to 5 times the average consumption. The durations are randomly generated so that around 10% of the activities in each graph are much longer than the remaining 90%. We compare the *CROSS* approach with blocked and the unfolded scheduling, for unfolding factors up to 5 (this value is sufficient to see the overall trend). We use the classical (and very effective) schedule or postpone search strategy for the blocked and the unfolded approach. For the *CROSS* one, we couple the Random Restart method from Section 4.5.2 with a binary search on $\lambda$ (using a time limit of 2 seconds on each iteration).

Table 4.5 shows the result of the comparison, grouped by the number of activities in the graph. In particular, the whole set of 200 instances has been partitioned into two subsets, respectively with 5-14 and 15-25 activities. The *Time* column shows the average time taken by the *CROSS* approach to reach the best solution found within 300 seconds. The table then reports, for all the considered approaches, the (average) quality gap w.r.t. the *CROSS* solution (i.e. $\frac{\lambda - \lambda_{CROSS}}{\lambda_{CROSS}}$) and an (average) $\Delta time$ value, representing the difference between the time taken by the considered approach and by *CROSS* to reach the best solution found within 300 second.

The results show the existence of a relevant trade-off between solution time and quality. The blocked scheduling approach represents one of the extremes, providing its best solution in a fraction of second, but with a quality gap as large as 216%. Increasing the unfolding factor leads to better $\lambda$ values, at the cost of increased solution time. The Unfold5 configuration

eventually manages to produce better solutions compared to *CROSS* (1.3% improvement), but the process is up to 6 times longer. Our approach seems to provide a very good compromise, providing high quality solutions in a short amount of time.

| N | Time | | Blocked | Unfold2 | Unfold3 | Unfold4 | Unfold5 |
|---|---|---|---|---|---|---|---|
| 5-14 | 2.75 | gap w.r.t *CROSS* | 216.37% | 98.75% | 51.71% | 24.52% | 10.99% |
| | | $\Delta$time for best sol | -2.74 | 1.4 | 2.7 | 4.47 | 4.9 |
| 15-25 | 9.19 | gap w.r.t *CROSS* | 68.37% | 20.49% | 7.10% | 0.88% | -1.3% |
| | | $\Delta$time for best sol | -9.18 | 5.6 | 47.7 | 58.92 | 63.45 |

Table 4.5: *CROSS* Unfolding set results

### 4.6.4 Throughput/Resource Trade-off investigation

Cyclic scheduling allows to improve the resource efficiency by partially overlapping different schedule iterations. In particular, it is possible to exploit the available resources to reduce the period, even when the makespan cannot be further minimized (e.g. due to precedence constraints). Loops in the project graph limit the degree of such an improvement (see Section 2.1.3.1): if the graph is acyclic, however, the throughput can be arbitrarily increased by augmenting the resource capacity. A large number of practical problems (e.g. in VLIW compilation or stream computing) are described by project graphs with a few cycles or no cycle at all. In such a case, identifying the optimal throughput/resource-usage trade-off is the primary optimization problem.

We performed an experimentation on two groups of 20 synthetically generated instances, respectively consisting of cyclic and acyclic graphs. Durations are again unevenly distributed, as described in Section 4.6.3. In order to investigate the throughput/resource trade-off, we solved a set of period minimization problems with different resource availability levels. In detail, we considered a single resource and activities in each graph were labeled with random resource requirements, following a normal distribution. The resource capacity (referred to as $CAP$) ranges between 4 times and 14 times the average consumption level: the minimum value is chosen so as to guarantee the

problem feasibility, while the maximum one is chosen to assess the solution quality in case of abundant resources.

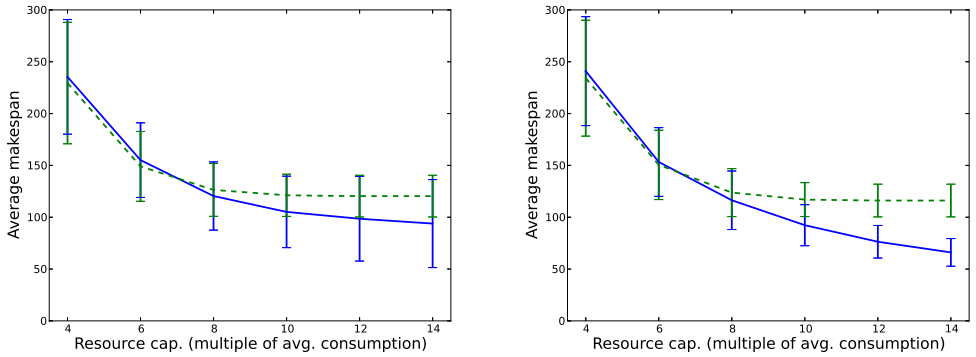


Figure 4.7: Makespan over resource capacity for cyclic (left) and acylic (right) graphs.

Figure 4.7 shows the average period after 300 seconds (the time limit was hit in all cases) obtained for different resource capacity values. The vertical bars report the corresponding standard deviation. The solid line corresponds to the *CROSS* approach and the dashed one to *CROSS*$^*$. Note that *CROSS* obtains considerably better results for higher capacity values, i.e. the scenario when we expected the highest benefit from allowing activities to cross iterations. The gap is larger for acyclic graphs, where the lack of loops enables to fully exploit the available resources.

The iteration period difference corresponds to a much larger gap in terms of "wasted" resource capacity, which can be assessed by measuring the value of the expression:

$$slack = CAP \cdot \lambda - \sum_{i \in \mathbb{V}} r_i \cdot d_i$$

The average slack values for this experimentation are reported in Figure 4.8, where the overall resource waste is shown to grow according to a roughly quadratic law for the *CROSS*$^*$ approach. The growth is much slower for *CROSS* (in fact, it is approximately constant for acyclic graphs). The amount of wasted resource capacity is an important measure of how efficiently the resources are used and in a practical setting directly translates to

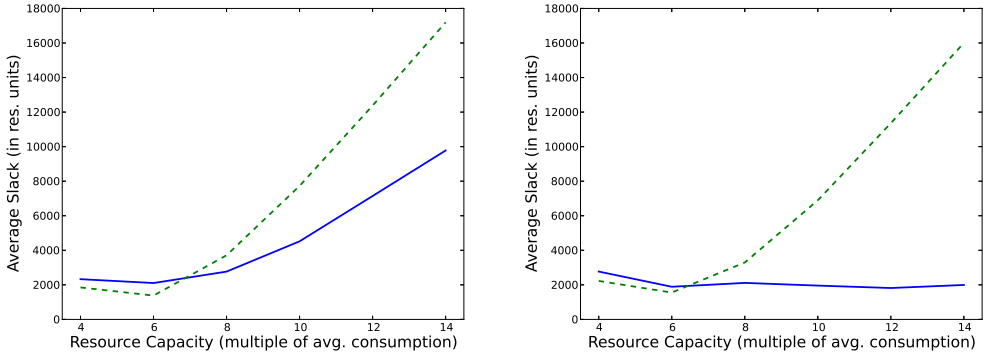


Figure 4.8: Average slack over resource capacity for cyclic (left) and acylic (right) graphs.

platform/machine costs.

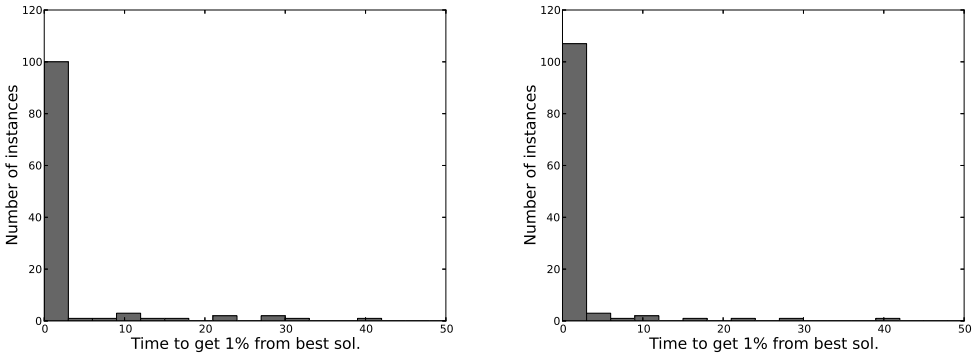


Figure 4.9: Instance distribution w.r.t. the time to get 1% from the best solution, for the *CROSS* approach (left) and *CROSS*$^*$.

Interestingly, the two approaches have comparable performance for small capacity values. This suggest that the time limit is not severely limiting the search effectiveness. This is a relevant remark, since we expected *CROSS* to be considerably slower in finding good solutions. More details are reported in the histograms from Figure 4.9, that show the instance count, grouped by the time (in seconds) employed by each method to get 1% close to the final best solution. As one can see, *CROSS*$^*$ is indeed faster on average, but both methods manage to provide high quality schedules in seconds.

# Chapter 5

# Putting *CROSS* into practice: the MPOpt-Cell Use Case

## A High-Performance Data-Flow Programming Environment for the Cell BE Processor

Multicore processors have been embraced by the whole computer industry, since hardware manufacturers finally realized that the effort required for further improvements of single core chips trying to increase instruction-level parallelism is no longer worth the benefits eventually achieved. Thread-level parallelism (TLP) is currently under the focus of microprocessor vendors by designing chips with multiple internal parallel cores (such as, for instance the NVIDIA CUDA Programming Guide [1] and the Intel Single-chip Cloud Computer [2]). However, this process does not automatically translate into greater system performance. The multicore solution exhibits for sure a superior peak performance, which can however only be achieved at the cost of significant software development effort [99] [60].

In the last two decades, the international scientific community has aimed

[1] `http://developer.download.nvidia.com/compute/cuda/1_0/NVIDIA_CUDA_Programming_Guide_1.0.pdf`

[2] `http://www.intel.com/content/www/us/en/research/intel-labs-single-chip-cloud-computer.html`

at designing computer languages and tools to support application design/-port and performance tuning for such parallel architectures [55]. Modern multicore processors are more and more limited by communication rather than computation [91]. However, communication or dependency information can often be difficult to determine at compile time. To effectively exploit this parallelism it is often necessary that the programmer manually re-structures the application in such a way that communication and task dependencies are explicit [78]. This is the case of streaming programming models, which have demonstrated significant performance advantages over standard (automatic) parallelization techniques in domains such as signal processing, multimedia and graphics [101, 22].

Synchronous Data-Flow (see Appendix 3.2) is a representative example of such model of computation. Stream programs provide a vast amount of parallelism, which makes them well-suited to run efficiently on multi-core architectures, in particular on distributed memory architectures where communication is overlapped with computation (i.e. stream processors) [96].

