Efficient evaluation of mappings of dataflow applications onto distributed memory architectures

Youen Lesparre

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Youen Lesparre

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Évaluation de l'affectation des tâches sur une architecture à mémoire distribuée pour des modèles flot de données.

soutenue le 2 mars 2017

devant le jury composé de :

Mme. Alix Munier-Kordon	Directeur de thèse
M. Jean-Marc Delosme	Directeur de thèse
M. Jean-François Nézan	Rapporteur
M. Renault Sirdey	Rapporteur
M. Albert Cohen	Examinateur
M. Benoît Dupont de Dinechin	Examinateur
Mme. Safia Kedad-Sidhoum	Examinateur

Contents

Remerciements

Je remercie chaleureusement Jean-Marc Delosme et Alix Munier-Kordon qui durant 3 ans, et même un peu plus, m'ont encadré dans mes recherches. J'ai grandement apprécié leurs qualités tant humaines que pedagogiques.

Je remercie aussi les membres du jury pour l'intérêt qu'ils ont manifesté pour mes travaux de recherche.

Merci à Bruno Bodin et Olivier Marchetti pour leur aide durant ma thèse et Roselyne Avot qui m'a initié à de nouvelles méthodes d'enseignement.

Merci à mes "co-bureau" : Andi Dresde et Afef Salhi ; et aussi Éric Lao, Cédric Klikpo et Jad Khatib. Merci aussi à Éric Lao et Olivier Tsiakaka qui ont élargi mes horizons culturels.

Merci à Jean-Paul Chaput et Shahin Mahmoodian qui m'ont facilité la vie au sein du LIP6.

Enfin, merci à ma sœur Nolwenn dont l'exemple m'a inspiré pour m'orienter vers la recherche. Merci à mes parents Daniel et Anne-Françoise qui m'ont soutenu pendant toute la période de préparation de ma thèse.

Abstract

With the increasing use of smart-phones, connected objects or automated vehicles, embedded systems have become ubiquitous in our living environment. These systems are often highly constrained in terms of power consumption and size. They are more and more implemented with multi- or many-core processor arrays that allow, through massive parallelism, rapid design to meet stringent real-time constraints while operating at relatively low frequency, with reduced power consumption.

Running an application on a processor array requires dispatching its tasks on the processors in order to meet capacity and performance constraints. This mapping problem is known to be NP-complete.

The contributions of this thesis are threefold:

First we extend the notions of consistency, precedence constraint and useful tokens to the Phased Computation Graph model. Two equivalent sufficient conditions of liveness, used for dataflow graph generation and mapping evaluation, are also extended to this model.

Second, we present a random dataflow graph generator able to generate Synchonous Dataflow Graphs, Cyclo-Static Dataflow Graphs and Phased Computation Graphs. The Generator, called `Turbine`, is able to generate live dataflow graphs of up to 10,000 tasks in less than 30 seconds. It is illustrated through several experiments and compared to SDF3 and PREESM. It is available on github `https://github.com/bbodin/turbine`.

Third and most important, we propose a new method of evaluation of a mapping using the Synchonous Dataflow Graph and the Cyclo-Static Dataflow Graph models. The method evaluates efficiently the memory footprint of the communications of a dataflow graph mapped on a distributed architecture. The evaluation is declined in two versions, the first guarantees a live mapping while the second accounts for a constraint on throughput.

The evaluation method is illustrated through experiments on dataflow graphs generated with Turbine and on several real-life applications.

1 Introduction

Le domaine de l'embarqué et des systèmes sur puce est en plein essor. Avec l'avènement des smartphones, des véhicules automatiques et des objets connectés la demande en puissance de calcul avec une faible consommation énergétique ne cesse d'augmenter.

A cause des limites de fréquence atteintes ces dernières années, le nombre de cœurs au sein des architectures a été multiplié. La multiplication des cœurs pose de nouveaux problèmes lors de la conception et de l'exécution d'une application.

Le problème de mapping (distribuer une application sur un multi-cœurs) est, pour le domaine de l'embarqué, considéré comme un des problèmes les plus urgents de cette décennie [Marwedel et al., 2011]. Cette thèse propose une méthode efficace pour évaluer un mapping sur une architecture multi-cœurs à mémoire distribuée. La méthode cible des applications de flux de données modélisées par plusieurs milliers de tâches.

Les applications de flux de données font partie des plus demandeuse en ressources. Il est possible de représenter ces applications par des modèles dataflow.

La Section 2 présente les modèles dataflow étudiés durant cette thèse. La Section 3 introduit des propriétés importantes pour le modèle Synchronous Dataflow. Ces propriétés ont ensuite étés étendues aux modèles Cyclo-Static Dataflow et Phased Computation Graph. La Section 4 décrit brièvement les travaux effectués sur ces deux derniers modèles. La Section 5 présente un générateur de graphe dataflow implémenté durant cette thèse. La Section 6 explique la méthode d'évaluation d'un mapping. La Section 7 propose de résoudre un problème de mapping en utilisant la méthode d'évaluation expliquée Section 6. Enfin, la Section 8 conclut ce résumé.

2 Les modèles dataflow

Cette section présente les trois modèles dataflow utilisés dans cette thèse : le modèle Synchronous Dataflow, le modèle Cyclo-Static Dataflow et le modèle Phased Computation Graph.

2.1 Le modèle Synchronous Dataflow

Le modèle Synchronous Dataflow Graph (SDFG) introduit dans [Lee and Messerschmitt, 1987] est le premier modèle dataflow à avoir vu le jour en 1987. Il modélise les tâches d'une application et leurs communications par un graphe orienté. Les nœuds du graphe correspondent aux tâches (ou acteurs) de l'application et les arcs correspondent aux communications entre les tâches. Un arc (ou buffer) fonctionne comme une mémoire First in First out (FIFO). Chaque arc du graphe est composé de trois poids, un poids en entrée qui symbolise la production de données lorsque la

tâche en amont est exécutée, un poids en sortie qui symbolise la consommation de données de la tâche en aval, et un marquage initial qui correspond à la quantité de données présentes dans l'arc au démarrage de l'application.

Plus formellement on a un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ avec $\mathcal{T}$ l'ensemble des tâches et $\mathcal{A}$ l'ensemble des arcs. La quantité de données écrites dans l'arc $a = (t_i, t_j)$ par la tâche t_i à la fin de son exécution est définie par p_a. La quantité de donnée consommées de l'arc $a = (t_i, t_j)$ par la tâche t_j au début de son exécution définie par c_a. Le marquage initial de l'arc a est noté $M_0(a)$. Enfin, la durée d'exécution d'une tâche t_i est notée ℓ_i.

La Figure 1 représente un SDFG avec deux tâches reliées par un arc.

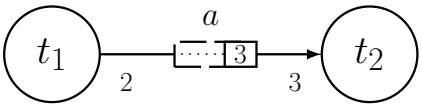


Figure 1 – Un SDFG composé de deux tâches t_1 et t_2 et d'un arc $a = (t_1, t_2)$ avec $p_a = 2$, $c_a = 3$ et $M_0(a) = 3$.

2.2 Le modèle Cyclo Static Dataflow

Le modèle Cyclo-Static Dataflow Graph (CSDFG) [Bilsen et al., 1995] est une évolution du modèle SDFG. Les consommation et production de données ne sont plus symbolisées par un entier comme pour le modèle SDFG mais par un vecteurs d'entiers dont l'exécution est cyclique. L'exécution d'une tâche est donc divisée en phases.

Plus formellement on a φ_i le nombre de phases de la tâche t_i. $t_i(k)$ désigne la $k^{ième}$ phase de t_i avec $k \in \{1, \cdots, \varphi_i\}$. La durée d'une phase est notée $\ell_i(k)$. On note $p_a(k)$ la quantité de données produite par la phase k et $c_a(k)$ la quantité de données consommées par la phase k.

Un CSDFG est illustré sur la Figure 2. La tâche t_1 est composée d'une phase alors que la tâche t_2 est composée de 2 phases.

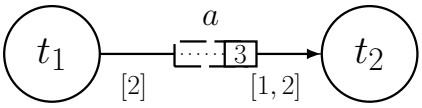


Figure 2 – Un graphe CSDFG composé d'un arc $a = (t_1, t_2)$ avec $[p_a(1)] = [2]$, $[c_a(1), c_a(2)] = [1, 2]$ et $M_0(a) = 3$. On a $\varphi_1 = 1$ et $\varphi_2 = 2$.

2.3 Le modèle Phased computation graph

Le modèle Phased Computation Graph (PCG) [Thies et al., 2002] étend le modèle CSDFG afin d'augmenter son expressivité. Le modèle reprend les caractéristiques du modèle CSDFG et y ajoute des phases initiales et des seuils. Les phase initiales ne sont exécutées qu'une seule fois au démarrage de l'application. Les seuil forcent

la présence d'une certaine quantité de données dans l'arc avant qu'une partie de celles-ci puisse être consommée.

Plus formellement on a φ_i le nombre de phases cycliques de la tâche t_i et σ_i le nombre de phases initiales. La production et la consommation d'une tâche dans un arc sont notés $p_a(k)$ et $c_a(k)$ avec $k \in \{1-\sigma_i, \cdots, 0\} \cup \{1, \cdots, \varphi_i\}$. Le seuil d'une phase de consommation est noté $\theta_a(k)$ avec $\theta_a(k) \geq c_a(k)$.

Un graphe PCG est illustré sur la Figure 3. La tâche t_2 est composée de deux phases initiales affichées entre parenthèses et de deux phases cycliques. Les seuils apparaissent à droite des poids $c_a(k)$ et sont séparés par ":" . Ils sont affichés uniquement lorsqu'ils sont strictement supérieurs à leur poids de consommation respectif.

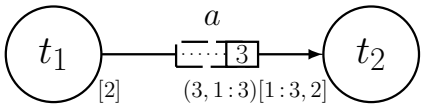


Figure 3 – Un PCG composé d'un arc $a = (t_1, t_2)$ avec $\theta_a(-1) = 3$ et $\theta_a(0) = 3$ pour les phases initiales et $\theta_a(1) = 3$ et $\theta_a(2) = 2$ pour les phases cycliques.

3 Propriétés du modèle Synchronous Dataflow

Cette section présente les principales propriétés du modèle SDFG. Ces propriétés son nécessaires pour comprendre les résultats démontrés sur le modèle PCG.

3.1 Facteur de répétition et consistance

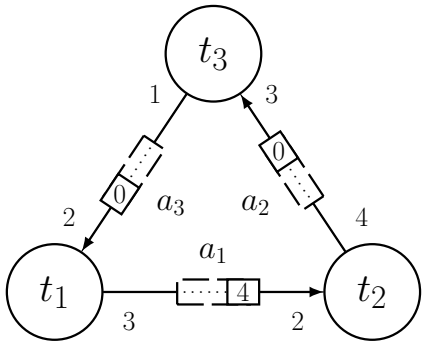
Le facteur de répétition d'une tâche t_i est noté R_i. À l'échelle du graphe on regroupe les facteurs de répétition en un vecteur noté R. Ce vecteur représente le nombre d'exécutions minimum nécessaires aux tâches avant que le graphe retrouve son marquage initial. On a donc $R_i.p_a = R_j.c_a$ avec $a = (t_i, t_j)$.

Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ un SDFG. Si $\forall a = (t_i, t_j) \in \mathcal{A}$ on a $R_i.p_a = R_j.c_a$ alors le graphe est dit consistant. La consistance est une propriété nécessaire au bon fonctionnement de l'application. Si un SDFG n'est pas consistant, lors de son exécution, on observera soit un blocages dû à un manque de marquage initial soit une augmentation constante du marquage initial.

Soit la matrice topologique Γ du SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. On définit cette matrice comme suit:

$$\Gamma_{ai} = \begin{cases} p_a & \text{if } a = (t_i, \cdot) \\ -c_a & \text{if } a = (\cdot, t_i) \\ 0 & \text{otherwise} \end{cases}$$

Le SDFG est consistant si le rang de la matrice est $|\mathcal{T}| - 1$. La Figure 4 montre un SDFG avec un vecteur de répétition $R = [2, 3, 4]$ et sa matrice topologique.



(a)

$$\Gamma = \begin{bmatrix} 3 & 0 & -2 \\ -2 & 4 & 0 \\ 0 & -3 & 1 \end{bmatrix}$$

(b)

Figure 4 – La Figure (a) représente un SDFG consistant avec un vecteur de répétition $R = [2, 3, 4]$. Figure (b) montre la matrice topologique du SDFG.

3.2 Marquage utile

Cette section présente la notion de marquage utile. Cette notion a été introduite dans [Marchetti and Munier-Kordon, 2009] pour le modèle SDFG. Soit $gcd_a = gcd(c_a, p_a)$ le plus grand diviseur commun entre p_a et c_a. Un marquage est utile s'il est multiple de gcd_a. Les auteurs démontrent qu'un marquage initial, s'il n'est pas multiple de gcd_a, peut être arrondi au gcd_a inférieur sans interférer sur les contraintes de précédence ou l'ordonnançabilité du graphe. La notion de marquage utile est formellement décrite par le Lemme 1.

Lemme 1. [Marchetti and Munier-Kordon, 2009] *Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ un SDFG.*

$$\forall a \in \mathcal{A}, M_0(a) = \lfloor M_0(a) \rfloor_{gcd_a}$$

avec $gcd_a = gcd(c_a, p_a)$ et $\lfloor x \rfloor_{gcd_a} = \lfloor \frac{x}{gcd_a} \rfloor \times gcd_a$, ne modifie pas les contraintes de précédence ou l'ordonnançabilité du graphe.

La Figure 5 représente un SDFG avec un marquage initial $M_0(a) = 7$. D'après la notion de marquage utile la production et la consommation de données est toujours multiple de gcd_a. Le marquage initial peut donc être arrondi au gcd_a inférieur sans interférer avec le fonctionnement du graphe.

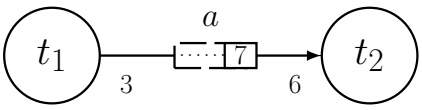


Figure 5 – Un graphe SDFG avec $M_0(a) = 7$. Si l'on applique la notion de marquage utile le marquage est arrondi au $gcd_a = 3$ inférieur, on obtient alors $M_0(a_1) = 6$.

3.3 Normalisation et notation Z_i

La normalisation est une transformation introduite dans [Marchetti and Munier-Kordon, 2009]. Cette transformation est utilisée pour simplifier les notations et les

formules lors de démonstrations sur les graphes dataflow. Un graphe ayant subi une normalisation est dit normalisé. La transformation modifie les poids de consommation et de production des arcs afin que, pour une tâche donnée, ses poids soient tous égaux.

La transformation consiste à multiplier les poids (p_a, c_a et $M_0(a)$) de chaque arc par un entier noté $N_a \in \mathbb{N} - \{0\}$. L'entier N_a est déduit par la formule $N_a = \frac{lcm_R}{R_i.p_a}$ avec $a = (t_i, \cdot)$ et $lcm_R = lcm(R_1, \cdots, R_{|\mathcal{T}|})$ le plus petit multiple commun parmi les éléments du vecteur de répétition R.

Les entiers N_a peuvent être regroupés dans un vecteur noté $N = [N_{a_1}, \cdots, N_{|\mathcal{A}|}]$ appelé le vecteur de normalisation. La transformation nécessite un graphe consistant. Elle est calculée en temps polynomiale.

Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ un SDFG. Le poids normalisé d'une tâche avec un arc sortant (réciproquement entrant) $a = (t_i, t_j)$ est $Z_i = p_a.N_a$ (réciproquement $Z_j = c_a.N_a$). La normalisation est illustrée sur la Figure 6.

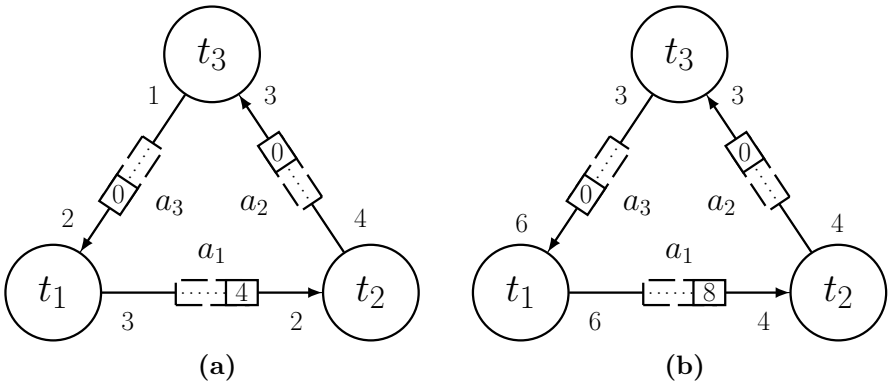


Figure 6 – En Figure (a), un SDFG avec un vecteur de répétition $R = [2, 3, 4]$, en Figure (b) le même graphe normalisé. Le vecteur de normalisation est $N = [2, 1, 3]$.

La normalisation est une transformation réversible. Un graphe dataflow peut être normalisé et dé-normalisé sans modifier son fonctionnement.

3.4 Vivacité

La vivacité est un problème difficile très étudié dans la communauté des dataflows. Cette propriété garantit que l'exécution des tâches d'un graphe dataflow ne rencontrera aucun blocage. À ce jour, la complexité du problème de vivacité est toujours inconnue pour le modèle SDFG et ses modèles dérivés.

Pour contourner cette complexité nous utilisons une condition suffisante de vivacité. Dans [Marchetti and Munier-Kordon, 2009] les auteurs proposent une méthode pour vérifier si un SDFG est vivant. Cette condition suffisante de vivacité est présentée dans le Théorème 1. Comme la condition est suffisante (mais pas nécessaire) un marquage qui ne respecte pas la condition suffisante de vivacité peut tout de même être vivant.

Théorème 1 (**SCL**). [Marchetti and Munier-Kordon, 2009] Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ un SDFG normalisé. $\mathcal{G}$ est vivant si pour chaque cycle μ dans $\mathcal{G}$ on a:

$$\sum_{a \in \mu} M_0(a) > \sum_{a \in \mu} p_a - gcd_a,$$

avec gcd_a le plus grand diviseur commun entre c_a et p_a.

La Figure 7(a) illustre un SDFG avec un marquage initial selon **SCL**. On a $\sum_{a \in \mu} M_0(a) = 8$ et $\sum_{a \in \mu} p_a - gcd_a = 7$. *A contrario*, le SDFG de la Figure 7(b) ne respecte pas la condition suffisante de vivacité puisque $\sum_{a \in \mu} M_0(a) = 20$ et $\sum_{a \in \mu} p_a - gcd_a = 21$. Le graphe est pourtant vivant puisque la séquence d'exécution $t_2t_2t_3t_3t_3t_1t_2t_3t_3t_1$ ramène le marquage à son état initial.

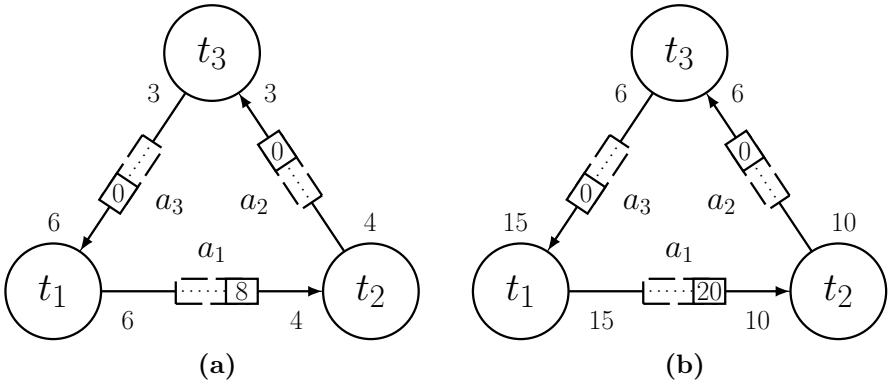


Figure 7 – La Figure (a) montre un SDFG vivant d'après la condition suffisante de vivacité du Théorème 1. La Figure (b) démontre que la condition n'est pas nécessaire.

4 Propriétés du modèle Cyclo-Static Dataflow et Extension vers le modèle Phased Computation Graph

Les propriétés introduites avec le modèle SDFG on toutes été étendues au modèle CSDFG. Dans [Bilsen et al., 1995], les auteurs étendent la consistance. La notion de marquage utile est étendue dans [Marchetti and Munier-Kordon, 2009; Stuijk et al., 2008]. La normalisation et la condition suffisante de vivacité sont étenduent dans [Benazouz et al., 2013]. Cette thèse étend ces quatre notions au modèle PCG.

5 Générateur de Dataflow : `Turbine`

Durant le première année de la thèse un générateur aléatoire de dataflow a été implémenté. Le générateur est appelé `Turbine` et est disponible à l'adresse `https://github.com/bbodin/turbine`. Son objectif est de générer, en un temps raisonnable, des graphes dataflow de grande taille (plusieurs milliers de tâches) avec un marquage initial vivant. `Turbine` est capable de générer des SDFG, CSDFG et PCG.

5.1 Génération de graphe dataflow

La génération d'un graphe dataflow est divisée en quatre étapes:

- la génération des nœuds et des arcs,
- la génération des poids de production et de consommation,
- le calcul d'un marquage initial vivant.

Lors de la première étape, l'utilisateur devra renseigner le nombre de tâches, le degré min/max des nœuds, si le graphe doit être cyclique, et si la génération d'arc ré-entrant (ayant la même tâche en entrée et en sortie) et de multi-arcs (deux arcs avec les même tâches en entrée et en sortie) est autorisé.

Pour garantir la consistance, le facteur de répétition est préalablement généré à partir d'un facteur de répétition moyen. Les poids de production/consommation sont ensuite déduits à partir des facteurs de répétition. Le graphe est généré normalisé ou non selon le choix de l'utilisateur. Pour les modèles CSDFG et PCG, les tailles des vecteurs est définie aléatoirement à partir d'un paramètre min/max.

L'étape de calcul d'un marquage initial vivant utilise la condition suffisante de vivacité décrite par le Théorème 1 et des versions étendues aux modèles CSDFG et PCG. Les conditions suffisantes de vivacité sont résolues à l'aide de la programmation linéaire.

5.2 Expérimentations

Il existe deux autres générateurs de dataflow aléatoire :SDFG3 et PREESM, ces deux générateurs sont présentés puis comparés à `Turbine`. Les expérimentations sont faites sur quatre tailles de graphe, avec respectivement, 10, 100, 1000 et 10000 tâches. Les graphes dataflow respectent les paramètres suivants : un facteur de répétition moyen $R_i = 5$ et un degré moyen de 3. Les graphes sont générés cycliques, sans arcs ré-entrants et sans multi-arcs.

Les Tableaux 1(a) et 1(b) présentent une comparaison du temps moyen de génération sur 100 instances entre `Turbine`, SDF For Free (SDF3) et PREESM pour les modèles SDFG et CSDFG. Les graphes ont été générés sur une simple machine de bureau excepté avec PREESM pour les graphes de 10000 tâches ou un serveur avec 32Go de RAM a été nécessaire. PREESM génère uniquement des graphes SDFG.

$\lvert\mathcal{T}\rvert$	`Turbine`	SDF3	PREESM
10	5ms	19ms	9ms
100	44ms	313ms	58ms
1000	592ms	1h24min	7,2s
10000	20,7s	-	1h39min*

(a)

$\lvert\mathcal{T}\rvert$	`Turbine`	SDF3
Tiny	7ms	23ms
Small	62ms	315ms
Medium	806ms	1h27min
Large	20,4s	-

(b)

Tableau 1 – Figure (a), le temps moyen de génération pour le modèle SDFG entre `Turbine`, SDF3 et PREESM. Figure (b), le temps moyen de génération pour le modèle CSDFG entre `Turbine` etSDF3.

6 Méthode polynomial d'évaluation de la consomation mémoire d'un mapping

Cette section décrit une nouvelle approche afin d'évaluer en temps polynomial le mapping d'un application. Cette nouvelle approche a pour but d'évaluer la mémoire consommée par une application sur une architecture multi-cœurs à mémoire distribuée. La méthode est appliquée sur les modèles SDFG, et est ensuite étendue au modèle CSDFG.

Le problème est décliné en deux versions, la première a pour objectif la minimisation de la mémoire, et la seconde ajoute une contrainte de débit.

6.1 Description du problème

Étant donné un mapping sur une architecture composée d'un certain nombre de clusters reliés par un NoC (Network on Chip ou réseau sur puce), l'objectif est de trouver la consommation mémoire des communications entre les tâches de l'application.

Cette nouvelle approche propose de mesurer la consommation mémoire des communications entre les tâches et l'impact des affectations sur cette consommation. Du fait de l'architecture distribuée, une communication entre deux tâches affectées dans deux clusters différents nécessitera une copie des données entre les deux clusters et ajoutera un temps de latence entre la production et la consommation des données.

6.2 Évaluation de la mémoire à partir du modèleSDFG

L'évaluation du mapping proposée dans cette thèse utilise des arcs retours pour borner le marquage d'un buffer. Soit un graphe SDFG $\mathcal{G}, (\mathcal{T}, \mathcal{A})$ et $a = (t_i, t_j) \in \mathcal{A}$. On note $\overline{a} = (t_j, t_i)$ l'arc retour de a avec $p_{\overline{a}} = c_a$ et $c_{\overline{a}} = p_a$. La quantité de données consommée par le buffer borné est notée $\sigma_a = M_0(a) + M_0(\overline{a})$. La Figure 8 illustre un buffer (a gauche) et sa version bornée (à droite).

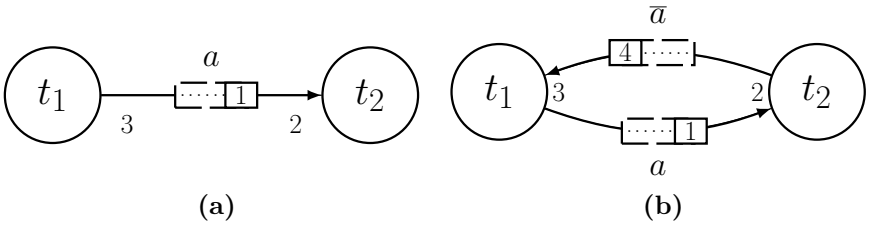


Figure 8 – Figure (a) illustre un buffer $a = (t_1, t_2)$ avec $p_a = 3$, $c_a = 2$ et $M_0(a) = 1$. Figure (b) représente la version bornée du buffer a avec $\sigma_a = 5$.

Lorsque deux tâches reliées par un arc sont affectées dans un même cluster leur consommation mémoire est σ_a. Si ces deux tâches sont affectées sur deux clusters différents les données transitent par le NoC. Pour modéliser ce NoC une nouvelle tâche appelée t_c (c pour communication) est ajoutée entre les deux tâches.

La Figure 9 illustre un buffer borné et un buffer borné partagé entre deux clusters.

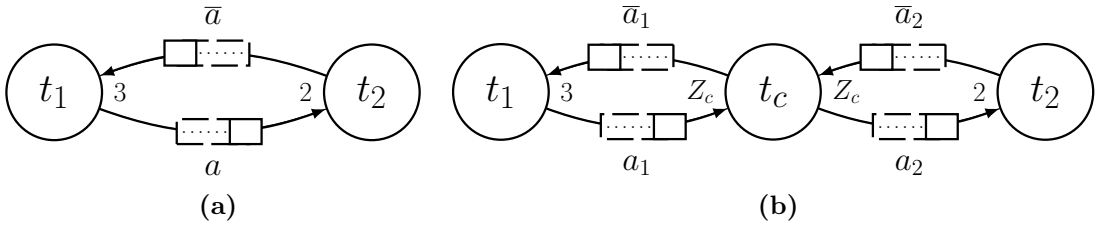


Figure 9 – Figure (a) illustre un buffer borné $a = (t_1, t_2)$. Figure (b) représente le même buffer borné partagé entre deux clusters. La tâche t_c représente le transfèrt de données entre deux clusters.

La consommation mémoire du buffer partagé entre deux clusters, illustré Figure 9(b), est de $\sigma_{a_1} = M_0(a_1) + M_0(\overline{a}_1)$ pour le cluster de la tâche t_1 et $\sigma_{a_2} = M_0(a_2) + M_0(\overline{a}_2)$ pour le cluster de la tâche t_2.

Cette thèse démontre que pour minimiser le marquage initial d'un buffer partagé entre deux clusters le taux de transfert noté Z_c sur la Figure 9(b) doit être fixé à $Z_c = gcd_a$. Un marquage initial minimal vivant est ensuite démontré avec et sans contrainte de débit. Ces résultats sont ensuite étendus au modèle CSDFG.

7 Résolution du problème de mapping

Afin de mettre en pratique la méthode d'évaluation présentée section précédente, plusieurs algorithmes ont été implémentés afin de résoudre un problem de mapping dont le but est de minimiser le nombre de clusters consommés. Ces algorithmes ont étés testés sur des applications réelles et aléatoires.

7.1 Présentation du problème de mapping

Le problème de mapping choisi à pour objectif l'affectation d'une application sur une architecture composée de clusters (eux même composés de processeurs) avec pour but, la minimisation du nombre de clusters utilisés. Chaque cluster est doté d'une certaine quantité de mémoire ce qui limite le nombre de tâche au sein d'un même cluster.

7.2 Représentation de la mémoire par un graphe multivalué

Cette section présente un graphe non-orienté et multivalué qui simplifie la représentation mémoire d'un dataflow. Le graphe est noté $\mathcal{U} = (\mathcal{T}, \mathcal{E})$ avec $\mathcal{T}$ l'ensemble des tâches correspondant aux tâches du dataflow et $\mathcal{E}$ l'ensemble des arêtes correspondantes au buffer borné du dataflow. A chaque arête $e = (t_i, t_j)$ on associe trois poids notés w_e, w_{ei} et w_{ej}.

1. w_e est la taille en mémoire si t_i et t_j sont affectées dans le même buffer. On a donc $w_e = \sigma_a$.
2. w_{ei} et w_{ej} représentent les quantités de mémoire des tâches t_i et t_j lorsque celles-ci sont affectées dans deux clusters différents. On a donc $w_{ei} = M_0(a_i) + M_0(\overline{a}_i)$ et $w_{ej} = M_0(a_j) + M_0(\overline{a}_j)$.

La Figure 10 illustre un graphe $\mathcal{U}$ et son SDFG associé.

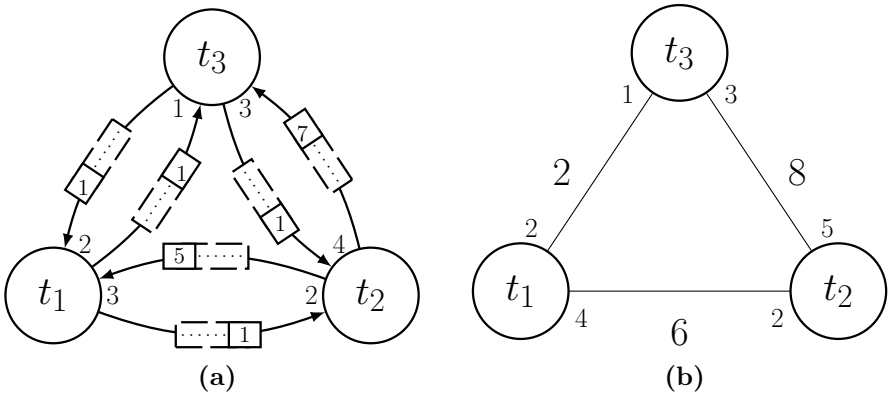


Figure 10 – Un graphe $\mathcal{U}$ à droite déduit du SDFG à gauche.

7.3 Formalisation du problème de mapping à l'aide de la programmation linéaire en nombre entier

Le problème de mapping a été formalisé en un Programme Linéaire en Nombre Entier (PLNE) décrit dans cette section. Soit $\mathcal{U} = (\mathcal{T}, \mathcal{E})$ un graphe non-orienté multi-valué, et $\mathcal{S}_c$ un ensemble de clusters.

On définit :

$$x_{t_i c} = \begin{cases} 1 \text{ si la tâche } t_i \in \mathcal{T} \text{ est affectée au cluster } c \in \mathcal{S}_c, \\ 0 \text{ sinon.} \end{cases}$$

$$y_{ec} = \begin{cases} 1 \text{ si } x_{t_i c} = 1 \text{ et } x_{t_j c} = 1 \text{ avec } e = (t_i, t_j) \in \mathcal{E}, c \in \mathcal{S}_c, \\ 0 \text{ sinon.} \end{cases}$$

$$z_c = \begin{cases} 1 \text{ si le cluster } c \in \mathcal{S}_c \text{ est utilisé,} \\ 0 \text{ sinon.} \end{cases}$$

$$\min \sum_{c \in \mathcal{S}_c} z_c$$

$$\sum_{e \in \mathcal{E}} \left(y_{ec}.w_e + \sum_{t \in e} (x_{t_i c} - y_{ec}).w_{ei} \right) \leq M_{max} \qquad \forall c \in \mathcal{S}_c$$

$$y_{ec} \geq x_{t_i c} + x_{t_j c} - 1 \qquad \forall e = (t_i, t_j) \in \mathcal{E},\ \forall c \in \mathcal{S}_c$$

$$y_{ec} \leq x_{t_i c} \qquad \forall e = (t_i, \cdot) \in \mathcal{E},\ \forall c \in \mathcal{S}_c$$

$$z_c \geq x_{t_i c} \qquad \forall t_i \in \mathcal{T},\ \forall c \in \mathcal{S}_c$$

$$\sum_{c \in \mathcal{S}_c} x_{t_i c} = 1 \qquad \forall t_i \in \mathcal{T}$$

$$x_{t_i c} \in \{0, 1\} \qquad \forall t_i \in \mathcal{T},\ \forall c \in \mathcal{S}_c$$

$$y_{ec} \in \{0, 1\} \qquad \forall e \in \mathcal{E},\ \forall c \in \mathcal{S}_c$$

$$z_c \in \{0, 1\} \qquad \forall c \in \mathcal{S}_c$$

La fonction objective du PLNE minimise le nombre de clusters utilisés. La première contrainte modélise la contrainte de mémoire pour chaque cluster avec M_{max} la mémoire disponible dans un cluster. Les deux contraintes suivantes forcent la variable y_{ec} à respecter sa définition. La quatrième contrainte force $z_c = 1$ si le cluster c est utilisé. La dernière contrainte impose qu'une tâche ne soit affectée que dans un seul cluster.