The stream processing abstraction comes with some drawbacks: algorithms that cannot be naturally mapped to the paradigm must often be completely rewritten [70, 47]. Moreover, current programming practices and performance demands dictate that the programmer chooses a low-level language in which he can explicitly control the degree of parallelism and arduously tune his code for performance [89]. A good stream processing abstraction should reflect the underlying hardware model to properly map desired computation to the target hardware architecture [92]. Programming frameworks and middleware support for streaming applications (i.e. compilers, libraries, tools, runtime, etc.) need to be strongly specialized for the target architecture. The bottom layers of the framework stack (i.e. the compiler backend and runtime system) have to be highly tuned for the target hardware resources.

At the topmost levels of the software development stack, on the contrary, the programming abstractions should be as generic and architecture-agnostic as possible to increase ease of use and productivity. Still, knowledgeable programmers should be allowed to specialize the compilation flow for the

target architecture by providing "hints" to the compiler. Simple language features should thus be designed to achieve both goals.

*MPOpt-Cell* (presented in [39]) is a highly optimized framework for efficient development and execution of stream applications on the CELL BE Processor[3]. Cell is a heterogeneous multicore architecture composed by a standard general purpose microprocessor (called PPE), with eight coprocessing units (called SPEs) integrated on the same chip [50]. Cell has already demonstrated impressive performance ratings in computationally intensive applications and kernels. It has been exploited by several application domains, ranging from gaming to high performance computing [79, 104]. Thanks to its innovative architectural features, this architecture has been also adopted in the embedded system domain: e.g. in a work of Daniel Stasiak et al. [95] and in the Toshiba Spurs Engine[4].

The real-time requirements frequently found at the heart of many streaming applications promote predictability as a first-class design goal. To achieve this goal MPOpt-Cell approach relies on strong off-line optimization and static scheduling. This allows to provide both robustness and high performance, different from several other related techniques based on dynamic scheduling and simple heuristics. One of the limitations of static approaches is that they often trade robustness for performance due to the introduction of schedule over-constraining. We address this issue by introducing additional precedence relations in the form of low-overhead fake data communications. This allows the schedule to stretch depending on the actual execution time of tasks at runtime.

The MPOpt-Cell framework[5] is a software chain structured in three separated components:

1. A runtime system (RTS) which enables efficient execution of SDF ap-

[3] `https://www-01.ibm.com/chips/techlib/techlib.nsf/products/Cell_Broadband_Engine`

[4] `http://www.semicon.toshiba.co.jp/eng/shared/\pdf/SpursEngine_leaf_e_2008-11.pdf`

[5] The framework is described at `http://mpopt.ing.unibo.it`

plications on the Cell processor. The RTS manages all the aspects related to efficient hardware resource management, like task dispatching, data movement and task synchronization. Resource management adopts a centralized approach, being all decisions (e.g. task activations and DMA commands) issued by the PPE. This allows to effectively exploit advanced architectural features like double buffering, task migration and memory management, thus leading to a more efficient and flexible management of the overall system.

2. A compiler backend which leverages on *CROSS* solver (see Chapter 4) to generate accurate scheduling and mapping decisions for the target SDF application, optimized for maximal throughput.

3. A simple and intuitive programming interface based on standard C augmented with annotations. A set of compiler directives have been identified, that capture the key abstractions of SDF. The programmer can easily describe SDF Graphs (SDFG) by enclosing code portions within our custom directives and specifying the flow of data among tasks. The compiler automatically extracts the tasks and the data-flow.

The three components, integrated in an unique application development framework, achieve the two-fold goal of easing application development and obtaining high performance from Cell-based processors.

In the following sections we report the state-of-the-art framework supporting streaming computation and then we describe in details the MPOpt-Cell framework chain.

## 5.1 Related Work

In the past few years several programming models and tools aimed at easing the task of efficiently mapping parallel applications on top of the Cell processors have seen the light.

*Sequoia* [38] is a programming language that abstractly exposes hierarchical memory in the programming model and provides language mechanisms to describe communication vertically through the machine and to localize computation to particular memory locations within it. This execution model is particularly well suited to data parallel computations. It enforces strict locality of computation, since tasks run in isolation on a processor and can only access data from within local memories. On the other hand, inter-node communication is much more complicated and much less performance efficient, since it has to take place through dedicated sub-tasks. This, in turn, makes it very difficult to model different kind of parallelism (such as those targeted by this work) with *Sequoia* constructs.

*Offload* [29] is a programming model from Codeplay for offloading parts of a C++ application to run on the SPEs of the Cell BE. *Offload* provides an efficient mechanism to automatically generate code for different ISAs in a heterogeneous MPSoC, and to orchestrate data transfers in a transparent manner to the programmer. However, *Offload* constructs are not specific for data-flow applications, which can only be modeled at the price of significant coding effort. Moreover, no support for efficient scheduling of streaming tasks is natively provided by *Offload*.

*Cell-Space* [76] is a framework for developing streaming applications for the Cell BE. Developers construct applications by means of data flow components that are then scheduled to PPE/SPEs by a runtime system acting as a streaming communication interface. Different from our user-friendly annotation-based programming interface, with *Cell-Space* developers are required to construct data flow applications from a library of components which presents an application as an XML description of a data flow graph. In our framework a front-end compiler abstracts away these details of the internal representation of a data-flow application, thus requiring much less programmer involvement. Furthermore *Cell-Space* runtime system ensures load balancing through dynamic scheduling techniques, which however cannot guarantee robust and predictable execution times such as ours essential for real-time streaming applications.

StreamIt [101] is probably one of the most representative examples of a streaming language based on SDF available for the Cell processor. The StreamIt project provides a source language, a publicly available compiler, and a benchmark suite. Writing a *StreamIt* program, however, requires significant effort. Outlining all the tasks and communication channels that describe a streaming computation is left to the programmer. Moreover, the stream structures supported by StreamIt are limited to three representative patterns, namely pipelines (i.e., sequential composition), split-joins (i.e., parallel composition), and feedback loops (i.e., cyclic composition). MPOpt-Cell coding style based on annotations provides a much easier and expressive interface to data-flow programming than *StreamIt* constructs.

The most wide-spread programming model based on code annotations is undoubtedly *OpenMP*[6]. While an *OpenMP* implementation for the Cell BE has been provided by authors of [77], the standard *OpenMP* model of computation is mainly focused on data parallelism at the loop level, and thus is not suitable to describing streaming parallelism. Still, the appealing easy-to-use coding style of *OpenMP* has led several researchers to extend the basic interface with custom constructs to describe data-flow parallelism.

Streaming extensions for *OpenMP* have been proposed within the *ACOTES* project [75]. Similar to what we propose here, the *ACOTES* programming model is based on a small set of key compiler directives that allow a programmer to identify *streaming tasks*, *streams* and *ports*. The focus of the optimization engine in *ACOTES*, however, is on loop transformations based on the polyhedral model for efficient loop parallelization and vectorization. We do not target data parallelism in our work, and our optimization framework is rather aimed at guaranteeing a task schedule which maximizes throughput.

*Cell SuperScalar* [10] is another project which uses compiler directives as code annotations to model data-flow computation. The user has to identify the parallel parts of the application, which are then automatically offloaded. *CellSS* focuses on high-level parallelism and maintains a data flow graph of

[6]see OpenMP C and C++ API v.3.0. at `www.openmp.org`

pending tasks. Among all the cited approaches, *CellSS* is probably the most closely related to the MPOpt-Cell one, and for this reason we chose it as a direct term of comparison for our experiments.

### 5.1.1 Target Architecture

Figure 5.1 shows a pictorial overview of the STI Cell Broadband Engine Hardware Architecture. The Cell BE is a non-homogeneous multi-core processor which includes a 64-bit PowerPC processor element (PPE) and eight synergistic processor elements (SPEs), connected by an internal high bandwidth Element Interconnect Bus (EIB).

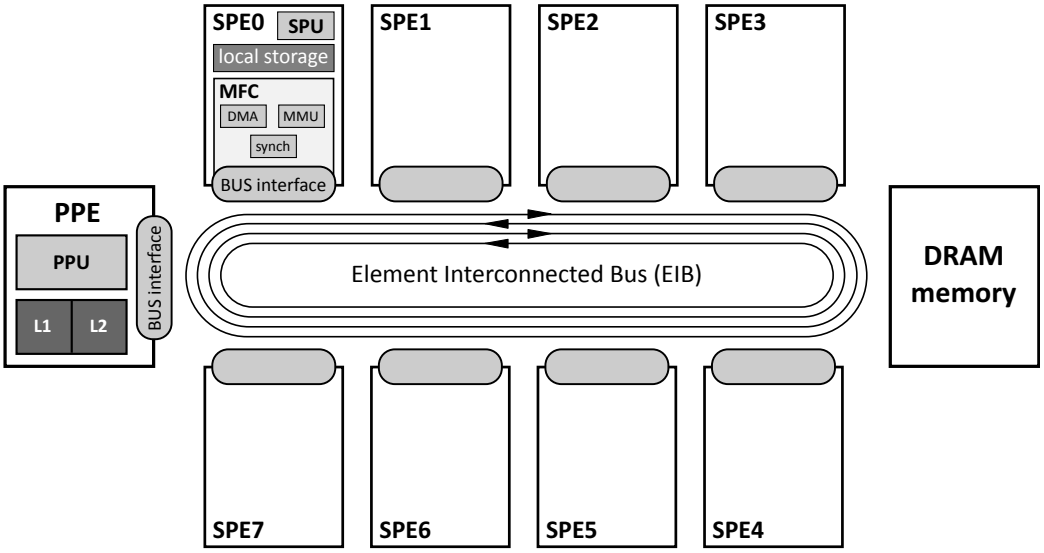


Figure 5.1: Block diagram of the Cell BE

The PPE is dedicated to the operating system and acts as the master of the system, while the eight synergistic processors are optimized for computation-intensive applications. The PPE is a multithreaded core featuring two levels of on-chip cache. The SPE is a computation-intensive coprocessor designed to accelerate media and streaming workloads. Each SPE consists of a synergistic processor unit (SPU) and a memory flow controller (MFC). The MFC includes a DMA controller, a memory management unit (MMU), a bus interface unit, and an atomic unit for synchronization with

other SPUs and the PPE.

Efficient SPE software should heavily consider memory usage, since the SPEs operate on a limited on-chip memory (only 256 KB local store) that stores both instructions and data required by the program. The local memory of the SPEs is not coherent with the PPE main memory, and data transfers to and from the SPE local memories must be explicitly managed by using asynchronous coherent DMA commands.

## 5.2 Framework Overview

In this section we describe our approach[7] to mapping a data-flow application on the Cell processor. The overall flow of our framework is depicted in Figure 5.2.

The interaction between framework components is based on three hardware-software models:

1. a *Architecture Description Language (ADL)*, providing an abstract view of the target hardware platform

2. a *Graph Description Language (GDL)*, providing a unified notation for distinct graph-based software design methodologies. *GDL* is used as an input format for the CROSS Solver, which produces as output an enhanced version of the same to which we refer to as *GDL+*. *GDL+* contains information about the affinity between tasks and SPEs as well as scheduling decisions.