Le problème de mapping peut être réduit à un problème de bin-packing si on a $w_e = w_{ei} + w_{ej}$, $\forall e \in \mathcal{E}$. Grâce à cette réduction, nous avons démontré que le problème de mapping est NP-complet.

7.4 Algorithmes

Sept algorithmes sont proposés. Quatre d'entre eux sont inspirés d'algorithmes du bin-packing et trois autres ont été conçus et implémentés par Ahlam Mouaci durant son stage de fin d'étude. Les algorithmes ont été implémentés et testés sur des instances générées par `Turbine` et cinq instances réelles.

Le Tableau 2 présente les cinq applications réelles testées. Toutes les applications sont des CSDFG. La colonne "Size" indique la taille de l'application, la colonne "$||R||$" la somme de son vecteur de répétition, la colonne "$|\varphi|$" la somme des phases sur l'ensemble des tâches, enfin, "Cycles" indique si le graphe de l'application comporte des cycles.

Name	$\lvert\mathcal{T}\rvert$	$\lvert\mathcal{A}\rvert$	Size	$\lvert\lvert R\rvert\rvert$	$\lvert\varphi\rvert$	Cycles
BlackScholes	41	40	16KB	923	261	No
Echo	38	82	28KB	35003	45	Yes
H264	666	3128	1368KB	762	1375	Yes
JPEG2000	240	703	3807KB	24676	639	No
Pdetect	58	76	3859KB	58	4045	No

Table 2 – Caractéristiques des applications réelles mappées sous contrainte de débit.

Les applications réelles sont mappées avec un algorithme utilisant un système de matching pour fusionner les clusters. Les testes sont fait sur la version CSDFG et la version SDFG fonctionnellement équivalente. Le Tableau 3 récapitule les résultats. La colonne “Model” indique quel modèle est testé. La colonne “Time” le temps nécessaire pour effectuer le mapping en utilisant la méthode d'évaluation définie précédemment. La colonne “$|\mathcal{S}|$” donne le nombre de clusters utilisé par la solution du mapping, la colonne “M_{max}” la quantité de mémoire présente dans chaque cluster, la colonne “Mem” donne la quantité de mémoire utilisée par l'application sur le multi-cœurs, enfin, la colonne “Λ” indique si le mapping respecte la contrainte de débit imposée par l'application.

	Model	Time	$\lvert\mathcal{S}\rvert$	M_{max}	Mem	Λ
BlackScholes	SDFG	0.3s	3	23KB	48KB	Non
	CSDFG	0.6s	3	8KB	17KB	Oui
Echo	SDFG	0.3s	4	10KB	30KB	Oui
	CSDFG	0.3s	3	10KB	29KB	Oui
H264	SDFG	1mn3s	18	90KB	1328KB	Oui
	CSDFG	1mn1s	18	90KB	1372KB	Oui
JPEG2000	SDFG	3.8s	2	3532KB	6168KB	Oui
	CSDFG	5.6s	3	1695KB	3807KB	Oui
Pdetect	SDFG	3mn31s	4	928KB	3643KB	Non
	CSDFG	5mn11s	4	928KB	3859KB	Oui

Table 3 – Résultats du mapping sur des applications réelles.

Comme on peut le remarquer sur le Tableau 3, le temps de calcul du mapping, incluant l'évaluation du mapping, est plus restreint sur le modèle SDFG que le modèle CSDFG. Cependant, le modèle CSDFG donne de meilleurs résultats concernant la mémoire consommée par l'application et le respect de la contrainte de débit. Certaines applications n'ont pas de solution qui respecte la contrainte de débit, cela est dû à l'architecture et aux communications entre clusters.

8 Conclusion

Cette thèse apporte deux principales contributions, elle présente un générateur de graphes dataflow appelé `Turbine` et propose une nouvelle méthode d'évaluation d'un mapping sur une architecture à mémoire distribuée.

Le générateur de graphes dataflow appelé `Turbine` est capable de générer des SDFG, des CSDFG et des PCG jusqu'à 10000 tâches. `Turbine` propose aussi de nombreuses fonctionnalités pour analyser des graphes dataflow :

- calcul du débit;
- calcul d'un ordonnancement 1-periodic;
- calcul d'un ordonnancement au plus tôt;

- calcul du marquage initial;
- calcul des facteurs de répétition;
- normalisation/dé-normalisation;
- dessiner un dataflow (sortie PDF).

En seconde partie, une méthode d'évaluation d'un mapping est proposée. D'abord appliquée au modèle SDFG, elle est ensuite étendue au modèle CSDFG. Le but de cette méthode est d'évaluer la consommation mémoire pour des applications de grandes tailles (plusieurs milliers de tâches) en un temps raisonnable. La méthode d'évaluation est testée avec sept algorithmes sur des instances générées aléatoirement par `Turbine` et sur une sélection de cinq applications réelles.

Chapter 1

Introduction

Contents

Unlike a general-purpose computer, an embedded system is dedicated to particular tasks, often with real-time constraints. Embedded systems are ubiquitous and can be found in home appliances, vehicles, communication and multimedia equipment or medical equipment. Typical requirements for embedded systems are low power consumption, high reliability and small size.

Dedicated processors fulfill all the criteria required for embedded systems. Since they focus on only one application, their design can be greatly optimized which makes their design long and expensive. The opposite solution consists in using a general purpose processor. These processors are a good answer in a context of a rapid design. Development of the applications on those processors is fast and they offer flexibility since one processor could run many different applications.

High performance low power general-purpose processors should not run at high frequency, instead, designers tend to multiplicate processing elements. There processors are called multi-core or many-core and are composed of dozens to thousands of cores linked with a Network on Chip (NoC).

Even with high potential performance a multi-core or many-core architecture requires special care in decomposing an application into computational tasks and dispatching them among the processing elements in order to meet the performance requirements. This dispatching is called a mapping and is considered as one of the most urgent problems of this decade [Marwedel et al., 2011].

A Model of Computation (MoC) is a set of definition used to represent algorithms, it provides tools to evaluate the performance of applications. A dataflow MoC is a representation of the application using weighted graphs. Among the dataflow MoC the Synchronous Dataflow Graph (SDFG) model is very popular. The SDFG is a static MoC used to detect deadlock or to evaluate the performance of the application especially its throughput, latency or memory consumption.

1.1 Contributions

This thesis is based on the SDFG MoC and two of its extensions, the Cyclo-Static Dataflow Graph (CSDFG) and the Phased Computation Graph (PCG).

The first contribution is the extension of important notions from the SDFG model to the PCG model: consistency, useful tokens, normalization, precedence constraints and liveness. Since the PCG is an extension of the CSDFG model, itself an extension of the SDFG model, these notions are first presented on the SDFG model, then on the CSDFG model, to be extended finally to the PCG model.

The second contribution of this thesis is the implementation of `Turbine`, a dataflow generator, which handles the SDFG, CSDFG and the PCG models.

The third and main contribution is an evaluation method for an arbitrary mapping of one such model of an application on an architecture with distributed memory. This method is presented for the SDFG and CSDFG models. The method is then applied to a mapping problem where the main objective is to minimize the number

of processing elements (with their own memory) employed for the application while satisfying the memory constraints.

1.2 Thesis Organization

Chapter 2 presents the following MoCs: Kahn Process Network (KPN), Homogeneous Synchronous Dataflow Graph (HSDFG), Synchronous Dataflow Graph (SDFG), Cyclo-Static Dataflow Graph (CSDFG), Computation Graph (CG), CSDFG with initial phases and Phased Computation Graph (PCG). Then, the SDFG model is used to introduce the notions of repetition vector, consistency, useful tokens, normalization and liveness. A sufficient condition of liveness with a low computational complexity is described. Finally, three types of scheduling are presented, As Soon As Possible (ASAP), one-periodic and K-periodic.

Chapter 3 extends the notions of repetition vector, consistency, useful tokens, normalization and liveness to the CSDFG and PCG models. The sufficient condition of liveness is extended to both models. The three scheduling methods are then presented for the CSDFG model.

Chapter 4 presents the dataflow generator `Turbine`. The generator is compared with two of its competitors, SDF For Free (SDF3) and PREESM. The experiments compare generation times and numbers of tokens for the generated dataflow graphs. They demonstrate the improvement between consecutive versions of `Turbine` and the quality of the approximation provided by the sufficient condition of liveness.

Chapter 5 presents our method for the evaluation of a memory allocated to a mapping on a distributed memory architecture. The method is introduced on the SDFG model. Two versions of memory evaluation are proposed: under liveness or throughput constraint. The first uses the sufficient condition of liveness presented earlier. The second introduces a throughput constraint where the throughput is evaluated using a one-periodic schedule, where each task is executed with its own period. The two versions of the evaluation method are then extended to the CSDFG model.

Chapter 6 illustrates the use of our memory evaluation method when mapping a dataflow application on a distributed memory architecture. The targeted architecture is composed of clusters. An un-directed tri-valuated graph is proposed to characterize the memory consumption of an application modeled by an SDFG or a CSDFG. An H263 encoder is used for illustration. The evaluation method is then employed in conjunction with mapping heuristics to minimize the number of clusters used by the application. Several heuristics, some inspired by bin-packing, are compared on random and real applications.

Chapter 7 concludes the thesis and presents some perspectives opened by our work.

1.3 Mathematical Notation

We denote by $\mathbb{N}^*$ the set of strictly positive integer, $\mathbb{R}^+$ the set of positive reals and $\mathbb{R}^{*+}$ the set of strictly positive reals. The number of elements of in set $\mathcal{E}$ is denoted $|\mathcal{E}|$. Let $x \in \mathbb{R}$, the operations $\lceil x \rceil$ and $\lfloor x \rfloor$ denote respectively the smallest integer greater than x and the largest integer smaller than x. Similarly, $\lceil x \rceil_z$ and $\lfloor x \rfloor_z$ are respectively the smallest multiple of z greater than x and the largest multiple of z smaller than x. For instance $\lceil 5 \rceil_3 = 6$ and $\lfloor 5 \rfloor_3 = 3$. We use the notations *gcd* and *lcm* for the greatest common divisor and the least common multiple of elements of $\mathbb{N}^*$.

Chapter 2

Dataflow Models

Contents

2.1 Introduction

Among the many Models of Computation (MoCs), the Kahn Process Network (KPN), one of the first attempts to model the dataflow of an application using First in First out (FIFO) queues between tasks, is too expressive to allow the analysis tools needed by the dataflow community. The Synchronous Dataflow Graph (SDFG) model is now the most widely used in the dataflow community as it provides a good compromise between expressiveness and analysis complexity. Other models have emerged after the SDFG model that are more expressive yet amenable to automated analysis.

This chapter provides an overview of dataflow models considered in this thesis while introducing important notions concerning them.

Section 2.2 presents the dataflow models. Section 2.3 illustrates the gain in expressiveness afforded by the Cyclo-Static Dataflow Graph (CSDFG) and Phased Computation Graph (PCG) models through two examples. In Section 2.4, the SDFG behavior is discussed and basic notions are explained. Execution schedules are described in Section 2.5. Section 2.6 concludes the chapter.

2.2 Dataflow Models of Computation

Dataflow modeling is a widely used method for specifying the functionality of embedded systems. It first appears in the context of parallel computation in the 60's [Karp and Miller, 1966]. Dataflow modeling is used in the design process of embedded systems, real time systems, Digital Signal Processing (DSP) systems and parallel computing systems. Computations and communications are modeled by actors (nodes) and communication channels (arcs). A communication channel is unidirectional between two tasks and behaves as a FIFO buffer. Dataflow MoCs are tokens stream models [Davis and Keller, 1982], in which data are seen as discrete streams of tokens, the data values being abstracted out.

This section presents the most common dataflow models:

- Khan Process Network in Section 2.2.1
- Homogeneous Synchronous Dataflow Graph in Section 2.2.2
- Synchronous Dataflow Graph in Section 2.2.3
- Cyclo Static Dataflow Graph in Section 2.2.4
- Computation Graph in Section 2.2.5
- Cyclo Static Dataflow Graph with initial phases in Section 2.2.6
- Phased Computation Graph in Section 2.2.7.

2.2.1 Kahn Process Network model

The Kahn Process Network (KPN) model [Kahn, 1974] was introduced in 1974 by Kahn as a parallel programming model. In [Buck and Lee, 1993] the authors proved that the Boolean Dataflow Graph model, a subclass of the KPN is Turing complete, making the KPN himself Turing complete.

A KPN is composed of computing stations (nodes) linked by communication lines (arcs) which behave as FIFOs with infinite memory. A computing station has a sequential behavior, thus it cannot output data on more than one communication line at a given time. A communication line has only one input and one output. If there is not enough data on the communication line toward a computing station, the station waits until enough data is available to perform a computation. The amount of data produced or consumed by a computing station is fixed.

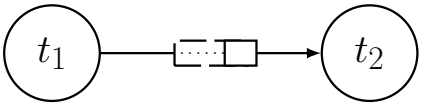


Figure 2.1 – Kahn Process Network with two computing stations linked by a communication line.

The KPN model has a high expressiveness but does not allow for deadlock analysis nor evaluation of bounds on capacity (maximum amount of data that can be reached on channels).

Models presented below are all static subclasses of the KPN model. In constrast with the KPN model, static dataflow models are not Turing complete. These models consider data as simple tokens, meaning that data values have no impact on the system's behavior. Static models are non-reconfigurable and do not to express conditional state, data dependent iteration or recursion. A considerable benefit from these restrictions is that they are decidable, fairly easy to analyze, and can be scheduled at compile time [Lee and Parks, 1995].

2.2.2 Homogeneous Synchronous Dataflow model

The Homogeneous Synchronous Dataflow Graph (HSDFG) model [Moreira et al., 2010], also known as Single Rate Dataflow model, is composed of actors (or tasks) and arcs (equivalent to communication links). An arc symbolizes a FIFO queue. This MoC existed before the apparition of the SDFG model (discussed in the next section) under the name of Event Graph in the Petri net community.

A Homogeneous Synchronous Dataflow Graph (HSDFG) is a directed graph $\mathcal{G}_{hsdf} = (\mathcal{T}, \mathcal{A}, \mathcal{M}, \mathcal{L})$

- $\mathcal{T}$ is the set of actors,
- $\mathcal{A}$ is the set of arcs,

- $\mathcal{M}$ is the set of initial markings: $\mathcal{M} = \{M_0(a)|a \in \mathcal{A}\}$ is the set of values taken by the function $M_0 : \mathcal{A} \to \mathbb{N}$ which associates to each arc $a \in \mathcal{A}$ an initial number of tokens.
- $\mathcal{L}$ is the set of execution times: $\mathcal{L} = \{\ell(t_i)|t_i \in \mathcal{T}\}$ is the set of values taken by the function $\ell : \mathcal{T} \to \mathbb{R}^+$ which associates to each actor $t_i \in \mathcal{T}$ an execution time $\ell(t_i) = \ell_i$.

The initial marking gives the number of tokens present on each arc at the beginning of the application, where the tokens represent the data items manipulated by the application.

Figure 2.4 shows an HSDFG with two tasks t_1 and t_2 linked by an arc a. The rectangle on the arc represents the FIFO queue, with the initial marking shown inside.

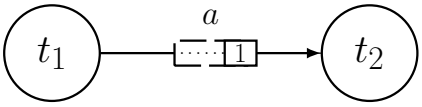


Figure 2.2 – An HSDFG graph composed of an arc $a = (t_1, t_2)$ with $M_0(a) = 1$.

The execution times of the actors do not yet appear in the figure. They will be shown when scheduling and performance are discussed. This remark also applies to the models described below.

2.2.3 Synchronous Dataflow model

The Synchronous Dataflow Graph (SDFG) model has been introduced in [Lee and Messerschmitt, 1987]. This model is also known as Multi-Rate Dataflow model in the signal processing community or Weighted Event Graph model in the Petri net community.

An SDFG is a directed weighted graph $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ where:

- $\mathcal{T}$ is the set of actors,
- $\mathcal{A}$ is the set of arcs,
- $\mathcal{P}$ is the set of production weights: $\mathcal{P} = \{p(a)|a \in \mathcal{A}\}$ is the set of values taken by the function $p : \mathcal{A} \to \mathbb{N}^*$ which associates to each arc $a \in \mathcal{A}$ a production weight $p(a) = p_a$.
- $\mathcal{C}$ is the set of production weights: $\mathcal{C} = \{c(a)|a \in \mathcal{A}\}$ is the set of values taken by the function $c : \mathcal{A} \to \mathbb{N}^*$ which associates to each arc $a \in \mathcal{A}$ a consumption weight $c(a) = c_a$.
- $\mathcal{M}$ is the set of initial markings: $\mathcal{M} = \{M_0(a)|a \in \mathcal{A}\}$ is the set of values taken by the function $M_0 : \mathcal{A} \to \mathbb{N}$ which associates to each arc $a \in \mathcal{A}$ an initial number of tokens $M_0(a)$.
- $\mathcal{L}$ is the set of execution times: $\mathcal{L} = \{\ell(t_i)|t_i \in \mathcal{T}\}$ is the set of values taken by

the function $\ell : \mathcal{T} \to \mathbb{R}^+$ which associates to each actor $t_i \in \mathcal{T}$ an execution time $\ell(t_i) = \ell_i$.

Three weights are associated to each arc: the production rate, the consumption rate and the initial marking. The production rate p_a is the number of data items produced by t_i on the arc $a = (t_i, t_j)$ at the end of each computation and the consumption rate c_a is the number of items consumed by t_j from a at the beginning of each computation. Note that t_j is not executed if the number of items in the FIFO queue is insufficient. The initial marking of the arc a is denoted $M_0(a)$ and indicates the number of items present at the beginning of the execution of the application. The computation of a task t_i requires ℓ_i time units.

Figure 2.3 shows an SDFG composed of two tasks t_1 and t_2 linked by an arc a. Production and consumption weights (or rates) are indicated respectively at the origin and the end of the arc.

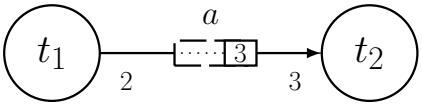


Figure 2.3 – An SDFG graph with two tasks t_1 and t_2 and an arc $a = (t_1, t_2)$ with $p_a = 2$, $c_a = 3$ and $M_0(a) = 3$.

The expansion is a technique which consists in converting an SDFG into an HSDFG. This transformation is done at the expense of a possibly exponential growth of the size of the graph. A first version of the expansion is presented in [Lee and Messerschmitt, 1987]. The transformation is based on token ordering. If an actor t of an SDFG graph produces p tokens, the expanded HSDFG graph has p outgoing arcs connected to the consumer. Also, each actor is duplicated as many times as it is executed in one iteration, where an iteration consists in executing each actor just as many times as necessary to bring the system back to its initial marking.

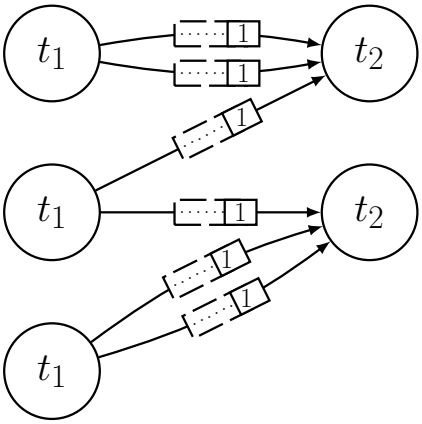


Figure 2.4 – An HSDFG equivalent to the SDFG in Figure 2.3 obtained by expansion. Example from [Lee and Messerschmitt, 1987].

Several other versions of the expansion have been proposed [Marchetti and Munier-Kordon, 2009a; de Groote et al., 2012; Sriram and Bhattacharyya,

2009; Geilen and Stuijk, 2010; Saha et al., 2006]. For instance, in [Ito and Parhi, 1994] a version is presented which focuses of the conservation of the precedence constraints. Compared to the original version of the expansion this results in an HSDFG with fewer arcs and speeds up the algorithms working on the HSDFG model.

The HSDFG obtained by expanding an SDFG is potentially of exponential size compared to the SDFG. This implies that polynomial time algorithms on HSDFG models obtained by expansion are not necessarily polynomial with respect to the underlying SDFG models.

2.2.4 Cyclo-Static Dataflow model

The Cyclo-Static Dataflow Graph (CSDFG) model [Bilsen et al., 1995] is an extension of the SDFG model with consumption and production of actors decomposed into phases executed cyclically. Each actor has a fixed number of phases and each phase produces or consumes a specific number of items.

A CSDFG is a directed weighted graph $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ where:

- $\mathcal{T}$ is the set of actors,
- $\mathcal{A}$ is the set of arcs,
- $\mathcal{P}$ is the set of production vectors: $\mathcal{P} = \{p(a)|a \in \mathcal{A}\}$ is the set of values taken by the function p which associates to each arc $a = (t_i, \cdot) \in \mathcal{A}$ a production vector $p(a) = [p_a(1), \cdots, p_a(\varphi_i)] \in (\mathbb{N})^{\varphi_i}$ where φ_i is the number of phases associated to the actor t_i and with $\sum_{k=1}^{\varphi_i} p_a(k) = p_a > 0$.
- $\mathcal{C}$ is the set of consumption vectors: $\mathcal{C} = \{c(a)|a \in \mathcal{A}\}$ is the set of values taken by the function c which associates to each arc $a = (\cdot, t_i) \in \mathcal{A}$ a consumption vector $c(a) = [c_a(1), \cdots, c_a(\varphi_i)] \in (\mathbb{N})^{\varphi_i}$ where φ_i is the number of phases associated to the actor t_i and with $\sum_{k=1}^{\varphi_i} c_a(k) = c_a > 0$.
- $\mathcal{M}$ is the set of initial markings: $\mathcal{M} = \{M_0(a)|a \in \mathcal{A}\}$ is the set of values taken by the function $M_0 : \mathcal{A} \to \mathbb{N}$ which associates to each arc $a \in \mathcal{A}$ an initial number of tokens.
- $\mathcal{L}$ is the set of vectors of execution times: $\mathcal{L} = \{\ell(t_i)|t_i \in \mathcal{T}\}$ is the set of values taken by the function ℓ which associates to each actor $t_i \in \mathcal{T}$ a vector of execution times $\ell(t_i) = [\ell_i(1), \cdots, \ell_i(\varphi_i)] \in (\mathbb{R}^+)^{\varphi_i}$ where φ_i is the number of phases associated to the actor t_i.

The k^{th} phase of task t_i is denoted $t_i(k)$, $k \in \{1, \cdots, \varphi_i\}$, and its execution time $\ell_i(k)$. The cyclic execution of actors is expressed as follows: the k^{th} firing of actor t_i executes phase k *mod* φ_i. The number of data items produced on (*resp.* consumed from) arc $a = (t_i, t_j)$ during phase k is denoted $p_a(k)$ (*resp.* $c_a(k)$). The total production (*resp.* consumption) over all the phases is denoted $p_a = \sum_{k=1}^{\varphi_i} p_a(k)$ (*resp.* $c_a = \sum_{k=1}^{\varphi_j} c_a(k)$).

Figure 2.5 shows an example of CSDFG graph with one phase for task t_1 and two phases for task t_2. Production and consumption vectors are shown between square

brackets.

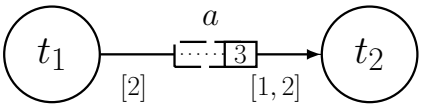


Figure 2.5 – A CSDFG graph with a single arc $a = (t_1, t_2)$ and weights $[p_a(1)] = [2]$, $[c_a(1), c_a(2)] = [1, 2]$ and $M_0(a) = 3$. Thus $\varphi_1 = 1$, $\varphi_2 = 2$, $p_a = 2$ and $c_a = 3$.

2.2.5 Computation Graph model

The Computation Graph (CG) model [Karp and Miller, 1966] is another extension of the SDFG model, with a threshold associated to a consumption. The threshold indicates when the number of data items on arc $a = (\cdot, t)$ is sufficient to execute actor t. At the beginning of a computation the actor peeks, that is reads from the FIFO queue, as many data items as indicated by a threshold, denoted θ_a, and consumes c_a of these items. The consumption rate c_a must clearly be smaller than the threshold on consumption, $c_a \leq \theta_a$. Note that if the number of items in the FIFO queue is less than θ_a the actor cannot be executed.

Figure 2.6 shows an example of CG with a threshold on consumption. The consumption weight and the threshold appear separated by a colon, $c_a : \theta_a$.

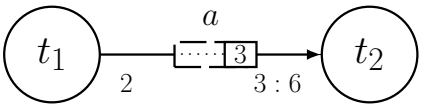


Figure 2.6 – A CG with a single arc $a = (t_1, t_2)$ and a threshold $\theta_a = 6$. Task t_2 requires 6 data items in the FIFO queue to be executed and, when it is executed, it consumes only $c_a = 3$ data items.

2.2.6 Cyclo-Static Dataflow model with initial phases

Expressing the way an actor t_i is initialized can be done with initial phases. The initialization is decomposed into σ_i initial phases executed once, at the start of the application. The $\sigma_i + \varphi_i$ phases of t_i are numbered from $1 - \sigma_i$ to φ_i, with $\{1 - \sigma_i, \cdots, 0\}$ the set of initial phases and $\{1, \cdots, \varphi_i\}$ the set of cyclic execution phases.

Figure 2.7 shows a CSDFG with two initial phases and two cyclic phases for actor t_2. The initial phases are indicated between parentheses before the cyclic phases.

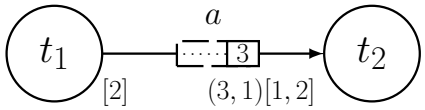


Figure 2.7 – A CSDFG with initial phases. The graph consists in a single arc $a = (t_1, t_2)$ with $[p_a(1)] = [2]$, $(c_a(-1), c_a(0)) = (3, 1)$, and $[c_a(1), c_a(2)] = [1, 2]$, thus $\varphi_1 = 1$, $\sigma_1 = 0$, $\varphi_2 = 2$ and $\sigma_2 = 2$.

Note that the CSDFG with initial phases is an intermediate model introduced for the purpose of the presentation of the initialization steps for static dataflow graphs.

2.2.7 Phased Computation Graph model

The Phased Computation Graph (PCG) [Thies et al., 2002] extends the CSDFG model with initial phases by associating to each consumption phase a threshold as in the CG model. Every actor's execution is divided into initial phases followed by cyclic execution phases. As explained in the previous section, initial phases are executed once at the start of the application.

An PCG graph is a directed weighted graph $\mathcal{G} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ where:

- $\mathcal{T}$ is the set of actors,
- $\mathcal{A}$ is the set of arcs,
- $\mathcal{P}$ is the set of production vectors: $\mathcal{P} = \{p(a)|a \in \mathcal{A}\}$ is the set of values taken by the function p which associates to each arc $a = (t_i, \cdot) \in \mathcal{A}$ a production vector $p(a) = (p_a(1-\sigma_i), \cdots, p_a(0))\,[p_a(1), \cdots, p_a(\varphi_i)] \in (\mathbb{N})^{\sigma_i} \times (\mathbb{N})^{\varphi_i}$ where σ_i is the number of initial phases and φ_i the number of cyclic phases associated to the actor t_i, with $\sum_{k=1}^{\varphi_i} p_a(k) = p_a > 0$. The initial production rates appear between the parentheses and the cyclic production rates appear between the square brackets.
- $\mathcal{C}$ is the set of consumption vectors: $\mathcal{C} = \{c(a)|a \in \mathcal{A}\}$ is the set of values taken by the function c which associates to each arc $a = (\cdot, t_i) \in \mathcal{A}$ a consumption vector $c(a) = (c_a(1-\sigma_i), \cdots, c_a(0))\,[c_a(1), \cdots, c_a(\varphi_i)] \in (\mathbb{N})^{\sigma_i} \times (\mathbb{N})^{\varphi_i}$ where σ_i is the number of initial phases and φ_i the number of cyclic phases associated to the actor t_i, with $\sum_{k=1}^{\varphi_i} c_a(k) = c_a > 0$. The initial consumption rates appear between the parentheses and the cyclic consumption rates appear between the square brackets.
- Θ is the set of threshold vectors: $\Theta = \{\theta(a)|a \in \mathcal{A}\}$ is the set of values taken by the function θ which associates to each arc $a = (\cdot, t_i) \in \mathcal{A}$ a threshold vector $\theta(a) = (\theta_a(1-\sigma_i), \cdots, \theta_a(0))\,[\theta_a(1), \cdots, \theta_a(\varphi_i)] \in (\mathbb{N})^{\sigma_i} \times (\mathbb{N})^{\varphi_i}$ where σ_i is the number of initial phases and φ_i the number of cyclic phases associated to the actor t_i, with $c_a(k) \leq \theta_a(k), \forall k \in \{1-\sigma_i, \cdots, \varphi_i\}$. The initial thresholds appear between the parentheses and the cyclic thresholds appear between the square brackets.
- $\mathcal{M}$ is the set of initial markings: $\mathcal{M} = \{M_0(a)|a \in \mathcal{A}\}$ is the set of values

taken by the function $M_0 : \mathcal{A} \rightarrow \mathbb{N}$ which associates to each arc $a \in \mathcal{A}$ an initial number of tokens.

- $\mathcal{L}$ is the set of vector of execution times: $\mathcal{L} = \{\ell(t_i)|t_i \in \mathcal{T}\}$ is the set of values taken by the function $\ell(t_i)$ which associates to each actor $t_i \in \mathcal{T}$ a vector of execution times $\ell(t_i) = (\ell_i(1-\sigma_i), \cdots, \ell_i(0))\,[\ell_i(1), \cdots, \ell_i(\varphi_i)] \in (\mathbb{R}^+)^{\sigma_i} \times (\mathbb{R}^+)^{\varphi_i}$ where σ_i is the number of initial phases and φ_i the number of cyclic phases associated to the actor t_i. The initial execution times appear between the parentheses and the cyclic execution times appear between the square brackets.

Every consumption phase of a task t_i on arc $a = (\cdot, t_i)$ has a threshold $\theta_a(k)$, such that $c_a(k) \leq \theta_a(k)$, where $k \in \{1-\sigma_i, \cdots, 0\} \cup \{1, \cdots, \varphi_i\}$ is the set of phases of task t_i.

Figure 2.8 pictures a PCG graph where task t_2 has two initial phases and two cyclic phases. A threshold appears only when strictly superior to the associated consumption rate.

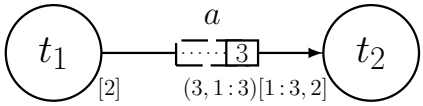


Figure 2.8 – A PCG with a single arc $a = (t_1, t_2)$ and thresholds $\theta_a(-1) = 3$ and $\theta_a(0) = 3$ for the $\sigma_2 = 2$ initial phases and $\theta_a(1) = 3$ and $\theta_a(2) = 2$ for the $\varphi_2 = 2$ cyclic execution phases.

2.3 Comparison between MoCs

The remainder of the thesis exploits the SDFG, CSDFG and PCG models. This section illustrates the benefit of the CSDFG and PCG models. Section 2.3.1 compares an SDFG and a CSDFG version of an H263 encoder. Section 2.3.2 illustrates the PCG behavior with a three-tap filter. Finally, Section 2.3.3 explains the notion of reentrance.

2.3.1 SDFG and CSDFG

The CSDFG model is an extension of the SDFG model, it refines actor execution into phases. This change allows a better representation of a dataflow application and a finer analysis. We illustrate the benefit of a CSDFG compared to an SDFG model with Figure 2.9 from [Oh and Ha, 2002], showing an CSDFG and a SDFG version of the H263 encoder. The encoder is composed of 8 actors: Read From Device (RFD), Motion Estimation (ME), Distributor (Dist), Motion Block Encoding (MBE), Motion Block Decoding (MBD), Motion Compensation (MC), Variable Length Coding (VLC) and Write To Device (WTD).

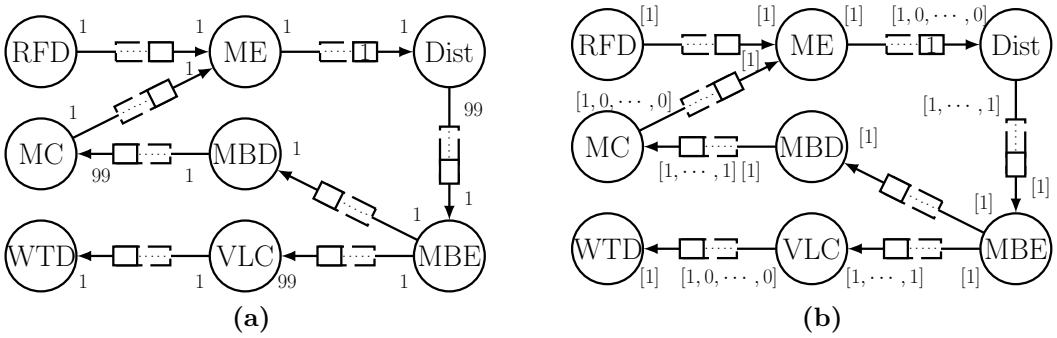


Figure 2.9 – (a) The H263 encoder modeled by an SDFG; (b) the same application modeled by a CSDFG, where the vectors $[1, 0, \cdots, 0]$ and $[1, \cdots, 1]$ have 99 elements.