3. a *Mapping Description Language (MDL)*, handling features that affect both hardware and software domains.

We consider a layered bottom-up approach, where three main building blocks incrementally abstract away architectural details from the developer's view.

[7]More details can be found at `http://mpopt.ing.unibo.it`

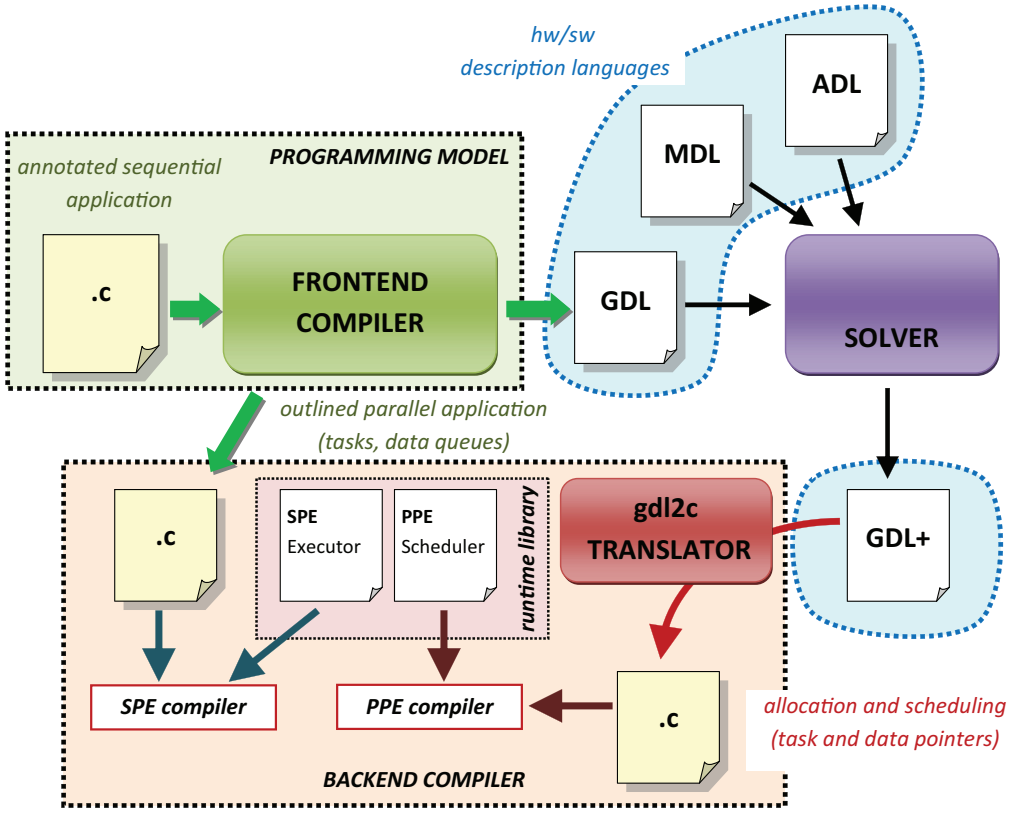


Figure 5.2: Compilation flow

### 5.2.1 Runtime System

The bottom layer consists of a target-specific **Backend Compiler**, made up of static and dynamic components. At the heart of our execution engine there is a efficient *runtime system* (RTS) which encapsulates hardware specificities and is in charge of ensuring efficient task management, data transfer and synchronization. The RTS effectively maps computation (tasks) and communication (data-flow) on top of hardware resources, but requires that a schedule for task execution is provided as an input.

The runtime system introduces an abstraction layer which hides the difficulties of the parallel architecture, such as load balancing, synchronization, and communication between the main components of the streaming application. It orchestrates the execution of all the components in the data flow

graph application and provides both streaming and event communication primitives to the components. Moreover, using centralized resource management, the runtime can dynamically balance the load over the available SPE processors. The runtime system (RTS) leverages different structures for its execution, namely SDFG tasks, queues, executors and a resource manager (More details on this structures in Appendix B.1).

### 5.2.2 CROSS Solver

To ensure the most efficient task schedule for the considered application we developed a solver block operating on top of the RTS. The solver used in this approach is *CROSS* (see Sec. 4). The computed solution is guaranteed to satisfy user defined constraint which may be specified (e.g. a minimum throughput requirement).

The solver block requires an input description of hardware, software and cross-domain data in the form of *ADL*, *GDL* and *MDL* documents. A preprocessing step transforms the input SDFG into the corresponding *Homogeneous SDF Graph* (HSDFG) (see [14]), with unary rates over each arc. Eventually, the homogeneous graph is transformed into a *perfect-rate* HSDF graph (see [84]). Both transformations involve polynomial-time algorithms.

The CP *model* is based on modular algebra and is described in Chapter 4. Beside the application model, the Constraint Programming model takes into account the architectural components by describing resource constraints and architectural features. In this framework, we describe the Cell model as a *single* cumulative resource of capacity equal to the number of SPU.

As stated in Section 4.2.1 the restrictions on the resources are modeled through a cumulative constraint; the constraint prevents the solver from finding a number of concurrent executions higher than the capacity of the resource.

The basic idea of the solving process is to model the effects of mapping and scheduling choices by means of graph modifications. The solver block

assumes a self-timed scheduling policy[8] and the execution order is determined on the modified graph. In details, we modify the graph adding edges that force precedence relations between scheduled tasks so as to prevent tasks from competing for the SPEs. The runtime scheduler processes these edges as fake data communications between actors. Note that differently from the disjunctive approach presented in Chapter 3, where the graph is modified during search, in the MPOpt-Cell solver the augmented graph is constructed when the search stops with a feasible solution.

The self-timed execution of a periodic application modeled with a synchronous data-flow graph consists of two different phases (see [44]): the *transition* phase and the *periodic* phase. The former appears only once at the beginning of the execution while the latter is periodically repeated *ad infinitum*. In this context, a static scheduling strategy [14] should include a different schedule for each execution phase. Conversely, thanks to the use of modular algebra, our solver produces a single static schedule that represents both phases. The transient and the periodic phase schedules are easily inferred by considering the iteration values of each task in the solution. In fact, the solution includes the modulus value $\lambda$ and the set of start times $\mathtt{s}_i$ and iteration numbers $\beta_i$. Before the application enters the periodic behaviour, some of the tasks (namely those for which $\beta_i > \omega$, with $\omega$ as current execution iteration) simply do not start.

The type of solution provided by the CROSS solver[9] naturally suggests the use of a time-triggered schedule; on the opposite the MPOpt-Cell runtime system is based on a self-timed policy. The motivation for this choice is that modeling the precedence relations with arcs on the graph allows the solver to obtain flexible solutions. Moreover, if the actual execution times are shorter than the expected times it allows the runtime to deliver higher throughput.

At the end of the search, a novel algorithm translates an assignment of start and iteration variables to graph modifications and constructs the new

[8]Where each activity executes as soon as all of its input data are available; see also Appendix 3.2.1

[9]We recall that CROSS solver computes static-time schedules. The ordering decision are extracted after the search.

GDL+ document. Therefore the output of the MPOpt-Cell solver consists of one "enhanced" *GDL* document (referred to as *GDL+*), which is essentially the input *GDL* with additional allocation and scheduling information. Namely, we specify the node *affinity* that references a resource unit chosen for task allocation or we add *dummy* arcs (*precedence* arcs) used to force task ordering on a specific resource. The algorithm operates into two steps:

- Allocation: the algorithm extracts from the solution the *mapping* association for each node (even if the allocation problem is not considered in *CROSS*). In fact, the use a single shared resource of capacity equal to the number of SPU guarantees that the maximum number of concurrent tasks is equal to the number of SPU (see Section 2.1.1 for details).

- Scheduling: the algorithm inserts into the original graph new precedence arcs (called $\widehat{arcs}$) that guarantee the predicted execution. These $\widehat{arcs}$ guarantee the correctness of both the transient and the periodic execution phases.

Note that counter-intuitively an $\widehat{arc}$ $(i, j)$ can have a negative number $\dot{\delta}$ of tokens (i.e. $\delta_{i,j} = \dot{\delta} < 0$). This means that the source activity $i$ has to execute at least $\dot{\delta}$ times before the first execution of sink activity $j$.

### 5.2.3 Backend Compiler

This section describes MPOpt-Cell *Programming Model*, namely the language constructs we provide to express data flow semantics at code level and the *front-end compiler*.

MPOpt-Cell adopts a novel programming model based on the familiar C language, augmented with some compiler directives that allow the programmer to easily describe data-flow computation. The front-end compiler automatically outlines a parallel C program – describing tasks and data streams of the SDFG – which is sent to the backend Compiler.

Our approach borrows from the coding style of the well-known shared memory programming model OpenMP. OpenMP provides a set of compiler

directives that allow to outline parallelism and work sharing within a standard (sequential) C[10] program.

A compiler is in charge of transforming the annotations into code which spawns parallel threads at runtime and manages data sharing among them.

Therefore we define a set of directives which provide the needed abstractions to model a SDFG (i.e. actors, incoming/outgoing arcs, etc.). The role of our front-end compiler is twofold. First, the directives inserted by the programmer are processed so as to generate C code which will be compiled for the SPEs. Second, the structure of the SDFG described through the custom directives is extracted into a *GDL* representation for the solver to process. The compiler automatically extracts the SDFG representation required by the Solver.

The main benefits of this approach reside in an increased ease of use and productivity, since a programmer does not have to learn new language constructs or a brand-new programming language. More details on the programming model are presented in Appendix B.2.

## 5.3 Experimental Results

The framework has been tested using three representative algorithms from the multimedia domain, namely a *FFT kernel*, a *block matrix multiplication* and a *FM radio demodulator*.

We compared our approach to Cell Superscalar (CellSs), a framework based on code annotation that provides a compiler and a runtime library for Cell BE platform programming; since the approach is quite similar to MPOpt-Cell, CellSs is a good candidate for comparative benchmarks.

All experiments were executed on a PlayStation 3 (3.2 GHz Cell) running Yellow Dog Linux 6.0. Reference implementations for the benchmark algorithms are available on StreamIt website: the serial code has been annotated using both Cell Superscalar and MPOpt-Cell annotations, and it has been compiled activating maximum optimization level (O3) for both compilation

[10]or C++, or Fortran

chains.

Experimental results were calculated considering a set of 50 program executions: each execution runs the algorithm and collects statistics for a stream of 200 input data. Consequently, the calculated throughput on each run is equal to the ratio 200/execution time.

The evaluation of framework performance follows two fundamental metrics: *predictability* and *performance*.

Predictability is essential in the presence of hard real-time constraints: in this case, local throughput variations may result in violation of the deadlines, making the computation useless or even harmful.