The H263 encoder partitions images into blocks which are treated individually and also recombined for motion estimation to give a better compression of the next image. With the SDFG model, the *motion block encoding* (MBE) has to wait until the *distributor* (Dist) has finished to decompose the image into blocks. With the CSDFG model, *motion block encoding* can start immediately after the first block is stored into the FIFO queue by the *distributor*.

Using an As Soon As Possible (ASAP) schedule (formally described later), if one block takes one time unit to be produced by actor Dist, *motion block encoding* will start after 1 time unit with the CSDFG model while it will wait 99 time units with the SDFG model. Thus latency is significantly reduced with the CSDFG.

2.3.2 PCG expressiveness

The expressiveness of the PCG may be demonstrated by showing that it allows a more compact representation of the same applications than other static dataflow graph models. This section illustrates this improvement with the example shown in Figure 2.10.

Figure 2.10 is a three-tap filter from [Parhi, 1995]. The input stream is decomposed in even and odd samples, $x(2k)$ and $x(2k+1)$, similarly the output stream is decomposed into streams of even and odd samples, $y(2k)$ and $y(2k+1)$. Values a_0, a_1 and a_2 are coefficients. Block D performs a one-sample delay operation on its input sequence. Nodes marked $\times$ and $+$ represent arithmetic operations.

To compute $y(2k)$ the filter combines $x(2k)$, $x(2k-1)$ and $x(2k-2)$ while $y(2k+1)$ is a combination of $x(2k+1)$, $x(2k)$ and $x(2k-1)$. This representation allows parallel computation of $y(2k)$ and $y(2k+1)$

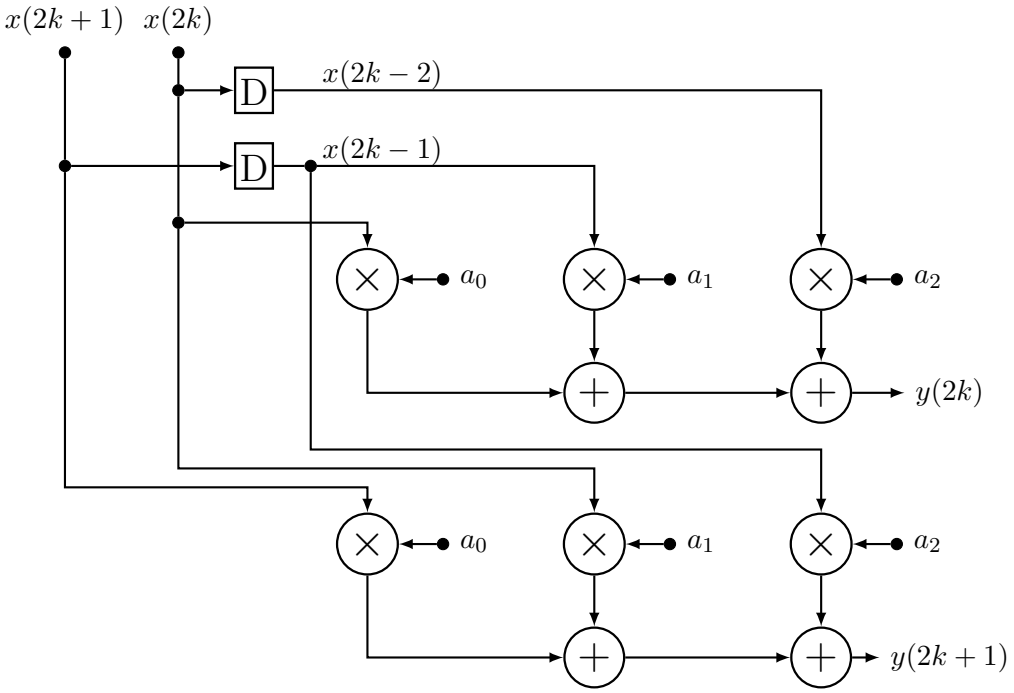


Figure 2.10 – Three-tap filter from [Parhi, 1995].

Figure 2.11 illustrates the three-tap filter modeled by a SDFG (Figure 2.11(a)) and a PCG (Figure 2.11(b)). The presence of initial tokens in the SDFG symbolizes the necessary delay to feed y_{2k} and y_{2k+1} with the previous values of x_{2k} and x_{2k+1}. Omitted weights are equal to 1. The initial tokens represent initial data. The behavior is represented in the PCG by using a threshold to peek at the last three items but consume only the oldest one. Note that we consider x_0 to be the first entry value.

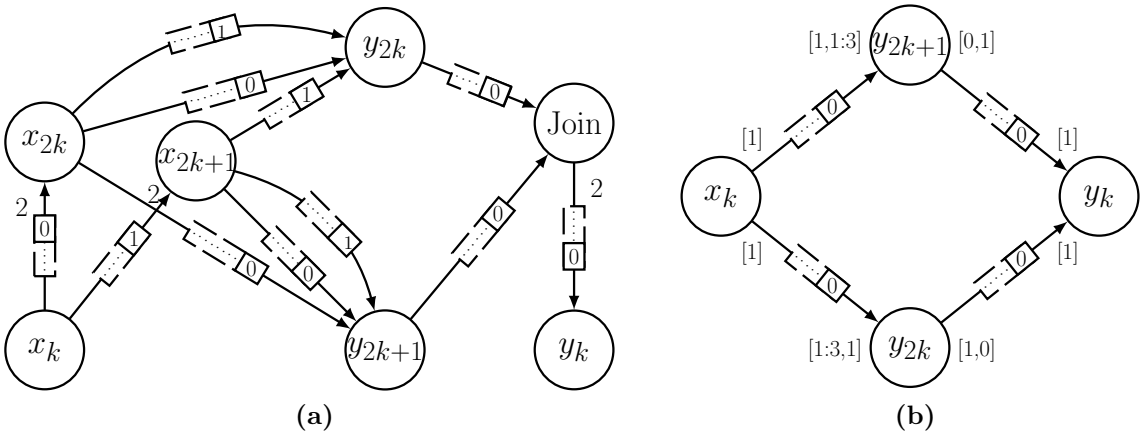


Figure 2.11 – (a) An SDFG representing the three tap filter of Figure 2.10; (b) the same application modeled with a PCG.

As illustrated with Figure 2.11(a) an initial token could also be considered as a delay. In this thesis, tokens will always be considered as data items.

2.3.3 Reentrance

The reentrance behavior (opposed to sequential behavior), also called overlapping behavior allows a task of a MoC to start a new execution without necessarily waiting for the end of its previous execution.

Originally the authors of [Lee and Messerschmitt, 1987] were considering sequential behavior. To improve the concurrency in a schedule the behavior of a MoC may be reentrant. For instance, in [Desnos et al., 2016] the authors propose a technique to reduce the memory footprint of a SDFG with a reentrant behavior.

Modeling reentrant behavior is a great improvement on the SDFG model since introduction of self-concurrency for the actor increase does not necessarily demand many modifications on existing techniques and gives additional freedom to the scheduling.

The work of this thesis has been made with non-reentrance constraints, as assumed in previous works. However, reentrancy can be exploited in a future work and will be discussed later. The non-reentrant behavior can be expressed by adding a self-loop to each task of a MoC with $p_a = c_a = 1$ and $M_0(a) = 1$ as illustrated in Figure 2.12. Adding self-loops could also be used to control of the reentrant behavior, for instance, a self-loop with an initial marking equal to 2 allows two simultaneous executions of the task. This technique is easily extended to the CSDFG and PCG models. For better readability figures in the sequel are shown without self-loops.

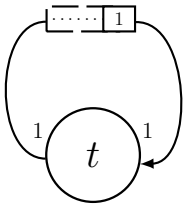


Figure 2.12 – A SDFG with task t and its self-loop.

2.4 Synchronous Dataflow behavior

This section focuses on the SDFG model. It presents basic SDFG model properties and recalls important results. The notions of consistency, useful tokens, normalization and liveness are introduced and explained. These notions will be extended to the CSDFG and the PCG model in the next chapter.

Section 2.4.1 recalls the notion of consistency and indicates how repetition factors are computed. Section 2.4.2 explains the notion of useful tokens. Normalization is described in Section 2.4.3. Section 2.4.4 recalls the notion of liveness; a sufficient condition of liveness and an algorithm to compute a live initial marking are also given.

2.4.1 Repetition vector and consistency

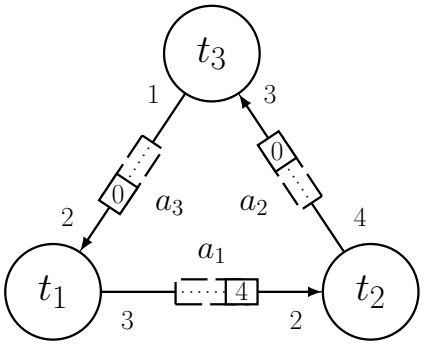
Consistency has been introduced in the original paper on the SDFG model [Lee and Messerschmitt, 1987]. Recall that p_a (*resp.* c_a) is the production (*resp.* consumption) rate on an arc a of an SDFG. Considering an SDFG $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ its topology matrix Γ, of size $|\mathcal{A}| \times |\mathcal{T}|$, is defined by:

$$\Gamma_{ai} = \begin{cases} p_a & \text{if } a = (t_i, \cdot) \\ -c_a & \text{if } a = (\cdot, t_i) \\ 0 & \text{otherwise} \end{cases} .$$

A connected SDFG is consistent if Γ has rank $|\mathcal{T}| - 1$. Then there exists a non-zero vector $R = [R_1, \cdots, R_{|\mathcal{T}|}] \in (\mathbb{N}^*)^{|\mathcal{T}|}$ with coprime components such that $\Gamma.R^T = 0$. The vector R is called the repetition vector.

The repetition factor R_i of a task $t_i \in \mathcal{T}$ gives the minimal number of times task t_i must be executed for the initial marking of the graph to be reached again. A sequence of task executions where each task is executed R_i times is called a system iteration or simply, an iteration. Figure 2.13(a) depicts an SDFG with repetition vector $R = [2, 3, 4]$.

Consistency ensures the existence of the repetition vector. The repetition factors must respect the following balance equations: $\forall a = (t_i, t_j) \in \mathcal{A}$, $p_a.R_i = c_a.R_j$.



$$\Gamma = \begin{bmatrix} 3 & 0 & -2 \\ -2 & 4 & 0 \\ 0 & -3 & 1 \end{bmatrix}$$

Figure 2.13 – The SDFG on the left has for topology matrix the matrix Γ on the right. The SDFG is consistent and its repetition vector is $R = [2, 3, 4]$

Consistency guarantees that if the initial marking is sufficient for a complete iteration to be feasible then the number of times the tasks can be executed may grow unbounded while keeping the number of tokens bounded.

2.4.2 Useful tokens

The notion of useful tokens has been introduced in [Marchetti and Munier-Kordon, 2009a] for the SDFG model. Let $gcd_a = gcd(p_a, c_a)$ be the greatest common divisor of p_a and c_a. During the execution the number of tokens on arc a cannot go below

$M_0(a)$ *mod* gcd_a. The idea of useful tokens is that any arc a in the SDFG model could have its initial marking $M_0(a)$ rounded to the largest multiple of gcd_a smaller than itself without changing precedence constraints or schedulability of the SDFG. The useful tokens property is formally described in Lemma 1.

Lemma 1. [Marchetti and Munier-Kordon, 2009a] *Let* $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ *be an SDFG. Replacing, for every arc* $a \in \mathcal{A}, M_0(a)$ *by* $\lfloor M_0(a) \rfloor_{gcd_a}$*, where* $gcd_a = gcd(p_a, c_a)$ *and* $\lfloor x \rfloor_{gcd_a} = \lfloor \frac{x}{gcd_a} \rfloor \times gcd_a$*, does not change precedence constraints nor schedulability of the SDFG.*

Figure 2.14 illustrates an SDFG with $M_0(a) = 7$. Since the numbers of data items produced or consumed are multiples of $gcd_a = 3$, the initial marking can be rounded down to 6 without affecting the dataflow behavior.

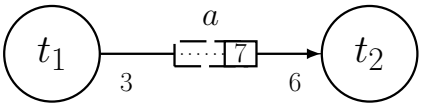


Figure 2.14 – An SDFG with $M_0(a) = 7$. By applying the useful tokens property, the initial marking of a can be reduced to $\lfloor M_0(a) \rfloor_3 = 6$ without effect on the precedence constraints.

2.4.3 Normalization and Z_i notation

Normalization is a transformation introduced in [Marchetti and Munier-Kordon, 2009a] to simplify the analysis of SDFG models. This transformation yields a normalized SDFG such that all consumption and production weights associated to a task are identical. As a result, the total number of tokens on each circuit of the normalized graph is invariant.

Consider a consistent SDFG and let $lcm_R = lcm(R_1, \cdots, R_{|\mathcal{T}|})$ be the least common multiple of the components of its repetition vector R. For each arc $a = (t_i, t_j)$, let $N_a = \frac{lcm_R}{R_i.p_a}$ (or equivalently $N_a = \frac{lcm_R}{R_j.c_a}$). Upon multiplying the weights (p_a, c_a and $M_0(a)$) of each arc a by N_a, the weights adjacent to each task are made equal and the graph is normalized. This transformation does not modify the precedence constraints.

The values N_a make up a vector $N = [N_{a_1}, \cdots, N_{a_{|\mathcal{A}|}}]$ called the normalization vector. The transformation is applicable only on consistent graphs and calls for simple computations. Note however that the components R_i of the repetition vector and their least common multiple may in some cases be large integers.

The normalized weight of task t_i (*resp.* t_j) with outgoing (*resp.* incoming) arc $a = (t_i, t_j)$ is $Z_i = p_a.N_a$ (*resp.* $Z_j = c_a.N_a$). Normalization is illustrated in Figure 2.15.

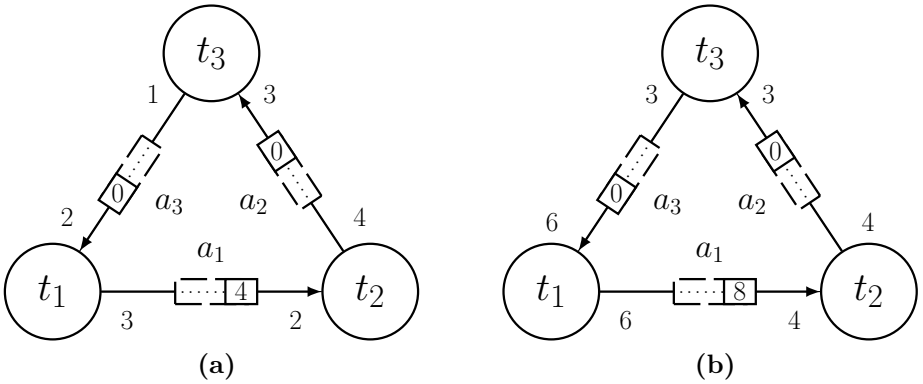


Figure 2.15 – (a) An SDFG with repetition vector $R = [2, 3, 4]$. (b) The equivalent normalized graph obtained with the normalization vector $N = [2, 1, 3]$.

Normalization is a reversible transformation; it is easily seen that dividing for each arc a each weight by N_a gives back the initial non-normalized graph. This implies that a consistent SDFG can be normalized without any loss of information. Normalization will be used to simplify multiple proofs and will also be applied in conjunction with a sufficient condition of liveness to compute a live initial marking.

2.4.4 Liveness

Checking liveness is a difficult problem widely studied in the dataflow community. The liveness property guarantees proper execution of the application modeled by a dataflow graph: a live marking ensures that each actor may be fired infinitely often hence as many times as required during a dataflow execution.

After recalling different techniques to check liveness for the SDFG model, a sufficient condition of liveness is presented in this section that will be exploited later on because of its low time complexity. At the end of the section an algorithm is given to compute a live marking for an SDFG model.

State of the art of the liveness problem for SDFG models

The complexity of liveness checking is still unknown for SDFG and more expressive models. However, checking liveness on an HSDFG model is done in polynomial time: remove every arc with non-zero initial marking, if the resulting graph is acyclic the HSDFG is live [Commoner et al., 1971].