On the other hand, a predictable schedule takes the risk to be extremely conservative, with an outcome of poor performance. For this reason we also decided to evaluate the overall performance of our approach, in terms of average throughput, making a comparison with Cell Superscalar results.

### 5.3.1 Predictability evaluation

Predictability has two main facets: the capability to produce a regular throughput with minor local variations and the ability to predict whether throughput constraints may be satisfied or not before deployment stage.

In the case of real-time applications it is essential to ensure the respect of definite deadlines and consequently a wide variability range is unacceptable.

Figure 5.3 depicts the results of a first experimental set: it graphically represents the variability of the measured execution times on both CellSs (left side) and MPOpt-Cell (right side).

A numerical scale was not reported in the chart because absolute values are not significant to understand the variability dynamics.

The distance between a point and the chart center shows the execution time of a single experiment. Hence, all points equidistant from the center represent different experimental values with the same execution time. Each application is depicted by a line whose points map the values measured at subsequent application launches. The circularity factor of each line directly traces the regularity of the throughput for the correspondent application:

the closer the line to a perfect circle, the more regular execution times.

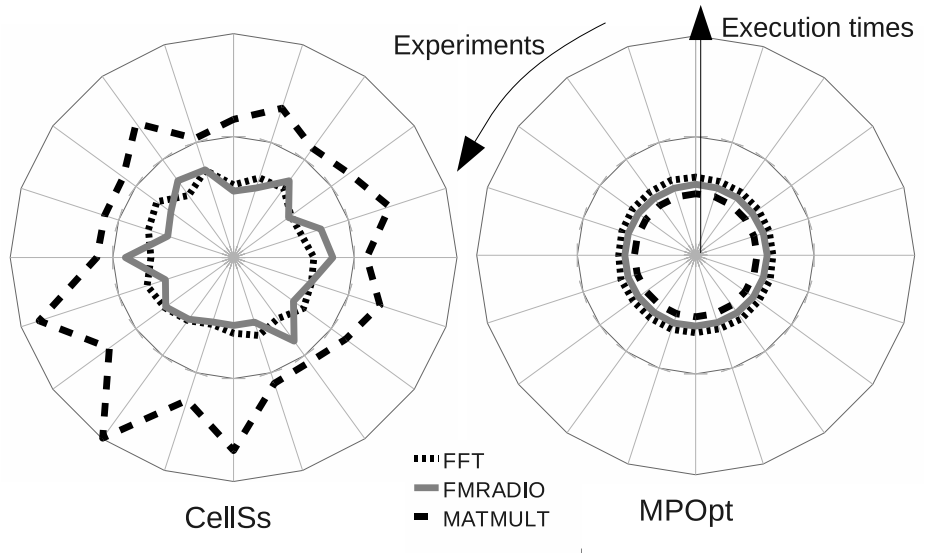


Figure 5.3: Variability of experimental results

The experimental values related to MPOpt-Cell have a good circular factor, namely a limited variability. This aspect can be attributed to the high determinism derived by the use of a static scheduler.

Conversely, Cell Superscalar provides a dynamic scheduler which is based on a task dependency graph which sometimes allows locally better results, but exhibits an unpredictable runtime overhead. Therefore, the bad circular factor of corresponding lines highlights this behaviour.

Figure 5.4 shows the result for a second experimental setup: this was conducted by feeding the solver with task worst case/best case execution times, instead of average ones.

The off-line predicted throughput values are compared with the measured one at runtime, which is based on a schedule obtained by average execution times. In detail, each column in the figure refers to a different schedule, computed by taking into account worst case/average case/best case execution times; for each benchmark, the left- and right-most columns present off-line predicted throughput values of the considered schedule, while the middle column reports an experimentally measured value.

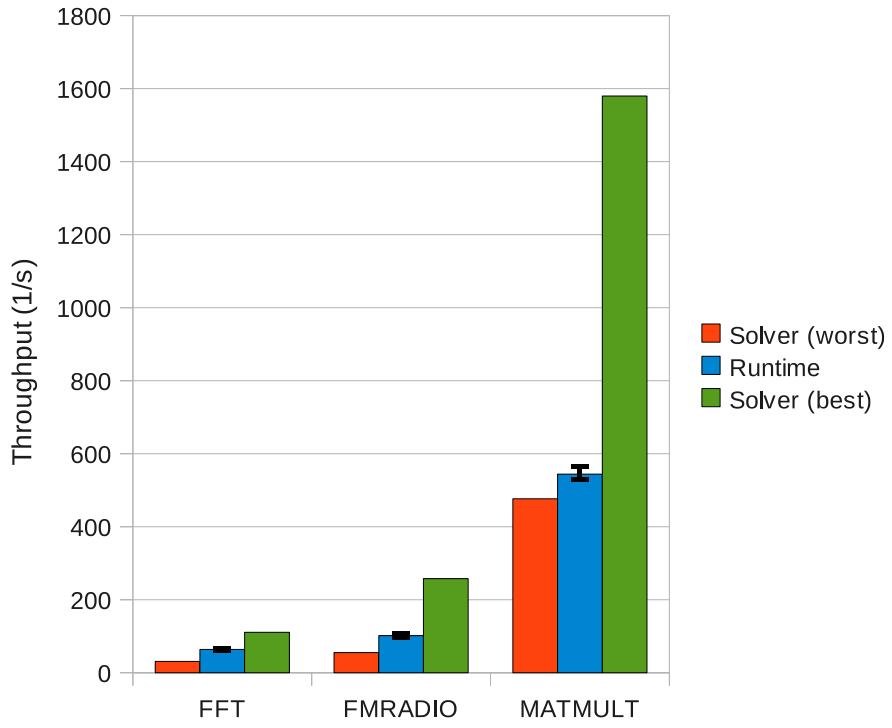


Figure 5.4: Solver and runtime throughput comparison

One can see that the worst case execution times (WCET) and the best case execution times (BCET) schedules establish strong bounds for the eligibility range of runtime throughput values (depicted in Figure 5.4 by overlying vertical lines). In particular, the solution provided when all tasks are assumed to execute with the BCET provides a conservative upper bound on the throughput. More interestingly, the WCET solution provides an estimate of the best safe throughput value, i.e. the tightest throughput requirement the system can met.

Moreover, by providing the solver with WCETs, the predicted value is very close to the actual one, assessing the high accuracy of the adopted model. This allows to check in the early steps of the design process if the performance requirements can be met or an upstream optimization stage is required. Overall, this can potentially reduce the development time.

### 5.3.2 Performance evaluation

Figure 5.5 shows the results of a bare performance comparison (in terms of throughput) with Cell Superscalar. The figure bars represent the throughput values calculated as a mean of 200 program iterations: overlying vertical lines outline the interval of samples variability, i.e. the range of measured throughput values. The endpoints of these lines represent the maximum and minimum values originated by experimental results.

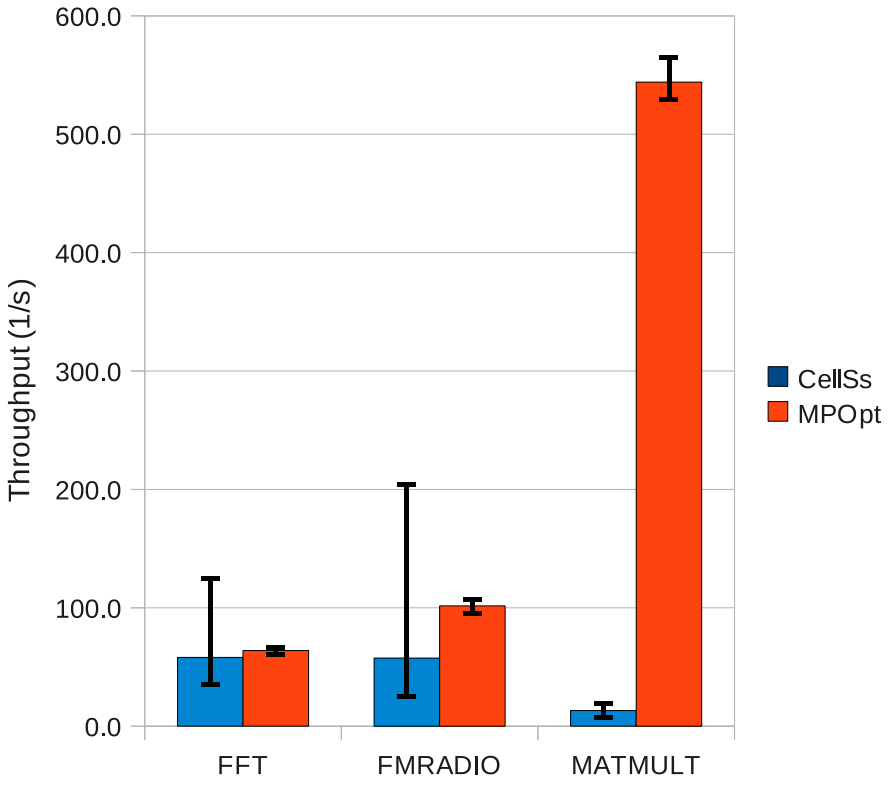


Figure 5.5: Performance evaluation: Cell Superscalar and MPOpt-Cell

The throughput mean value is an assessment of the performance one can expect with a high number of iterations: it is particularly significant in a scenario characterized by continuous data streaming and in this context it can be naturally used as a performance indicator. Our approach is comparable to Cell Superscalar on FFT, it has better performance on FM radio and even

better performance on matrix multiplication.

The runtime performance of MPOpt-Cell framework is due to a set of concurrent factors: (1) the effective use of the *SDF model*, that fully exploits the data parallelism of streaming applications and avoids unnecessary synchronization barriers; (2) the *static scheduling* technique, which provides low-overhead resolution of possible resource conflicts by compile time allocation and scheduling; (3) the *double buffering* technique, that reduces the overhead of data transfers; (4) the use of an *optimized schedule*, which is based on appended precedence relations and allows the framework to stretch to accommodate actual execution times.

The low performance measured on the matrix multiplication benchmark using CellSS (40:1) motivated a deeper analysis. The performance gap appears to be attributable to framework design issues. Since CellSS does not natively support a streaming model, it is necessary to insert a synchronization barrier after the computation of the submatrices multiplication at each iteration step: in this way we can obtain the correct result without using ad-hoc implementation tricks. To investigate the impact of such limitation, we ran some experiments on MPOpt-Cell by introducing an unnecessary barrier. We observed that the barrier represent an important limitation to the application throughput, indeed with this barrier the throughput achieved on MPOpt-Cell is halved. Still, the barrier alone does not explain the remaining performance gap (20:1). A further conjecture involves optimization issues related to data transfers, which are treated by CellSS using a locality-aware heuristic.

# Chapter 6

# Conclusions

We have proposed a number of CP approaches for Resource Allocation and Cyclic Scheduling problems, enforced by original filtering algorithms.

In particular, in Chapter 3 we presented a CP-based method for allocating and scheduling HSDFGs on multiprocessor platforms; to the best of our knowledge this has been the first CP-Based complete approach for the target problem. The core of the system is a global throughput constraint embedding an incremental extension of the computation procedure which proved to be crucial for the performance. The method obtained promising results on realistic size graphs.

Then, in Chapter 4 we have proposed a constraint approach (*CROSS*) to solve cyclic scheduling problems, based on modular arithmetic. In particular, we have devised global constraints to model temporal dependencies (the Modular Precedence Constraint, *ModPC*) and resource restrictions (The Global Cyclic Cumulative Constraint, *GCCC*). For both of them, we devised original filtering algorithms. We have also described a restricted version of our solver (*CROSS*$^*$) where we rely on a specific assumption to model the resource restrictions via traditional cumulative constraints, rather than via the GCCC.

The approaches have been tested extensively on industrial as well as synthetically generated instances. The methods were able to return solutions very close to the known optimum or to a lower bound in a matter of seconds

(or a fraction of second), on problems of practical size. Depending on the considered benchmarks, the approaches either outperformed existing heuristic or unfolding-based techniques, or provided a very good compromise between solution quality and time. All this makes our methods one of the best available solvers for cyclic scheduling problems with cumulative resources.

Future research directions include improving the effectiveness of the proof of optimality, which is currently a weak point of the method. A second, very interesting, research topic concerns the design of a more effective search strategy for the unrestricted *CROSS* approach. Since the start times assigned by our Random Restart strategy depend on the period upper bound $\overline{\lambda}$, they tend to lead to solutions with period quite close to $\overline{\lambda}$ itself, thus making the optimization process slower. This could be addressed by exploiting ideas from the Precedence Constrain Posting technique, developed for non cyclic scheduling problems.

Finally, in Chapter 5 we presented MPOpt-Cell, a complete environment for enabling efficient development and execution of streaming programs on the Cell Broadband Engine processor. Its infrastructure leverages an intuitive programming model based on compiler directives which allows designers to easily describe streaming applications. Compile-time and runtime optimizations ensure an efficient execution of the application through a finely tuned mapping of tasks and data-flow on top of available hardware resources. Furthermore, the compile-time optimizations are computed through the *CROSS*$^*$ solver, described in Chapter 4. Experimental results demonstrate the efficiency of the framework.

Part of the developed work has been published on international conferences [21, 18, 68, 19, 20, 39]

# Appendix A

# Constraint Programming

Constraint Programming (CP) [34, 88] is a programming paradigm used to solve hard combinatorial problems. It is currently applied with success to many domains such as planning, vehicle routing, configuration, scheduling and bioinformatics [5, 7, 8, 13].

The key concept of constraint programming is the clear separation between constraint modeling and constraint solving.

A constraint model is defined in terms of variables and constraints. Each variable $X_i$ has an associated domain $D_i$ containing values that the variable can assume (the notation for linking variables and domains is $X_i :: D_i$). Constraints define combinations of consistent assignments (i.e., a subset of the Cartesian product of the variable domains). The model might have an objective function defining a (possibly partial) order in the solution space.

Once the constraint model is stated, constraint solving is started by interleaving propagation and search. The search process enumerates all possible variable-value assignments (possibly guided by a proper variable and value selection heuristics), until we find a solution or we prove that none exists. To reduce the exponential number of variable-value pairs in the search tree, domain filtering and constraint propagation are applied at each node of the search tree. Domain filtering operates on individual constraints and removes provably inconsistent domain values. Since variables are involved in several constraints, domain updates are propagated to the other constraints whose

filtering algorithms are triggered and possibly remove other domain values.

As domain filtering is local to each constraint, it is a common practice in Constraint Programming to define the so called global constraints, that compactly represent combination of elementary constraints, but embed more powerful filtering algorithms exploiting a global view.

As an example consider the $\mathit{AllDiff}([X_1..X_n])$ constraint [87]. Declaratively it is equivalent to a set of pairwise inequalities ($X_i \neq X_j, \forall i \neq j$). However, by reasoning globally, it infers more deletions in general.

As a simple example, consider the following variables and their domain:
*X :: [1,2,3]*
*Y :: [1,2]*
*Z :: [1,2]*
and the following constraint: $\mathit{AllDiff}(X, Y, Z)$.

By considering the set of elementary constraints ($X \neq Y, Z \neq Y, X \neq Z$) the propagation[1] cannot remove any value, while the AllDiff global constraint removes values $[1, 2]$ from $X$ as they should be assigned (no matter how) to $Y$ and $Z$. The *AllDiff* constraint leverages network flow algorithms to perform the described filtering in polynomial time [87].

Constraint propagation is not complete. This means that if a value is removed by a filtering algorithm it is proved to be infeasible. Instead, if a value is left in the domain of a variable, it can happen that it does not belong to any consistent solution. For this reason, tree search is employed to explore the values left in the domain. At each node of the search tree, constraint propagation is triggered thus interleaving propagation and search. As far as search is concerned two main factors affect the solution process: the early evaluation of a partial solution and the variable-value selection strategy. The former is usually performed via an (upper/lower) bound computation, the latter is an heuristic function that guides the search.

In this thesis we use Constraint Programming as underlying programming paradigm.

[1] In this example we assume that arc consistency is enforced; note that there exist stronger consistency techniques that are, however, rarely used in constraint solvers.

# Appendix B

# MPOpt Framework Implementations

## B.1 Runtime System Structure

Several application developing aspects make streaming application programming challenging on the Cell processor. Moreover, heterogeneity of this platform makes more complex the overall scenario adding another difficulty layer. When developing a Cell application, the developer has to find efficient solutions to several questions raised by the following issues:

1. Resource utilization, scheduling and workload load-balancing
2. Communication and synchronization.
3. Memory management with distributed memory and inter-core data transfers;

The default programming environment for the Cell processor provides all the means to effectively set the low level knobs of the architecture, but it lacks high-level programming support. The Cell development libraries give indeed the capability to use a range of different options for a given purpose, but it is not so straightforward to choose the most suitable for the target application case. Programmers are forced to consider among many design

alternatives to achieve good performance. For instance, programmers must carefully consider the advantages and drawbacks of interrupts versus DMA, the optimal size of code for execution on SPEs, how to partition applications in components, how to schedule jobs on SPEs, etc. In addition to mentioned issues multi-buffering schemes should also be devised, which efficiently overlap computation and data transfers. If a programmer had to explicitly handle all of the above mentioned development issues he would really be involved in low-level and architecture-specific details. This could easily lead to wrong or non-optimal implementation decisions, which in turn result in poor performance, as all of the mentioned issues are crucial for efficiency. Moreover, finely tuning an application to a given target architecture compromises its portability.

As stated in Section 5.2.1, our runtime system introduces an abstraction layer which hides the difficulties of the parallel architecture, such as load balancing, synchronization, and communication between the main components of the streaming application. It orchestrates the execution of all the components in the data flow graph application and provides both streaming and event communication primitives to the components. The runtime system (RTS) leverages different structures for its execution, namely SDFG tasks, queues, executors and a resource manager.

#### B.1.0.1 SDFG Tasks

The tasks represent the functional core of SDFG nodes. A task is a self-contained application part that executes on a SPE, performs some computation on input data and produces output data. In streaming applications these computations are commonly indicated as *kernels* or *filters*. Our runtime provides local addresses for input and output data to the SPE executing the target kernel. Tasks are described by an unique identifier, the number of input and output queues they are linked to, the number of tokens required on each input queue to initiate execution, the number of tokens that each execution produces in the output queues, the ordering in which queues are consumed/filled, and the number of iterations they have to do at each

execution. Tasks have also a status, which can be:

- Ready: the task is ready to be scheduled, i.e. input local buffers contain data for computation, output local buffers are ready to receive data.
- Running: the task has been scheduled and its execution is taking place.
- Waiting: the task cannot be scheduled because either input queues do not contain data, or output queues are not ready to receive data.

#### B.1.0.2 SDFG Queues

Tasks communicate via *queues*. Queues are circular data buffers stored in main memory. A queue is composed by several slots. The slot size is 16 Byte, which is the minimum DMA transfer size. Moreover, slots are properly aligned to memory boundaries to allow for efficient DMA transfers. During application execution, SDFG tasks store and read tokens from the queues. In the queues, tokens are made by several slots. The RTS handles tokens as atomic data elements, without information about the type of the transferred data or the layout of the token content. Only tasks are aware of the actual data structure of tokens (i.e. type, size and number of program variables). Queues have also a status, which can be:

- Empty: the queue has enough free space for producer tokens.
- Full: the queue has not enough free space for producer tokens.
- Buffering: a DMA memory transfer is taking place.

#### B.1.0.3 Executors

Our runtime facilitates Cell programming by providing a simple interface to using SPEs. The *executors* are wrappers around the executing units of the architecture, i.e. SPEs. They are responsible for managing the efficient execution of a task, abstracting all the complexity due to hardware low level programming. For example, executors handle data transfers and synchronization. When an executor is available for execution, it queries the resource

manager to receive a new task to be executed. If any task is available and ready, the allocation process begins and the task is executed to the target executing unit.

The implementation of the executor is decomposed in two elements: the real executing part, which is a thread running on the wrapped SPE (SPE-side), and the PPE part (PPE-side) which is responsible for orchestrating the communication with the SPEs. The SPE thread and the PPE-side of the executor communicate via mailbox. The two components of the executor exchange a small packet that describes a command and a parameter.

When a task is going to be activated on an executor, the PPE-side programs the DMA of the given SPE to transfer the input data required by the task from queues (which are stored in main memory) to the LS. The transfers of adjacent blocks are collapsed in atomic DMA transfer (if applicable based on token alignment) to take advantage of the high bus width available. When the DMA transfer is completed, a message with the execution command and the identifier of the task to be executed is sent to the SPE-side. The identifier is related to the functional code of the task. To implement a single infrastructure for the execution of different functional codes, we require that each task function has a common signature which can be referenced by a function pointer:

```
unsigned int * task_function_ptr (void *buffer).
```

The task functions have an integer return value (the exit code) and a pointer to the buffer of the local data of the task (both input and output).

When the SPE-side thread completes the execution of a task, it sends back to the PPE-side a complete message with the exit code of the function. Then it waits for the next command on the mailbox. The RTS is responsible for locating the input and output area by means of the information given on the task input/output layout. Thanks to the multiple-buffering technique and the ability of mailboxes to queue multiple messages, the SPE should theoretically execute tasks non-stop. In fact, while a SPE-side is executing a task, the PPE-side can transfer via DMA the input data of the next task

and then send the corresponding execution message, so when the current task ends the SPE-side can immediately run the next one. In the future, the executor can be extended with the introduction of other commands to obtain different behaviors.