For consistent SDFG models two equivalent techniques—in terms of complexity—exist to check liveness. A first technique consists in expanding the SDFG into an HSDFG and then applying the algorithm described in [Commoner et al., 1971]. A second technique consists in executing symbolically the SDFG until it reaches a cyclic pattern. Symbolic execution techniques are presented for the SDFG model in [Ghamarian et al., 2006a; Khasawneh, 2007]. The technique consists in attempting to execute all the tasks exactly as many times as indicated by their repetition

factor (to perform a system iteration). If it is possible to perform an iteration then it is possible to perform an infinite number of iterations and the SDFG is live.

Both expansion and symbolic execution techniques have exponential complexity; the expansion may give exponential size graphs while symbolic execution could imply an exponential number of computations.

Sufficient condition of liveness for SDFG models

One technique to get arround the complexity issue is to use a sufficient condition of liveness. A technique is proposed in [Marchetti and Munier-Kordon, 2009a] to identify a live marking in polynomial time. This sufficient condition of liveness (**SCL**) is given in Theorem 1. Since the condition is sufficient but not necessary, an initial marking may not respect the condition and yet be live.

Theorem 1 (**SCL**)**.** [Marchetti and Munier-Kordon, 2009a] Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized SDFG with only useful tokens. $\mathcal{G}_{sdf}$ is live if for every cycle μ in $\mathcal{G}_{sdf}$ the initial marking satisfies:

$$\sum_{a \in \mu} M_0(a) > \sum_{a \in \mu} (c_a - gcd_a),$$

with gcd_a the greatest common divisor of p_a and c_a.

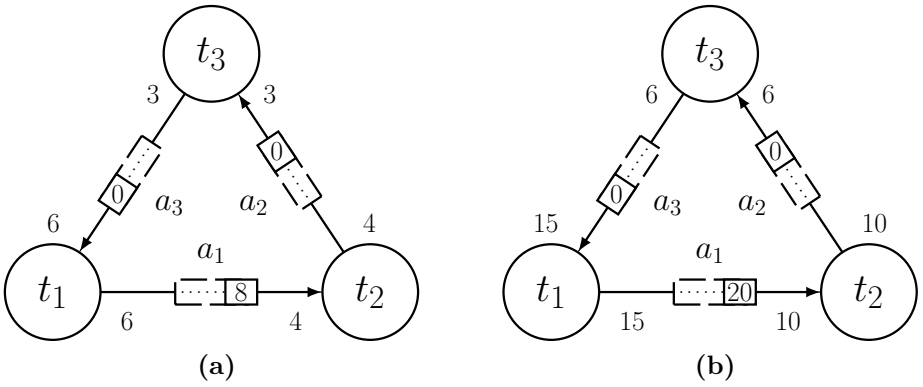


Figure 2.16 – (a) Normalized SDFG with an initial marking that is live according to Theorem 1. (b) Counter example showing that the sufficient condition of liveness of Theorem 1 is not necessary.

Figure 2.16(a) shows an SDFG with a cycle $\mu = (t_1, t_2, t_3)$ and with an initial marking live according to **SCL** since $\sum_{a \in \mu} M_0(a) = 8$ and $\sum_{a \in \mu}(c_a - gcd_a) = 7$. The SDFG of Figure 2.16(b), which does not respect the sufficient condition since $\sum_{a \in \mu} M_0(a) = 20$ and $\sum_{a \in \mu}(c_a - gcd_a) = 21$ is live since the execution sequence $t_2t_2t_3t_3t_3t_1t_2t_3t_3t_1$ bringing the system back to its initial state is feasible.

Liveness computation

This section presents a linear program deduced from Theorem 1 to compute a live marking on a normalized SDFG.

Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized SDFG. The mixed integer linear program 1 [Benabid-Najjar et al., 2012] expresses the condition **SCL**:

Mixed-Integer Linear Program 1: live SDFG (SCL)

$$\text{minimize} \quad \sum_{a \in \mathcal{A}} M_0(a)$$

$$\text{subject to} \begin{cases} \gamma_{t_j} - \gamma_{t_i} + M_0(a) - \varepsilon \geq c_a - gcd_a & \forall a = (t_i, t_j) \in \mathcal{A} \\ M_0(a) = gcd_a \cdot m_0(a) & \forall a \in \mathcal{A} \\ M_0(a) \in \mathbb{N}, m_0(a) \in \mathbb{N} & \forall a \in \mathcal{A} \\ \gamma_{t_i} \in \mathbb{R} & \forall t_i \in \mathcal{T} \\ \varepsilon \in \mathbb{R}^{*+} \text{ very small.} \end{cases}$$

There cannot be constraints with strict inequalities in a linear program. The introduction of the value ε transforms constraints with a strict inequality into non-strict inequality constraints. The value $\varepsilon \in \mathbb{R}^{*+}$ is chosen very small in order to have a minimal impact on the result. The first constraint thus differs slightly from the inequality of **SCL**.

The mixed-integer program 1 is not scalable since the integer part makes the problem require non polynomial-time solution techniques such as branch-and-bound. However, the problem can be solved approximately by relaxing the constraint that variables be integer and rounding the result up to the closest integer. Using this approximation technique gives a polynomial-time algorithm.

As the solution obtained by rounding to the closest integer is not compatible with the useful tokens assumption (see Section 2.4.2), resulting initial markings are rounded to the first larger multiple of gcd_a.

2.5 SDFG scheduling and throughput computation

Dataflow scheduling analysis has been the object of many studies. Scheduling an SDFG requires that it be consistent and its initial marking be live. These two conditions ensure that the marking will be bounded and deadlock will be avoided during an execution. Different performance issues can be analyzed in relation with scheduling: latency, throughput, memory consumption, Network on Chip (NoC) bottlenecks, etc.

There are three main types of static dataflow graph schedules: the ASAP schedule, the K-periodic schedule and the one-periodic schedule. The ASAP schedule has been introduced in Section 2.4.4 with the symbolic execution to verify liveness [Ghamarian et al., 2006b]. The K-periodic schedule introduced in [Bodin et al.,

2012] allows to compute a periodic-constrained optimal schedule with a better time complexity (still exponential). The one-periodic schedule (also called strictly periodic schedule) is a particular case of the K-periodic schedule and the minimum period one-periodic schedule can be computed with a polynomial time complexity [Benabid-Najjar et al., 2012].

This section recalls necessary notions for the scheduling problem and details the three types of dataflow scheduling methods: ASAP, one-periodic and K-periodic. It is divided into six subsections. The first explains the boundedness property. Section 2.5.2 defines a feasible schedule and formally describes precedence constraints. Section 2.5.3 defines the throughput and period associated to a schedule and the relation between them. Section 2.5.4 defines and characterizes the ASAP schedule. Section 2.5.5 details the one-periodic schedule. Section 2.5.6 defines the K-periodic schedule.

2.5.1 Boundedness

A dataflow graph is said to be bounded if, during its execution, its marking remains bounded. This property depends on the schedule employed.

Consider the single rate application of Figure 2.17 composed of two strongly connected components and an arc from the first to the second. If the schedule, for instance ASAP, gives the first strongly connected component a higher throughput than the second, the number of tokens on arc a will grow unbounded during the execution.

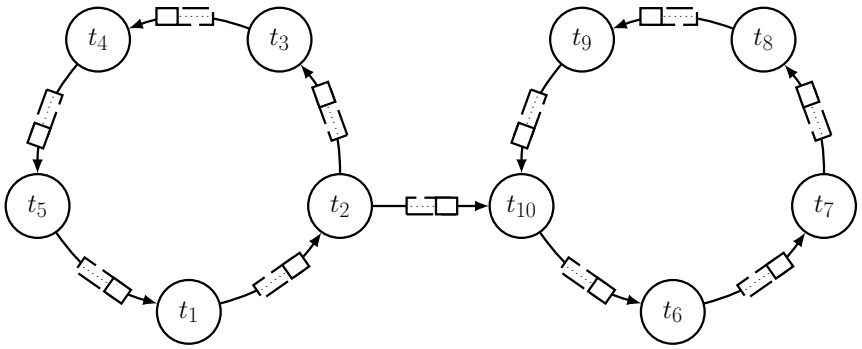


Figure 2.17 – A dataflow graph composed of two strongly connected components linked by arc $a = (t_2, t_{10})$.

One way to ensure boundedness, is to restrict scheduling to strongly connected graphs. This assumption is very restrictive since many applications modeled by dataflow graphs, for instance acyclic, do not respect it. This thesis proposes scheduling methods that ensure boundedness without this restriction.

2.5.2 Schedule and precedence constraints

Scheduling a dataflow graph consists in finding the starting time of each task according to a given strategy (ASAP, one-periodic, ...). All schedules must satisfy the precedence constraints imposed by the model, such that a task cannot begin to be executed before its inputs are available. Precedence constraints were formally described in terms of the consumption/production rates and the initial marking in [Marchetti and Munier-Kordon, 2009a] for the SDFG model. The first part of this section introduces the notion of feasible schedule, the second part formally describes precedence constraints.

Feasible schedule definition

For an SDFG, a schedule s is characterized by the starting times $s\langle t, n\rangle$ of the executions $\langle t, n\rangle$, with t a task of the SDFG and n the n^{th} execution of t. Note that n is unbounded.

A schedule s is feasible if all its executions fulfill the precedence constraints. Satisfaction of precedence constraints ensures that the computation of a task t_i cannot start before all the data items it requires are available.

The characterization of the precedence constraints between tasks for SDFG models can be expressed formally in terms of the positiveness of the markings as we shall see now.

SDFG precedence constraints

Consider an SDFG $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$. $M_0(a)$ is the initial marking of arc $a = (t_i, t_j) \in \mathcal{A}$ and $M(a)$ is the marking after several executions have occurred. After n_i executions of t_i and n_j executions of t_j,

$$M(a) = M_0(a) + n_i.p_a - n_j.c_a.$$

There is a precedence constraint between the executions $\langle t_i, n_i\rangle$ and $\langle t_j, n_j\rangle$ when the n_j^{th} execution of t_j cannot begin before the n_i^{th} execution of t_i. This may be expressed by writing that the marking $M(a)$ cannot be negative:

1. After the execution of $\langle t_i, n_i\rangle$, there are enough data items on arc a to execute $\langle t_j, n_j\rangle$ if:

$$M_0(a) + n_i.p_a - n_j.c_a \geq 0. \tag{2.1}$$

Redundant constraints do not need to be taken into account. This may be expressed in terms of the marking on arc a as follows:

2. Before the execution $\langle t_i, n_i\rangle$, there are enough data items to execute $\langle t_j, n_j - 1\rangle$

but not enough for the execution $\langle t_j, n_j \rangle$ if:

$$c_a > M_0(a) + p_a.(n_i - 1) - c_a.(n_j - 1) \geq 0. \tag{2.2}$$

Combining equations 2.1 and 2.2 gives the following lemma:

Lemma 2. [Marchetti and Munier-Kordon, 2009a] *Consider an arc $a = (t_i, t_j)$ of an SDFG $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$. There is a direct precedence constraint between executions $\langle t_i, n_i \rangle$ and $\langle t_j, n_j \rangle$ if:*

$$p_a > M_0(a) + p_a.n_i - c_a.n_j \geq max(p_a - c_a, 0).$$

As in the sequel we shall only consider direct, or non-redundant, precedence constraints, we shall drop the qualificatives direct and non-redundant.

For the SDFG depicted in Figure 2.18 there is a precedence constraint between $\langle t_1, 1 \rangle$ and $\langle t_2, 2 \rangle$. Indeed, the first execution of t_2 can be executed before the first execution of t_1 while the second execution of t_2 has to wait until the first execution of t_1 occurred. We check that the formula of Lemma 2 gives $3 > 2 + 3 - 4 \geq 1$, meaning that a precedence constraint exists between executions $\langle t_1, 1 \rangle$ and $\langle t_2, 2 \rangle$.

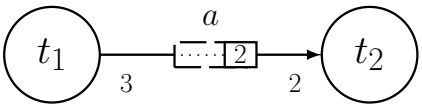


Figure 2.18 – An SDFG with a precedence constraint between the second execution of t_2 and the first execution of t_1.

Lemma 2 covers precedence constraints between tasks. It leaves the choice of following of imposing or not the constraint that the actors be sequential. Following [Lee and Messerschmitt, 1987; Marchetti and Munier-Kordon, 2009a] we impose that the actors be sequential, hence $\langle t_i, n \rangle$ directly precedes $\langle t_i, n + 1 \rangle$, without overlapping. Note that Lemma 2 is still valid with $t_i = t_j$, allowing sequential behavior of a task of an SDFG to be expressed with a self-loop.

2.5.3 Throughput and iteration period

The throughput of a task is the number of times the task is executed during a time unit. More precisely, the throughput of a task is defined in terms of a schedule as:

$$\lambda(t) = \lim_{n \to \infty} \frac{n}{s\langle t, n \rangle}.$$

For consistent live SDFG models, we define an iteration as an execution of the tasks as many times as indicated by the repetition vector. The iteration period T is the time interval between (the beginnings of) two consecutive iterations. The SDFG system throughput Λ is measured as the inverse of the iteration period $\Lambda = \frac{1}{T}$. Task

throughput is the product of the system throughput by the number of executions of the task in one iteration.

Worst case execution time

Schedules such as the one-periodic or the K-periodic schedule are computed at compile time. This means that the task execution times are evaluated before the application is executed. The dataflow community tends to use the Worst Case Execution Time (WCET) possible for each task. The main reason is that WCET gives functional guarantees.

Task execution time may be data dependent. By running tasks on representative data sets, their average and worst computation times as well as other statistics may be evaluated. Following a schedule obtained using the average computation time may prove unfeasible on specific data sets.

The only way to have a functional guarantee is to construct schedules based on WCET. Constructing a schedule using the WCETs gives a lower bound on throughput.

Throughput analysis

SDFG throughput analysis techniques are divided in two categories: they use either an expansion into an HSDFG, or the construction of a schedule with particular structure. Max-Plus algebra is used in [de Groote et al., 2012] to compute an expansion. This approach is similar to computing the expansion into an HSDFG, however the graph obtained is smaller and thus throughput analysis is faster. The throughput analysis presented in [Ghamarian et al., 2006b; Stuijk et al., 2006] avoids a transformation of the SDFG into a larger graph. The throughput is determined directly from the ASAP schedule. All the techniques described above are optimal in the sense that they give the maximal throughput reachable by the application assuming the task computation times are exactly known. Unfortunately, these techniques also have exponential complexity.

No polynomial technique has been found to evaluate maximum throughput for the SDFG model. One of the most powerful techniques is the K-periodic schedule which gives an optimal throughput evaluation in a reasonable time on instances up to thousand tasks if the components of the repetition vector are not too large [Bodin et al., 2012]. To reduce the time complexity, lower bound approximation techniques have been developed such as one-periodic analysis [Marchetti and Munier-Kordon, 2009b; Benabid-Najjar et al., 2012]. They allow throughput evaluation on graphs of up to ten thousand tasks.

2.5.4 ASAP scheduling

The ASAP schedule, which schedules a computation as soon as the necessary data is available, is the simplest way—from a conceptual point of view—to define a schedule for a dataflow graph. Moreover it has the advantage of being the fastest possible schedule. Functional simulation or symbolic execution provide ways of computing such a schedule. A self-timed implementation [Sriram and Bhattacharyya, 2009] follows the ASAP schedule, up to transfer delays between data production and consumption.

The formal characterization of the ASAP schedule is deduced from the precedence constraints described in subsection 2.5.2. The SDFG model definition and equation 2.1 give the following definition of the ASAP schedule:

Definition 1. *Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a SDFG and $s\langle t, n\rangle$ be the start time of the n^{th} execution of the task t according to schedule s. The schedule s is ASAP if*

- *$\forall t_i \in \mathcal{T}$, $s\langle t_i, n_i\rangle$ is minimum with*
  $$s\langle t_i, 1\rangle \geq 0$$
  and for every $n_i \in \mathbb{N}^*$,
  $$s\langle t_i, n_i + 1\rangle \geq s\langle t_i, n_i\rangle + \ell_i$$
- *and, $\forall a = (t_i, t_j) \in \mathcal{A}$,*
  $$s\langle t_j, n_j\rangle \geq s\langle t_i, n_i\rangle + \ell_i,$$
  for all precedence constraints between $\langle t_i, n_i\rangle$ and $\langle t_j, n_j\rangle$, where $n_j \in \mathbb{N}^*$.

The ASAP schedule exhibits two phases, a transient phase followed by a phase where the start times have a periodic behavior [Chrétienne, 1985; Cohen et al., 1985]. The transient phase is proven to be bounded in [Baccelli et al., 1992].

Figure 2.19 illustrates the ASAP schedule associated to a simple SDFG. The transient phase of the schedule reduces to one execution of t_3. The periodic phase follows, of which two periods are shown.