#### B.1.0.4 Resource Manager

The resource manager is in charge of deciding when a task should be executed and of monitoring the execution of the tasks in the executors. Upon a firing event the resource manager offloads tasks to target executors, i.e. SPEs. From a functional point of view, the resource manager relieves the programmer from the burden of coping with the following difficult tasks:

- Executors management. It performs all executors management tasks, including initialization, memory management, scheduling, and exception handling.
- Load balancing. It dynamically assigns jobs to the executors, based on their availability. When all executors are busy, it internally queues new tasks. When an executor completes a job, the runtime sends a task from this queue to the executor.

Tasks, queues and executors contain all the logic to execute the correct operation according to the current internal status. In this way, the resource manager is only responsible for maintaining the collection of executors and the list of the tasks to run on each executor (according to the given scheduling graph). The PPE handles the resource manager, and the status of tasks and queues. At program startup, a given number of executors are spawned, each on a different SPE. All the control logic is handled by the PPE, minimizing the duties of the SPEs, only responsible to executing the task functions. This strategy allows for a very small code footprint on the SPE, so the main part of the tiny Local Store can be reserved to the task functions and the data.

The runtime implements several enhancements targeted to the main specific architectural features of the Cell processor, like multi-buffering and effi-

cient communication. The resource manager follows indeed some guidelines in order to efficiently handle tasks:

- A task can be scheduled if its predecessor tasks in the graph have finished their execution
- To reduce the overhead of the DMA, resource managers pre-load input buffers before scheduling a new task (i.e. double buffering and computation/communication overlapping).
- Locality of data is exploited by keeping task outputs in the SPU local memory and scheduling tasks that reuse this data to the same SPU.

The runtime tracks dependencies and schedules tasks when all required inputs are available and output buffers have free space. The runtime DMAs input data to SPEs local memory and sends results back to the components running on the PPE. Efficiently implementing these communication and synchronization mechanisms requires in-depth knowledge of the Cell BE architecture.

The local storage (LS) of SPEs is statically divided according to the buffering parallelism rank and each portion is used consecutively. The two parts of an executor always work together so they are synchronized about which portion needs to be used by a given task, without the need to explicitly exchange such information. If a task is fired on a SPE, it will find its input data and will produce its output data in temporary buffers hosted in the Local Store of the SPE. The RTS will automatically pre/off-load data in/from temporary buffers from/to the right queues.

### B.1.1 The `gdl2c` translator

The `gdl2c` *translator* is a tool which analyzes the *GDL+* produced by the *solver* and generates the code which orchestrates the execution of the target SDFG. The tool parses the graph and analyzes the task function as well as the configuration of the input and output arcs of each node to determine the different task types, that is the various node topologies. It emits the C

code responsible for the initialization of the data queues, the tasks (with the determined task type and the references to the input and output queues) and the resource manager, assigning the tasks to the task list of the specific executing resource they are assigned to. It also emits the executor code responsible for mapping each task type to a different configuration of number of inputs/outputs and number of tokens from each one. This code is required by the PPE-side of the executor to transfer the appropriate data and in the correct order. For the SPE side it is just required a small piece of code that maps each task-type with the pointer of corresponding task functions.

## B.2 Programming Model

This section describes our *Programming Model*, namely the language constructs we provide to express data flow semantics at code level and the *front-end compiler*. Our approach borrows from the coding style of the well-known shared memory programming model *OpenMP*. *OpenMP* provides a set of compiler directives that allow to outline parallelism and work sharing within a standard (sequential) C[1] program. A compiler is in charge of transforming the annotations into code which spawns parallel threads at runtime and manages data sharing among them. The main benefits of this approach reside in an increased ease of use and productivity, since a programmer does not have to learn new language constructs or a brand-new programming language. We believe that our approach can be very beneficial for the purpose of modeling dataflow computation without the need for explicit (i.e. manual) outlining of tasks and data exchange management. The only burden is that of annotating an application – written in a familiar programming style – with directives that instruct the compiler on how to transform the program.

To achieve this goal, we define a set of directives which provide the needed abstractions to model a SDFG (i.e. actors, incoming/outgoing arcs, etc.). The role of our *front-end compiler* is twofold. First, the directives inserted by the programmer are processed so as to generate C code which will be

[1] or C++, or Fortran

compiled for the SPEs. Second, the structure of the SDFG described through the custom directives is extracted into a *GDL* representation for the *solver* to process.

### B.2.1 Programming Interface

Let us consider the example SDF shown in Figure B.1. It consists of four *actors*, known as *kernels* in our programming model. Arcs are labeled with the name of the variable holding data exchanged and production/consumption rates are specified at both ends of an edge. The example SDF can be modeled through custom directives as shown in the following listing.

```
void foo()
{
  /* Shared variables */
  double A[N], B[N], C[N], D[N];
  int i;

  /* N0 - input*/
#pragma sdf kernel arcout(A, 16) arcout(B,16) input
  for (i=0; i<N; i++)
  {
    A[i] = i*2;
    B[i] = sqrt(i);
  }

  /* N1 */
#pragma sdf kernel arcin(B, 32) arcout(C, 16)
  {
    for (i=0; i<N-1; i++)
      C[i] = B[i] + B[i+1];
  }

  /* N2 */
#pragma sdf kernel arcin(A, 16) arcin(C, 16) arcout(D,16)
  {
    for (i=0; i<N; i+=2)
    {
      D[i] = A[i] * C[i];
      D[i+1] = A[i] / C[i];
    }
  }

  /* N3 - output */
#pragma sdf kernel arcin(D, 1) output
  {
    for (i=0; i<N; i++)
      printf("D[%d] has value %f\n",i, D[i]);
  }}
```

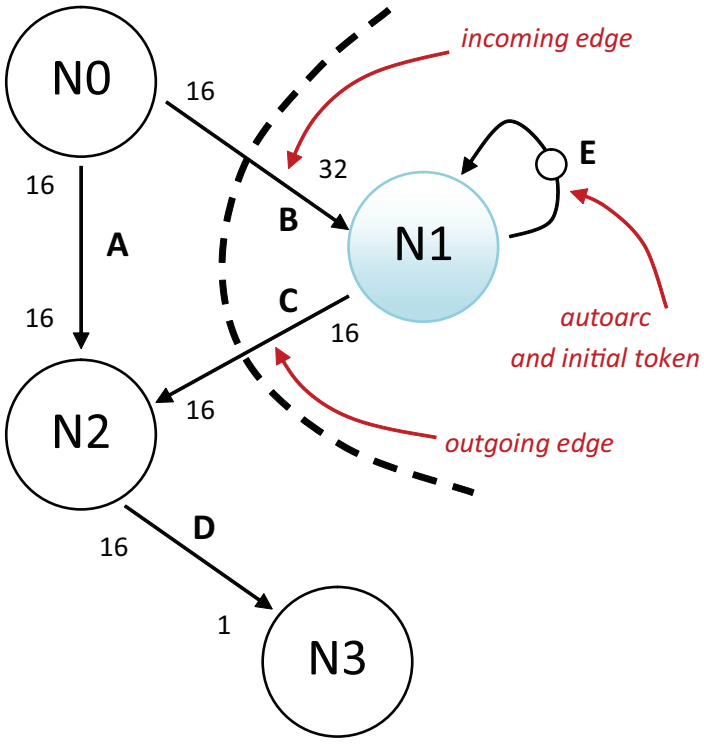


Figure B.1: Example SDF

An *actor* is easily identified by enclosing a portion of code within the `#pragma sdf kernel` directive. Data flow is modeled through specific *clauses* associated to a `kernel` directive. Focusing for instance on N1 (the second *kernel*, highlighted in Fig. B.1) we see that it has an incoming arc, associated to array `B`, and one outgoing arc, associated to `C`. N1 consumes 32 tokens and produces 16 tokens at every *firing*. We can capture this information by attaching a numerical parameter to the `arcin` and `arcout` clauses. *Kernel* N1 also has an *autoarc*. *Autoarcs* are modeled with a couple of matching (associated to the same variable) *arcin*/*arcout* clauses. *Input* (N0) and *output* (N3) nodes of a SDFG do not consume (or produce, respectively) any tokens, and often represent I/O tasks whose execution may be better suited for a computational unit other than the SPE. Thus, the solver (or the programmer) can decide to pin these tasks onto the PPE. To specify this behavior such nodes in the graph can be annotated with the *input* and *output* clauses.

### B.2.2 C Code Generation

Here we show how the source code is modified to be compiled for the hardware platform. The code enclosed within a `kernel` directive is outlined into a separate function with an unique name. This is necessary since PPE and SPEs have different views of the application being executed. More specifically, the PPE acts as a controller. It is in charge of loading on the local storage (LS) of the designated SPE the code of the task as well as the content of the data buffers. The PPE loads information about what pieces of code and data to transfer from the *GDL+*, but it simply initiates bulk transfers of known size and start address. Using *function outlining* with an unique name for the generated `C` code and *GDL* allows the runtime library executing on the PPE to correctly reference the desired ELF segment.

Looking back at the example code and still focusing on N1, we see how that *kernel* shares arrays B and C with other *kernels*. The outlined function – which will run on some SPEs – contains memory references (to variables B and C) that are no longer referred to a typed object in the program. Thus, a mean of correctly accessing these data must be provided.

We adopt a sort of *marshalling* technique, where referenced variables are grouped into a compiler-generated aggregate data type (i.e. a C-like `struct`). Each access to these variables is the outlined code is then replaced with an inspection of the corresponding field within the `struct` (see listing below).

```
/* Data marshalling */
typedef struct
{
  double B[N];
  double C[N];
} kernel_data_1;

/* Outlined kernel function */
unsigned int * kernel_fn_1 (void * input)
{
  struct kernel_data_1 *mdata =
    (struct kernel_data_1 *) input;

  /* Kernel code */
  for (i=0; i<N-1; i++)
    (mdata->C)[i] = (mdata->B)[i] + (mdata->B)[i+1];
}
```

The runtime library is in charge of efficiently and transparently moving data through the system and making it available in the LS of the target SPE before execution. A cast operation to the aggregate data type is automatically inserted to correctly handle data. Program variable marshalling only works correctly if we ensure that the data transfer preserves the order in which data items appear in the `struct`. To achieve this goal we augment the arc description in the *GDL* with a property which establishes the correct order for data transfers.

### B.2.3 GDL Generation

Within a *GDL* file (which is based on *XML*) a tag *node* is created for each `kernel` directive and starting from the *arcin*/*arcout* clauses we build the arcs section. Each node descriptor/tag has an associated property representing the pointer to the kernel code as the outlined function name. Each arc descriptor keeps trace of its startpoint and endpoint (as an index referencing

the nodes section) as well as input and output rates. Here we start collecting some "low-level" informations which will be directly passed to the backend support. For instance, we assign an order to the incoming/outgoing arcs to a node so to correctly and efficiently handle the data marshalling/unmarshalling into tokens.