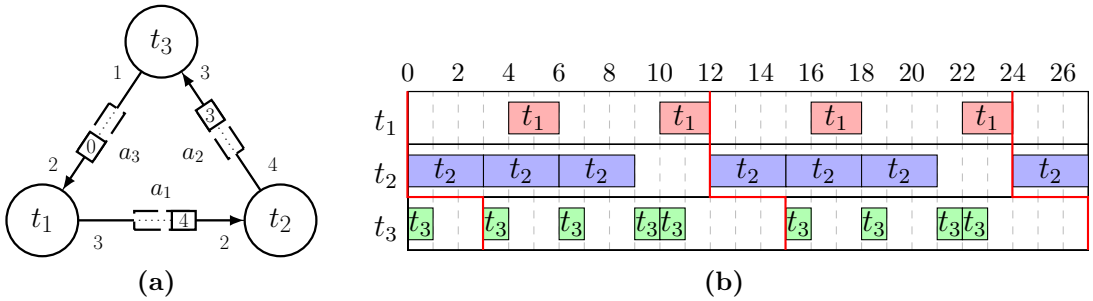


Figure 2.19 – (a) A live SDFG graph. (b) ASAP schedule of the SDFG graph. The transient phase of the ASAP schedule reduce to one execution of t_3 with start time $s\langle t_3, 1\rangle = 0$. The sequences repeated periodically are delimited by the red lines. Two periods are shown. The period is $T = 12$ and the system throughput is $\Lambda = \frac{1}{12}$.

Note that ASAP scheduling does not guarantee that the marking will be bounded in general, although guarantee is provided for strongly connected graphs or acyclic

graphs. Beside this restriction the drawback of this scheduling strategy is that techniques to compute a schedule, such as the functional simulation or symbolic execution, are done in exponential time.

2.5.5 One-periodic scheduling

The one-periodic schedule was introduced in [Marchetti and Munier-Kordon, 2009b] and [Benabid-Najjar et al., 2012] for the SDFG model. This section recalls the definition of a one-periodic schedule and presents a theorem to characterize it.

A one-periodic schedule is defined by its iteration period T and a first start time $s\langle t_i, 1\rangle$ for each task $t_i \in \mathcal{T}$. Each task t_i is executed at regular intervals. This time interval is the task's period and is denoted by w_i. It satisfies $w_i = \frac{T}{R_i}$.

More formally the definition of a one-periodic schedule for the SDFG model is:

Definition 2. *Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be an SDFG and $s\langle t, n\rangle$ be the start time of the n^{th} execution of task t according to schedule s. The schedule s is one-periodic of iteration period T if for every task $t_i \in \mathcal{T}$,*

$$s\langle t_i, n\rangle = s\langle t_i, 1\rangle + (n-1).w_i$$

where $w_i = \frac{T}{R_i} \geq \ell_i$ is the period of t_i and R_i is its repetition factor.

The following theorem [Benabid-Najjar et al., 2012] characterizes the existence of a one-periodic schedule for the SDFG model.

Theorem 2. [Benabid-Najjar et al., 2012] *Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized SDFG with an initial marking satisfying the useful tokens assumption. A schedule s is a feasible one-periodic schedule of iteration period T iff,*

- *for every task $t_i \in \mathcal{T}$, the initial starting time satisfies*
  $$s\langle t_i, 1\rangle \geq 0$$
  and the period $w_i = \frac{T}{R_i}$, where R_i is the repetition factor of t_i, verifies
  $$w_i \geq \ell_i,$$
- *and, for every arc $a = (t_i, t_j)$, the inequality*
  $$s\langle t_j, 1\rangle - s\langle t_i, 1\rangle \geq \ell_i + \frac{T}{Z_j.R_j}(Z_j - M_0(a) - gcd_a)$$
  where $gcd_a = gcd(Z_i, Z_j)$, holds.

The value $\frac{T}{Z_j.R_j}$ is independent of the task since $Z_i.R_i = Z_j.R_j,\ \forall t_i, t_j \in \mathcal{T}^2$.

Figure 2.20 illustrates a one-periodic schedule with iteration period $T = 18$, start times $s\langle t_1, 1\rangle = 7$, $s\langle t_2, 1\rangle = 0$, $s\langle t_3, 1\rangle = 1.5$ and task periods $w_1 = 9$, $w_2 = 6$, $w_3 = 4.5$.

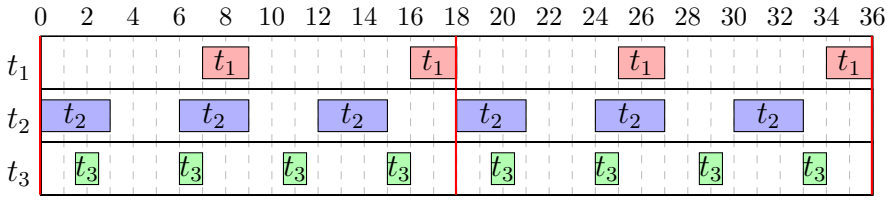


Figure 2.20 – One-periodic schedule for the SDFG of Figure 2.19(a), with repetition vector $R = [2, 3, 4]$. The iteration period is $T = 18$ and task periods are $w_1 = 9$, $w_2 = 6$ and $w_3 = 4.5$. Two iterations are shown

The one-periodic schedule executes each task t_i of the graph exactly R_i times during the iteration period T. The one-periodic schedule of $\mathcal{G}$ guarantees a bounded marking for any consistent SDFG $\mathcal{G}$, in contrast with the ASAP schedule, which requires additional assumptions such as strong connectivity or acyclic topology. Also, this schedule can be computed or verified in polynomial time by using linear programming.

2.5.6 K-Periodic scheduling

The K-periodic schedule is a generalization of the one-periodic schedule. Each task t_i has a K_i-periodic schedule, with time interval between the k^{th} execution of t_i and its $(k+K_i)^{th}$ execution equal to the time period w_i. The formula for the task period with the one-periodic schedule, $w_i = \frac{T}{R_i}$, where T is the iteration period, is now replaced by $\frac{w_i}{K_i} = \frac{T}{R_i}$, where K_i is the periodicity factor of task t_i.

K-periodic scheduling is formally defined in [Bodin et al., 2012] for the SDFG model. It is extended to the CSDFG model in [Bodin et al., 2016] and maximal throughput is shown to be attained when for every task t_i the periodicity factor K_i is a multiple of the repetition factor R_i. This result is also valid for SDFGs since an SDFG is a CSDFG with only one phase.

The characterization of a K-periodic schedule uses the precedence constraints from Lemma 2:

Definition 3. [Bodin et al., 2012] *Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be an SDFG with repetition vector $R = [R_1, \cdots, R_{|\mathcal{T}|}]$ and a fixed periodicity vector $K = [K_1, \cdots, K_{|\mathcal{T}|}]$. A schedule s is K-periodic of iteration period T if the start times of the tasks $t_i \in \mathcal{T}$ satisfy*

$$s\langle t_i, n_i.K_i + m_i\rangle = s\langle t_i, m_i\rangle + n_i.w_i,$$

for all $n_i \in \mathbb{N}^*$ and all $m_i \in \{1, \cdots, K_i\}$, where $w_i = K_i.\dfrac{T}{R_i} \geq K_i.\ell_i$ is the period of t_i.

A necessary and sufficient condition of existence of a K-periodic schedule follows.

Denote for each arc $a = (t_i, t_j) \in \mathcal{A}$,

$$\pi_a^{max}(m_i, m_j) = \lfloor M_0(a) - Z_i + gcd_a + Z_i.m_i - Z_j.m_j \rfloor_{gcd_{Ka}},$$

$$\pi_a^{min}(m_i, m_j) = \lceil M_0(a) - \max\{Z_i - Z_j, 0\} + Z_i.m_i - Z_j.m_j \rceil_{gcd_{Ka}},$$

with $gcd_{Ka} = gcd(K_i.Z_i, K_j.Z_j)$, $m_i \in \{1, \cdots, K_i\}$, $m_j \in \{1, \cdots, K_j\}$.

Theorem 3. [Bodin et al., 2012] *Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized SDFG. A K-periodic schedule of iteration period T and periodicity vector $K = [K_1, \cdots, K_{|\mathcal{T}|}]$ is feasible iff:*

- *for every task $t_i \in \mathcal{T}$, the initial starting time satisfies*
  $$s\langle t_i, 1\rangle \geq 0$$
  and the period $w_i = K_i.\frac{T}{R_i}$, where R_i is the repetition factor of t_i, verifies
  $$w_i \geq K_i.\ell_i,$$
- *and, for every arc $a = (t_i, t_j)$, the inequality*
  $$s\langle t_j, m_j\rangle - s\langle t_i, m_i\rangle = \ell_i - \frac{T}{Z_j.R_j}.\pi_a^{max}(m_i, m_j)$$
  $\forall (m_i, m_j) \in \{1, \cdots, K_i\} \times \{1, \cdots, K_j\}$ such as $\pi_a^{max}(m_i, m_j) \geq \pi_a^{min}(m_i, m_j)$, holds.

The relation between start times $s\langle t_i, n.K_i + m_i\rangle = s\langle t, m_i\rangle + n.w_i$ in Definition 3 expresses the fact that executions of task t_i separated by K_i in the sequence of task executions are separated in time by the period w_i. Figure 2.21 illustrates the K-periodic schedule of the SDFG pictured in Figure 2.19(a) with a periodicity vector $K = [2, 3, 4]$. The schedule satisfies the conditions $s\langle t_i, n.K_i + m_i\rangle = s\langle t_i, m_i\rangle + n.w_i$ of Definition 3. The task periods are $w_1 = w_2 = w_3 = 12$. For task t_1 and $n = 1$, we have $s\langle t_1, 3\rangle = s\langle t_1, 1\rangle + 12 = 16$ for $m_1 = 1$ and $s\langle t_1, 4\rangle = s\langle t_1, 2\rangle + 12 = 22$ for $m_1 = 2$. For task t_2 and $n = 1$, we have $s\langle t_2, 5\rangle = s\langle t_2, 2\rangle + 12 = 15$ for $m_2 = 2$.

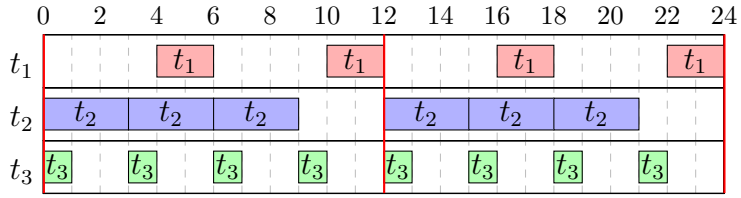


Figure 2.21 – The K-periodic schedule of the SDFG of Figure 2.19(a) with $K = [R_1, R_2, R_3] = [2, 3, 4]$, iteration period $T = 12$ and task periods $w_1 = w_2 = w_3 = 12$. K-periodic sequences are highlighted between the red lines; two system periods are shown.

The scheduling method is restricted to strongly connected SDFGs in [Bodin et al., 2012]. However, with $K_i = R_i$ for all tasks t_i of the graph, boundedness of the marking is ensured as for the one-periodic schedule. The time complexity of the proposed algorithms to find K-periodic schedules depends polynomially on the values of the vector K.

2.6 Conclusion

This chapter has introduced static dataflow models, with increasing expressiveness from the HSDFG to the PCG. Important notions were introduced in the context of the SDFG model: consistency, repetition vector, useful tokens, normalization and liveness. The remainder of the thesis will only address the SDFG, the CSDFG and the PCG models. All the notions introduced in this chapter will be extended to the CSDFG and PCG models. Three types of schedule were also introduced, ASAP, one-periodic and K-periodic. These schedules will be extended to the CSDFG model and the one-periodic schedule will be used for the mapping problem addressed in Chapter 5.

The next chapter presents the extension of these notions to the CSDFG and the PCG models. Extensions to the PCG model are a contribution of this thesis.

Chapter 3

CSDFG behavior and extension to the PCG model

Contents

3.1 Introduction

This chapter extends the notions presented for the SDFG model to the CSDFG and PCG models. The theoretical results thus obtained are used for the dataflow generator presented in Chapter 4.

Consistency, useful tokens, normalization and liveness for the CSDFG model are presented in Section 3.2. Section 3.3 details the ASAP, one-periodic and K-periodic schedule for the CSDFG model. Section 3.4 extends notions from Section 3.3 to the PCG model. Section 3.5 is the conclusion.

3.2 Cyclo-Static Dataflow Graph behavior

The CSDFG model, introduced in Section 2.2.4, is an extension of the SDFG model and many notions from the SDFG model carry over to this model.

Section 3.2.1 explains the notion of functional equivalence between the SDFG and the CSDFG model. Section 3.2.2 discusses the consistency of CSDFG models. Section 3.2.3 extends the notion of useful tokens to the CSDFG model. Section 3.2.4 extends normalization to the CSDFG model. Section 3.2.5 introduces some new notations for the CSDFG model. Finally, Section 3.2.6 extends the notion of liveness to the CSDFG model.

3.2.1 The functionally equivalent SDFG of a CSDFG

The notion of functionally equivalent SDFG has been introduced in [Bhattacharyya et al., 2000]. Let $\mathcal{G}^s = (\mathcal{T}^s, \mathcal{A}^s, \mathcal{P}^s, \mathcal{C}^s, \mathcal{M}^s, \mathcal{L}^s)$ be an SDFG and $\mathcal{G}^c = (\mathcal{T}^c, \mathcal{A}^c, \mathcal{P}^c, \mathcal{C}^c, \mathcal{M}^c, \mathcal{L}^c)$ be a CSDFG with $\mathcal{T}^s = \mathcal{T}^c$ and $\mathcal{A}^s = \mathcal{A}^c$. Two tasks, $t^c \in \mathcal{T}^c$ and the corresponding task $t^s \in \mathcal{T}^s$, are functionally equivalent if each consumption weight associated to an input arc $a^s = (\cdot, t^s)$ is the sum of the components of the consumption vector of the corresponding arc $a^c = (\cdot, t^c)$, each production weight associated to an output arc $a^s = (t^s, \cdot)$ is the sum of the components of the production vector of the corresponding arc $a^c = (t^c, \cdot)$ and the execution time of the task t^s is the sum of the components of the vector of execution times of t^c.

To start a task computation, the functionally equivalent SDFG must wait for all the data items needed for the computation phases of the corresponding CSDFG's task to be available. Similarly, the result of an SDFG's task computation are made available all at once at the end of the computation, instead of at the end of each phase computation as soon as the phase results are obtained. Figure 3.1 illustrates a CSDFG task and its functionally equivalent SDFG task.

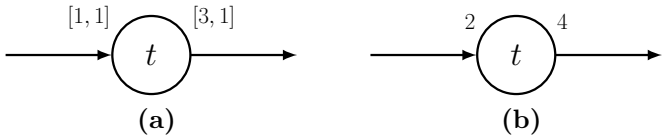


Figure 3.1 – (a) A CSDFG task; (b) its functionally equivalent SDFG task.

If every task of $\mathcal{G}^s$ is functionally equivalent to the corresponding task of $\mathcal{G}^c$ and if the initial markings of the two graphs are identical, then the SDFG $\mathcal{G}^s$ is functionally equivalent to the CSDFG $\mathcal{G}^c$.

We recall that p_{a^s} and c_{a^s} are the production and the consumption weights of arc $a^s \in \mathcal{A}^s$. Let $a^c = (t_i^c, t_j^c) \in \mathcal{A}^c$, p_{a^c} and c_{a^c} are defined for the CSDFG model such that $p_{a^c} = \sum_{k=1}^{\varphi_i} p_{a^c}(k)$ and $c_{a^c} = \sum_{k=1}^{\varphi_j} c_{a^c}(k)$.

If, for each arc $a^s \in \mathcal{A}^s$ and its topological equivalent arc $a^c \in \mathcal{A}^c$, $p_{a^s} = p_{a^c}$, $c_{a^s} = c_{a^c}$ and $M_0(a^s) = M_0(a^c)$ and, for each task t_i^s and its topological equivalent t_i^c, $\ell_i^s = \sum_{k=1}^{\varphi_i} \ell_i^c(k)$, then $\mathcal{G}^s$ is functionally equivalent to $\mathcal{G}^c$. Note that if its functionally equivalent SDFG is live, then a CSDFG is live, however, the converse is not true. Functional equivalence is illustrated in Figure 3.2.

3.2.2 Consistency and repetition vector

The SDFG notion of consistency is extended to the CSDFG model in [Bilsen et al., 1995]. If its functionally equivalent SDFG is consistent, then a CSDFG is consistent.

Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}^c, \mathcal{C}^c, \mathcal{M}, \mathcal{L}^c)$ be a CSDFG and $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}^s, \mathcal{C}^s, \mathcal{M}, \mathcal{L}^s)$ be its functionally equivalent SDFG. The topology matrix Γ of the CSDFG is identical to the topology matrix of its functionally equivalent SDFG, presented the Section 2.4.1. The matrix Γ, of size $|\mathcal{A}| \times |\mathcal{T}|$, has elements:

$$\Gamma_{a,i} = \begin{cases} p_a & \text{if } a = (t_i, \cdot) \\ -c_a & \text{if } a = (\cdot, t_i) \\ 0 & \text{otherwise} \end{cases}.$$

The repetition vector R of a consistent CSDFG is computed using the calues p_a and c_a, $a \in \mathcal{A}$, thus, the repetition vectors associated to $\mathcal{G}_{sdf}$ and $\mathcal{G}_{csdf}$ are identical. Like the SDFG model, the CSDFG model verifies $\forall a = (t_i, t_j) \in \mathcal{A}$, $p_a.R_i = c_a.R_j$.