# List of Figures

# List of Tables

# References


[1] T. Aipperspach, D. Boerstler, M. Bolliger, R. Chaudhry, D. Cox, P. Harvey, P. Harvey, H. Hofstee, C. Johns, J. Kahle, A. Kameyama, J. Keaty, Y. Masubuchi, M. Pham, J. Pille, S. Posluszny, M. Riley, D. Stasiak, M. Suzuoki, O. Takahashi, J. Warnock, S. Weitzel, D. Wendel, and K. Yazawa. Overview of the architecture, circuit design, and physical implementation of a first-generation cell processor. *IEEE Journal of Solid-State Circuits*, 41(1):179–196, January 2006.

[2] C. Artigues, S. Demassey, and E. Néron. *Resource-constrained Project Scheduling*. Control systems, robotics and manufacturing series. Wiley, 2010.

[3] M. Ayala and C. Artigues. On integer linear programming formulations for the resource-constrained modulo scheduling problem, 2010.

[4] M. Ayala, A. Benabid, C. Artigues, and C. Hanen. The resource-constrained modulo scheduling problem: an experimental study. *Computational Optimization and Applications*, (0926-6003):1–29, July 2012.

[5] D. Baatar, N. Boland, S. Brand, and P. J. Stuckey. Minimum cardinality matrix decomposition into consecutive-ones matrices: CP and IP approaches. In *Proc. of CPAIOR '07*, pages 1–15, 2007.

[6] P. Baptiste, C. Le Pape, and W. Nuijten. *Constrains-based scheduling: applying Constraint Programming to Scheduling*. Springer, 2001.

[7] P. Baptiste, C. Le Pape, and W. Nuijten. *Constraint-Based Scheduling*. Springer, 2001.

[8] R. Bartak and M. Salido. Constraint satisfaction for planning and scheduling problems. *Constraints*, 16:223–227, 2011. 10.1007/s10601-011-9109-4.

[9] S. Bell, B. Edwards, J. Amann, R. Conlin, K. Joyce, V. Leung, J. MacKay, M. Reif, B. Liewei, J. Brown, M. Mattina, M. Chyi-Chang, D. Ramey, C.and Wentzlaff, W. Anderson, E. Berger, N. Fairbanks, D. Khan, F. Montenegro, J. Stickney, and J. Zook. Tile64 processor: A 64-core soc with mesh interconnect. In *International Solid-State Circuits Conference*, pages 88 – 598, 2008.

[10] P. Bellens, J. Perez, R. Badia, and J. Labarta. Cellss: a programming model for the cell be architecture. In *SC 2006 Conference, Proceedings of the ACM/IEEE*, pages 5 –5, 2006.

[11] A. Benabid and C. Hanen. Worst case analysis of decomposed software pipelining for cyclic unitary RCPSP with precedence delays. *Journal of Scheduling*, 14(5):511–522, Jan. 2011.

[12] L. Benini, E. Flamand, D. Fuin, and D. Melpignano. P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proc of. DATE 2012*, pages 983 –987, March 2012.

[13] R. Bent and P. V. Hentenryck. Randomized Adaptive Spatial Decoupling For Large-Scale Vehicle Routing with Time Windows. In *Proc. of AAAI '07*, pages 173–178, 2007.

[14] S. S. Bhattacharyya and S. Sriram. *Embedded Multiprocessors - Scheduling and Synchronization (Signal Processing and Communications) (2nd Edition)*. CRC Press, 2009.

[15] T. Bijlsma, M. Bekooij, P. Jansen, and G. Smit. Communication between nested loop programs via circular buffers in an embedded multiprocessor system. In *Proceedings of the 11th international workshop on*

*Software Compilers for embedded systems*, SCOPES '08, pages 33–42, 2008.

[16] 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.

[17] G. Blake, R. Dreslinski, and T. Mudge. A survey of multicore processors. *IEEE Signal Processing Magazine*, 26(6):26–37, Nov. 2009.

[18] A. Bonfietti, L. Benini, M. Lombardi, and M. Milano. An Efficient and Complete Approach for Throughput-Maximal SDF Allocation and Scheduling on Multi-Core Platforms. In *Proc. of DATE*, pages 897—-902, 2010.

[19] A. Bonfietti, M. Lombardi, L. Benini, and M. Milano. A Constraint Based Approach to Cyclic RCPSP. In *Proc of CP*, pages 130–144, 2011.

[20] A. Bonfietti, M. Lombardi, L. Benini, and M. Milano. Global cyclic cumulative constraint. In *CPAIOR*, pages 81–96, 2012.

[21] A. Bonfietti, M. Lombardi, M. Milano, and L. Benini. Throughput Constraint for Synchronous Data Flow Graphs. In *Proc. of CPAIOR '09*, pages 26–40, 2009.

[22] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for gpus: stream computing on graphics hardware. In *SIGGRAPH '04: ACM SIGGRAPH 2004 Papers*, pages 777–786, New York, NY, USA, 2004. ACM Press.

[23] P.-y. Calland, A. Darte, and I. C. Society. Circuit Retiming Applied to Decomposed Software Pipelining. *IEEE Transactions on Parallel and Distributed Systems*, 9(1):24–35, 1998.

[24] W. Che and K. S. Chatha. Compilation of stream programs for multicore processors that incorporate scratchpad memories. In *, Design, Automation and Test in Europe Conference (DATE)*, DATE '10, 2010.

[25] W. Che and K. S. Chatha. Compilation of stream programs onto scratchpad memory based embedded multicore processors through retiming. In *Design Automation Conference (DAC)*, DAC '11, 2011.

[26] H. Chen, C. 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.

[27] Y. Choi, Y. Lin, N. Chong, S. Mahlke, and T. Mudge. Stream compilation for real-time embedded multicore systems. In *Proceedings of the 7th annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '09, pages 210–220, Washington, DC, USA, 2009. IEEE Computer Society.

[28] J. M. Codina, J. Llosa, and A. González. A comparative study of modulo scheduling techniques. In *Proc. of ICS '02*, pages 97—-106, 2002.

[29] P. Cooper, U. Dolinsky, A. F. Donaldson, A. Richards, C. Riley, and G. Russell. Offload - automating code migration to heterogeneous multicore systems. In *HiPEAC*, pages 337–352, 2010.

[30] A. Darte and G. Huard. Loop Shifting for loop compaction. *International Journal of Parallel Programming*, 28:28—-5, 2000.

[31] A. Dasdan. Experimental analysis of the fastest optimum cycle ratio and mean algorithms. *ACM Transactions on Design Automation of Electronic*, 9(4):385–418, Oct. 2004.

[32] A. Dasdan and R. Gupta. Faster maximum and minimum mean cycle algorithms for system-performance analysis. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(10):889–899, 1998.

[33] A. G. David E. Culler, Jaswinder Pal Singh. *Parallel Computer Architecture: A Hardware/Software Approach*. Gulf Professional Publishing, 1999.

[34] R. Dechter. *Constraint Processing*. Morgan Kaufmann, 2003.

[35] 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.

[36] B. Dupont de Dinechin. From Machine Scheduling to VLIW Instruction Scheduling, 2004.

[37] A. Eichenberger and E. Davidson. Efficient formulation for optimal modulo schedulers. *ACM SIGPLAN Notices*, 32(5):194–205, 1997.

[38] K. Fatahalian, D. R. Horn, T. J. Knight, L. Leem, M. Houston, J. Y. Park, M. Erez, M. Ren, A. Aiken, W. J. Dally, and P. Hanrahan. Sequoia: programming the memory hierarchy. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 83, New York, NY, USA, 2006. ACM.

[39] A. Franceschelli, P. Burgio, G. Tagliavini, A. Marongiu, M. Ruggiero, M. Lombardi, A. Bonfietti, M. Milano, and L. Benini. Mpopt-cell: a high-performance data-flow programming environment for the cell be processor. In *Conf. Computing Frontiers*, page 11, 2011.

[40] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[41] F. Gasperoni and U. Schwiegelshohn. Efficient Algorithms for Cyclic Scheduling. Technical report, 1991.

[42] L. Georgiadis, A. V. Goldberg, R. Tarjan, and R. F. Werneck. An experimental study of minimum mean cycle algorithms. In *Proceedings of the 11th Workshop on Algorithm Engineering and Experiments (ALENEX)*, page 13, 2009.

[43] A. Ghamarian, M. Geilen, T. Basten, B. Theelen, M. Mousavi, and S. Stuijk. Liveness and boundedness of synchronous data flow graphs.

In *IN FORMAL METHODS IN COMPUTER AIDED DESIGN, FMCAD 06, PROCEEDINGS. IEEE, 2006*, pages 68–75, 2006.

[44] A. H. Ghamarian, M. Geilen, S. Stuijk, T. Basten, B. D. Theelen, M. R. Mousavi, A. J. M. Moonen, and M. J. Bekooij. Throughput Analysis of Synchronous Data Flow Graphs. In *Proc. of ACSD '06*, pages 25–36, 2006.

[45] M. Hagog and A. Zaks. Swing modulo scheduling for gcc. Technical report, 2004.

[46] W. Haid, K. Huang, I. Bacivarov, and L. Thiele. Multiprocessor SoC software design flows. *IEEE Signal Processing Magazine*, 26(6):64–71, Nov. 2009.

[47] W. Haid, L. Schor, K. Huang, I. Bacivarov, and L. Thiele. Efficient execution of kahn process networks on multi-processor systems using protothreads and windowed fifos. In *Proc. IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 35–44, Grenoble, France, 2009. IEEE.

[48] C. Hanen. Study of a NP-hard cyclic scheduling problem: The recurrent job-shop. *European Journal of Operational Research*, 72(1):82–101, 1994.

[49] C. Hanen and A. Munier. A study of the cyclic scheduling problem on parallel processors. *Discrete Applied Mathematics*, 57(2-3):167–192, Feb. 1995.

[50] H. P. Hofstee. Power efficient processor architecture and the cell processor. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, pages 258–262, Washington, DC, USA, 2005. IEEE Computer Society.

[51] A. H. Hormati, Y. Choi, M. Kudlur, R. Rabbah, T. Mudge, and S. Mahlke. Flextream: Adaptive compilation of streaming applications

for heterogeneous architectures. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*, pages 214–223, 2009.

[52] R. R. a. Huff. Lifetime-sensitive modulo scheduling. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, volume 28, pages 258–267. ACM, June 1993.

[53] K. Ito and K. K. K. Parhi. Determining the minimum iteration period of an algorithm. *The Journal of VLSI Signal Processing*, 244(3):229–244, 1995.

[54] A. A. Jerraya, O. Franza, M. Levy, M. Nakaya, P. G. Paulin, U. Ramacher, D. Talla, and W. H. Wolf. Roundtable: Envisioning the future for roundtable: Envisioning the future for multiprocessor soc. *Design Test of Computers, IEEE*, 24(2):174–183, March-April 2007.

[55] U. J. Kapasi, S. Rixner, W. J. Dally, B. Khailany, J. H. Ahn, P. Mattson, and J. D. Owens. Programmable stream processors. *Computer*, 36(8):54–62, 2003.

[56] L. Karam, I. Alkamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans. Trends in multicore DSP platforms. *IEEE Signal Processing Magazine*, 26(6):38–49, Nov. 2009.

[57] M. Karczmarek, W. Thies, and S. Amarasinghe. Phased scheduling of stream programs. In *Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, LCTES '03, pages 103–112. ACM, 2003.

[58] R. Karp. A characterization of the minimum cycle mean in a digraph. *Discrete Mathematics*, 23(3):309–311, 1978.

[59] R. M. Karp and R. E. Miller. Properties of a model for parallel computations: Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):pp. 1390–1411, 1966.

[60] B. Khailany, T. Williams, J. Lin, E. Long, M. Rygh, D. Tovey, and W. Dally. A programmable 512 gops stream processor for signal, image, and video processing. *Solid-State Circuits, IEEE Journal of*, 43(1):202 –213, jan. 2008.

[61] M. Kudlur and S. Mahlke. Orchestrating the execution of stream programs on multicore platforms. In *Proc. of PLDI*, volume 43, pages 114–124, May 2008.

[62] C. Le Pape, P. Couronné, D. Vergamini, and V. Gosselin. Time-versus-capacity compromises in project scheduling. *AISB QUARTERLY*, page 19, 1995.

[63] E. A. Lee, S. S. Bhattacharyya, and P. K. Murthy. *Software synthesis from data flow graphs.* Kluwer Academic Press, 1996.

[64] E. A. Lee and D. G. Messerschmitt. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36(1):24–35, Jan. 1987.

[65] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow. *Proceedings of the IEEE*, 75(9):1235–1245, 1987.

[66] J. Llosa, A. Gonzalez, E. Ayguade, and M. Valero. Swing modulo scheduling: A lifetime-sensitive approach. In *pact*, pages 80–87. Published by the IEEE Computer Society, 1996.

[67] J. Llosa, A. Gonzalez, E. Ayguade, M. Valero, and J. Eckhardt. Lifetime-sensitive modulo scheduling in a production environment. *IEEE Trans. on Comps.*, 50(3):234 – 249, 2001.

[68] M. Lombardi, A. Bonfietti, M. Milano, and L. Benini. Precedence Constraint Posting for Cyclic Scheduling Problems. In *CPAIOR*, pages 137–153, 2011.

[69] M. Lombardi, M. Milano, and L. Benini. Robust scheduling of task graphs under execution time uncertainty. *IEEE Trans. Computers*, 62(1):98–111, 2013.

[70] M. D. McCool and B. D'Amora. Programming using rapidmind on the cell be. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 222, New York, NY, USA, 2006. ACM.

[71] O. Moreira, J.-D. Mol, M. J. Bekooij, and J. van Meerbergen. Multiprocessor Resource Allocation for Hard-Real-Time Streaming with a Dynamic Job-Mix. In *11th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'05)*, pages 332–341. IEEE, 2005.

[72] O. Moreira, P. Poplavko, M. Pastrnak, B. Mesman, J. Mol, S. Stuijk, V. Gheoghita, M. Bekooij, R. Hoes, and J. van Meerbergen. *Dynamic and robust streaming in and between connected consumer-electronic devices*. Springer, 2005.

[73] O. Moreira, F. Valente, and M. Bekooij. Scheduling multiple independent hard-real-time jobs on a heterogeneous multiprocessor. In *Proceedings of the 7th ACM & IEEE international conference on Embedded software*, EMSOFT '07, pages 57–66, New York, NY, USA, 2007. ACM.

[74] M. Muller. Embedded processing at the heart of life and style. pages 32–37, 2008.

[75] H. Munk, E. Ayguadé, C. Bastoul, P. Carpenter, Z. Chamski, A. Cohen, M. Cornero, P. Dumont, M. Duranton, M. Fellahi, R. Ferrer, R. Ladelsky, M. Lindwer, X. Martorell, C. Miranda, D. Nuzman, A. Ornstein, A. Pop, S. Pop, L.-N. Pouchet, A. Ramírez, D. Ródenas, E. Rohou, I. Rosen, U. Shvadron, K. Trifunovic, and A. Zaks. Acotes project: Advanced compiler technologies for embedded streaming. *International Journal of Parallel Programming*, pages 1–54, 2010. 10.1007/s10766-010-0132-7.

[76] M. Nijhuis, H. Bos, H. E. Bal, and C. Augonnet. Mapping and synchronizing streaming applications on cell processors. In *Proceedings of the*

*4th International Conference on High Performance Embedded Architectures and Compilers*, HiPEAC '09, pages 216–230, Berlin, Heidelberg, 2009. Springer-Verlag.

[77] K. O'Brien, K. O'Brien, Z. Sura, T. Chen, and T. Zhang. Supporting OpenMP on Cell. In *IWOMP '07: Proceedings of the 3rd international workshop on OpenMP*, pages 65–76, Berlin, Heidelberg, 2008. Springer-Verlag.

[78] M. Ohara, H. Inoue, Y. Sohda, H. Komatsu, and T. Nakatani. Mpi microtask for programming the cell broadband engine processor. *Online Game Technology*, 45(1):85, 03 2006.

[79] S. Olivier, J. Prins, J. Derby, and K. Vu. Porting the gromacs molecular dynamics code to the cell processor. In *IPDPS*, pages 1–8. IEEE, 2007.

[80] F. A. Omara and M. M. Arafa. Genetic algorithms for task scheduling problem. *J. Parallel Distrib. Comput.*, 70:13–22, January 2010.

[81] C. Ostler, K. S. Chatha, V. Ramamurthi, and K. Srinivasan. ILP and heuristic techniques for system-level design on network processor architectures. *ACM Transactions on Design Automation of Electronic Systems*, 12(4):48–es, Sept. 2007.

[82] M. Paganini. Nomadik: A mobile multimedia application processor platform. In *IEEE Asia and South Pacific Design Automation Conference*, pages 749–750, 2007.

[83] K. K. Parhi and D. G. Messerschmitt. Rate-optimal fully-static multiprocessor scheduling of data-flow signal processing programs. *IEEE International Symposium on Circuits and Systems*, (217):1923–1928, 1989.

[84] 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.

[85] G. Pesant, M. Gendreau, J.-y. Potvin, and J.-m. Rousseau. An Exact Constraint Logic Programming Algorithm for the Traveling Salesman Problem with Time Windows. *Transportation Science*, 32:12—-29, 1996.

[86] R. B. Rau. Iterative modulo scheduling: An algorithm for software pipelining loops. In *Proceedings of the 27th annual international symposium on Microarchitecture*, pages 63–74. ACM, 1994.

[87] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. In *Proc. of AAAI '94*, pages 362–367, 1994.

[88] F. Rossi, P. Van Beek, and T. Walsh. *Handbook of Constraint Programming*. Elsevier, 2006.

[89] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini. A Fast and Accurate Technique for Mapping Parallel Applications on Stream-Oriented MPSoC Platforms with Communication Awareness. *International Journal of Parallel Programming*, 36(1):3–36, 2008.

[90] M. Ruggiero, A. Guerri, D. Bertozzi, M. Milano, and L. Benini. A fast and accurate technique for mapping parallel applications on stream-oriented mpsoc platforms with communication awareness. *International Journal of Parallel Programming*, 36:3–36, 2008. 10.1007/s10766-007-0032-7.

[91] M. Ruggiero, A. Guerri, D. Bertozzi, F. Poletti, and M. Milano. Communication-aware allocation and scheduling framework for stream-oriented multi-processor systems-on-chip. In *DATE '06: Proceedings of the conference on Design, automation and test in Europe*, pages 3–8, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[92] M. Ruggiero, M. Lombardi, M. Milano, and L. Benini. Cellflow: A parallel application development environment with run-time support for

the cell be processor. In *DSD '08: Proceedings of the 2008 11th EUROMICRO Conference on Digital System Design Architectures, Methods and Tools*, pages 645–650, Washington, DC, USA, 2008. IEEE Computer Society.

[93] J. Sermulins, W. Thies, R. Rabbah, and S. Amarasinghe. Cache aware optimization of stream programs. In *Proceedings of the 2005 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, LCTES '05, pages 115–126. ACM, 2005.

[94] S. Sriram and E. A. Lee. Determining the order of processor transactions in statically scheduled multiprocessors. *The Journal of VLSI Signal Processing*, 15(3):207–220, 1997.

[95] D. Stasiak, R. Chaudhry, D. Cox, S. Posluszny, J. Warnock, S. Weitzel, D. Wendel, and M. Wang. Cell processor low-power design methodology. *IEEE Micro*, 25:71–78, 2005.

[96] R. Stephens. A survey of stream processing, 1995.

[97] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal. Multiprocessor Resource Allocation for Throughput-Constrained Synchronous Dataflow Graphs. In *Proc. of DAC '07*, pages 777–782. Ieee, June 2007.

[98] S. Stuijk, M. Geilen, and T. Basten. SDFˆ3: SDF For Free. In *Proc. of ACSD '06*, pages 276–278. Ieee, 2006.

[99] M. B. Taylor, J. Kim, J. Miller, D. Wentzlaff, F. Ghodrat, B. Greenwald, H. Hoffman, P. Johnson, J.-W. Lee, W. Lee, A. Ma, A. Saraf, M. Seneski, N. Shnidman, V. Strumpen, M. Frank, S. Amarasinghe, and A. Agarwal. The raw microprocessor: A computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, 2002.

[100] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *Proceedings of the 11th Interna-*

*tional Conference on Compiler Construction*, CC '02, pages 179–196, London, UK, 2002. Springer-Verlag.

[101] W. Thies, M. Karczmarek, and S. P. Amarasinghe. Streamit: A language for streaming applications. In *CC '02: Proceedings of the 11th International Conference on Compiler Construction*, pages 179–196, London, UK, 2002. Springer-Verlag.

[102] B. Ucar, C. Aykanat, K. Kaya, and M. Ikinci. Task assignment in heterogeneous computing systems. *J. Parallel Distrib. Comput.*, 66:32–46, January 2006.

[103] M. H. Wiggers, M. J. G. Bekooij, and G. J. M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, DAC '07, pages 658–663, 2007.

[104] S. Williams, J. Shalf, L. Oliker, S. Kamil, P. Husbands, and K. Yelick. Scientific computing kernels on the cell processor. *International Journal of Parallel Programming*, 35(3):263–298, June 2007.