Figure 3.2 shows a CSDFG and its functionally equivalent SDFG. As determined for the SDFG in Section 2.4.1, their repetition vector is $R = [2, 3, 4]$.

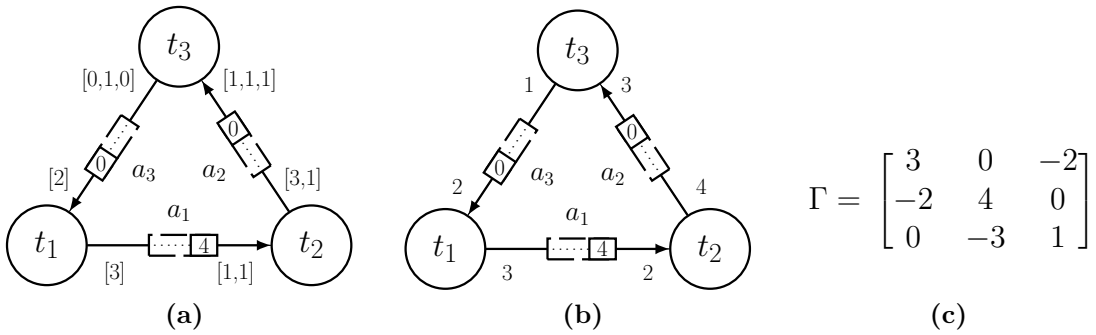


Figure 3.2 – (a) A CSDFG, (b) the functionally equivalent SDFG, and (c) their topology matrix, with associated repetition vector $R = [2, 3, 4]$.

3.2.3 Useful tokens

The notion of useful tokens introduced in Section 2.4.2 for the SDFG model is extended here to the CSDFG model. The notion must be adapted to the CSDFG model since it expresses data transfers with a finer granularity than the SDFG model. This has been done in [Benazouz et al., 2010; Stuijk et al., 2008].

Lemma 3. [Benazouz et al., 2010] *Let* $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ *be a CSDFG. Replacing* $M_0(a)$ *by* $\lfloor M_0(a) \rfloor_{step_a}$ *for every arc* $a = (t_i, t_j) \in \mathcal{A}$*, with* $step_a = gcd(p_a(1), \cdots, p_a(\varphi_i), c_a(1), \cdots, c_a(\varphi_j))$ *and* $\lfloor x \rfloor_{step_a} = \lfloor \frac{x}{step_a} \rfloor \times step_a$*, does not change the precedence constraints of the CSDFG.*

While for the SDFG model the useful tokens where obtained by rounding down to a multiple of gcd_a (Section 2.4.2), they are obtained by rounding down to a multiple of $step_a$ for the CSDFG model. Figure 3.3 illustrates the notion of useful tokens for the CSDFG model.

Computation of the number of useful tokens for the CSDFG of Figure 3.3(a) using Lemma 3 gives the same result as for its functionally equivalent SDFG since $step_a = gcd_a = 3$. In contrast, for the CSDFG of figure 3.3(b), all tokens are useful since $step_a = 1$.

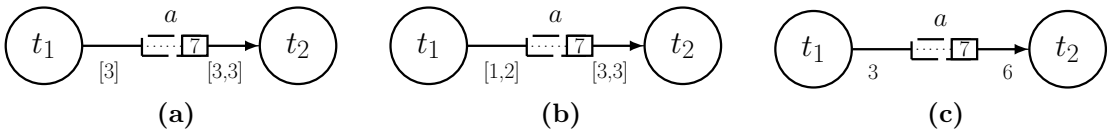


Figure 3.3 – Two CSDFGs with the same functionally equivalent SDFG. (a) CSDFG with 6 useful tokens; (b) CSDFG with 7 useful tokens; (c) functionally equivalent SDFG with 6 useful tokens.

3.2.4 Normalization

The normalization procedure introduced for the SDFG model in Section 2.4.3 and extended to the CSDFG model in [Benazouz et al., 2013] is now detailed. The normalization of a CSDFG is performed by multiplying the consumption and production vectors by the same scalars as the corresponding consumption and production rates in the functionally equivalent SDFG. Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}^c, \mathcal{C}^c, \mathcal{M}, \mathcal{L}^c)$ be a CSDFG and $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}^s, \mathcal{C}^s, \mathcal{M}, \mathcal{L}^s)$ be its functionally equivalent SDFG. The normalization vector $N \in (\mathbb{N}^*)^{|\mathcal{A}|}$ introduced for the SDFG in Section 2.4.3 is the same for $\mathcal{G}_{csdf}$. The value N_a is thus obtained by using the same formula for both models $N_a = \frac{lcm(R_1,\cdots,R_{|\mathcal{T}|})}{R_i.p_a}$ with $a = (t_i, \cdot)$ or $N_a = \frac{lcm(R_1,\cdots,R_{|\mathcal{T}|})}{R_i.c_a}$ with $a = (\cdot, t_i)$.

Figure 3.4 illustrates the normalization of a CSDFG.

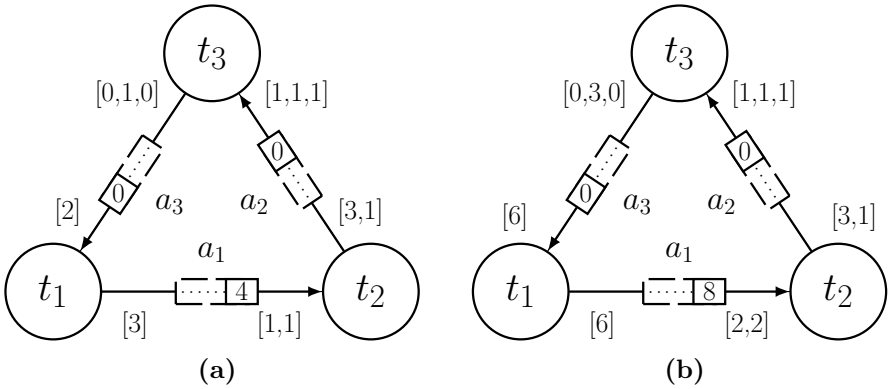


Figure 3.4 – (a) A CSDFG; (b) the equivalent normalized CSDFG. The normalization vector is $N = [2, 1, 3]$.

3.2.5 Notations P_a and C_a

This section introduces new notations for the CSDFG model required for the liveness notion. Consider a CSDFG $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$. The n^{th} execution of the k^{th} phase of task t_i is denoted by $\langle t_i(k), n\rangle$, with $n \in \mathbb{N}^*$. If $n = 1$, k belongs to $\{1, \cdots, \varphi_i\}$. $\langle t_i(k), n\rangle^{-1}$ is the phase execution preceding $\langle t_i(k), n\rangle$. It is formally defined as:

$$\langle t_i(k), n\rangle^{-1} = \begin{cases} \langle t_i(k-1), n\rangle & \text{if } k > 1 \text{ and } n > 1 \\ \langle t_i(\varphi_i), n-1\rangle & \text{if } k = 1 \text{ and } n > 1 \\ \langle t_i(0), 1\rangle & \text{if } k = 1 \text{ and } n = 1 \end{cases}$$

Execution $\langle t(0), 1\rangle$ is fictitious and precedes $\langle t(1), 1\rangle$.

Consider an execution $\langle t_i(k), n\rangle$ of task t_i, $P_a(k, n)$ denotes the total number of data items produced by t_i on $a = (t_i, \cdot)$ from its first phase to the end of $\langle t_i(k), n\rangle$. The cumulative production $P_a(k, n)$ satisfies the recurrence equation

$$P_a(k, n) = P_a^{-1}(k, n) + p_a(k) \text{ with } P_a(0, 1) = 0,$$

where $P_a^{-1}(k,n)$ denotes the cumulative production of t_i on a from the first phase to the end of $\langle t_i(k), n\rangle^{-1}$.

Consider for instance arc $a = (t_2, t_3)$ in Figure 3.5: $\varphi_1 = 2$, $P_a(1,1) = 3$, $P_a(2,2) = 8$ and $P_a^{-1}(2,2) = 7$. For any positive integer n, $P_a(2,n) = 4n$. Thus, for any execution $\langle t_i(k), n\rangle$ of t, $P_a(k,n) = P_a(k,1) + 4(n-1)$.

Now consider an execution $\langle t_i(k), n\rangle$ of task t_i. Denote by $C_a(k,n)$ (resp. $C_a^{-1}(k,n)$) the total number of data items consumed by t_i from $a = (\cdot, t_i)$ until the end of $\langle t_i(k), n\rangle$ (resp. $\langle t_i(k), n\rangle^{-1}$). The cumulative consumption $C_a(k,n)$ satisfies the recurrence equation

$$C_a(k,n) = C_a^{-1}(k,n) + c_a(k) \text{ with } C_a(0,1) = 0.$$

Pursuing with the example of Figure 3.5, $\varphi_2 = 3$, $C_a(1,1) = 1$, $C_a(3,2) = 6$ and $C_a^{-1}(3,2) = 5$. For any positive integer n, $C_a(3,n) = 3n$. Thus, for any execution $\langle t_2(k), n\rangle$ of t_2, $C_a(k,n) = C_a(k,1) + 3(n-1)$.

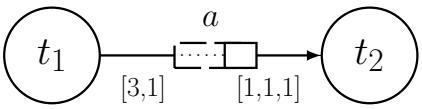


Figure 3.5 – A CSDFG with $P_a(1,1) = 3$, $P_a(2,2) = 8$, $P_a^{-1}(2,2) = 7$, $C_a(1,1) = 1$, $C_a(3,2) = 6$ and $C_a^{-1}(3,2) = 5$.

3.2.6 Liveness

In this section, the notion of liveness and the sufficient condition of liveness introduced for the SDFG model in Section 2.4.4 are extended to the CSDFG model.

Like for an SDFG, a live marking for a CSDFG ensures that each task may be executed infinitely often, hence as many times as required by the application modeled by the CSDFG.

Many notions and analysis methods introduced for the SDFG model have been extended to the CSDFG model. A symbolic execution technique for the CSDFG model is presented in [Chakilam and O'Neil, 2009]. Expansion techniques are also extended in [Bilsen et al., 1995] and [Sriram and Bhattacharyya, 2009].

The first part of this section is dedicated to the extension of the sufficient condition of liveness of Section 2.4.4 to the CSDFG model. The second part of the section presents algorithms to compute a live marking on a normalized CSDFG.

Sufficient condition of liveness

The sufficient condition of liveness has been extended to the CSDFG model in [Benazouz et al., 2013]. Two equivalent versions have been proposed, called sufficient condition of liveness 1 (**SCL1**) and sufficient condition of liveness 2 (**SCL2**).

SCL1 is given in Theorem 4 while **SCL2** is given in Theorem 5. Let $step_a = gcd(p_a(1), \cdots, p_a(\varphi_i), c_a(1), \cdots, c_a(\varphi_j))$ for $a = (t_i, t_j)$.

Theorem 4 (SCL1). [Benazouz et al., 2013] Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized CSDFG. $\mathcal{G}_{csdf}$ is live if for every cycle $\mu = (t_1, a_1, \cdots, t_m, a_m)$ in $\mathcal{G}_{csdf}$ and for all $k_i \in \{1, \cdots \varphi_i\}$ and $k_{i+1} \in \{1, \cdots \varphi_{i+1}\}$,

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} \left[C_{a_i}(k_{i+1}, 1) - P_{a_i}^{-1}(k_i, 1) \right] - \sum_{i=1}^{m} step_{a_i},$$

with $k_{m+1} = k_1$.

Theorem 5 (SCL2). [Benazouz et al., 2013] Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized CSDFG. $\mathcal{G}_{csdf}$ is live if for every cycle $\mu = (t_1, a_1, \cdots, t_m, a_m)$ in $\mathcal{G}_{csdf}$,

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} \max_{k_i \in \{1, \cdots, \varphi_i\}} \left[C_{a_{i-1}}(k_i, 1) - P_{a_i}^{-1}(k_i, 1) \right] - \sum_{i=1}^{m} step_{a_i}$$

with $a_0 = a_m$.

SCL1 and **SCL2** are shown to be equivalent in [Benazouz et al., 2013]. Despite their similarity, **SCL1** and **SCL2** are used in different contexts. **SCL1** requires to test for each arc all phase combinations between its two tasks, this condition is appropriate for CSDFGs composed of tasks with few phases. **SCL2** calls for testing, for each task, every combination between input and output arcs within a cycle while avoiding to test every phase of each task. This can be problematic for graphs with a high average degree but is appropriate for graphs with few cycles.

To illustrate **SCL2** consider the CSDFG of Figure 3.6. Since

$$\begin{aligned} \max_{k_i \in \{1, \cdots \varphi_i\}} [C_{a_{i-1}}(k_i, 1) - P_{a_i}^{-1}(k_i, 1)] - step_{a_i} \quad &= 1 \quad \text{between arc } a_1 \text{ and } a_2 \\ &= -1 \quad \text{between arc } a_2 \text{ and } a_3 \\ &= 4 \quad \text{between arc } a_3 \text{ and } a_1 \end{aligned}$$

the right-hand side in **SCL2** is

$$\sum_{i=1}^{m} \max_{k_i \in \{1, \cdots \varphi_i\}} [C_{a_{i-1}}(k_i, 1) - P_{a_i}^{-1}(k_i, 1)] - \sum_{i=1}^{m} step_{a_i} = 4,$$

and the initial marking, such that $\sum_{i=1}^{m} M_0(a_i) = 5$, is live.

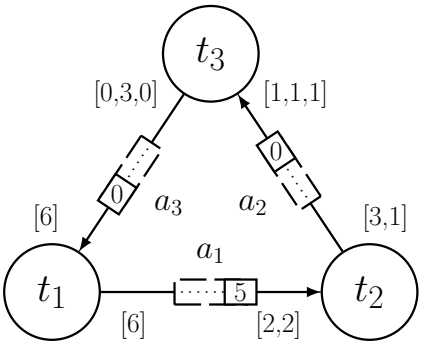


Figure 3.6 – Normalized CSDFG with a live initial marking according to theorems **SCL1** and **SCL2**

Liveness computation

Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized CSDFG model. We now describe two algorithms derived from **SCL1** and **SCL2**. Let $W_a^{csdf}(k_i, k_j) = C_a(k_j, 1) - P_a^{-1}(k_i, 1) - step_a$. The Mixed-Integer Linear Program 2 expresses the sufficient condition of liveness of Theorem 4:

Mixed-Integer Linear Program 2: liveness of CSDFG (SCL1)

$$\text{minimize} \quad \sum_{a \in \mathcal{A}} M_0(a)$$

$$\text{subject to} \begin{cases} \gamma_{t_j,k_j} - \gamma_{t_i,k_i} + M_0(a) - \varepsilon \geq W_a^{csdf}(k_i, k_j) & \forall a = (t_i, t_j) \in \mathcal{A}, \\ & \forall k_i \in \{1, \cdots \varphi_i\}, \\ & \forall k_j \in \{1, \cdots \varphi_j\} \\ M_0(a) = step_a \cdot m_0(a) & \forall a \in \mathcal{A} \\ \gamma_{t_i} \in \mathbb{R} & \forall t_i \in \mathcal{T} \\ M_0(a) \in \mathbb{N}, m_0(a) \in \mathbb{N} & \forall a \in \mathcal{A} \\ \varepsilon \in \mathbb{R}^{*+} \text{ very small.} \end{cases}$$

Introduction of ε is used to make a strict inequality non-strict, so that the program be linear.

We denote by $\mathcal{D}$ the set of pairs of arcs $a_i = (\cdot, t_e)$ and $a_j = (t_e, \cdot)$ where $t_e \in \mathcal{T}$. Let $W_{a_i,a_j}^{csdf} = \max_{k \in \{1, \cdots \varphi_e\}} \left[C_{a_i}(k, 1) - P_{a_j}^{-1}(k, 1) - step_{a_j} \right]$ with $(a_i, a_j) \in \mathcal{D}$ and t_e the task between a_i and a_j. The Mixed-Integer Linear Program 3 expresses the sufficient condition of liveness of Theorem 5:

Mixed-Integer Linear Program 3: liveness of CSDFG (SCL2)

$$\text{minimize} \quad \sum_{a \in \mathcal{A}} M_0(a)$$

$$\text{subject to} \begin{cases} \gamma_{a_j} - \gamma_{a_i} + M_0(a_j) - \varepsilon \geq W_{a_i,a_j}^{csdf} & \forall (a_i, a_j) \in \mathcal{D} \\ M_0(a) = step_a \cdot m_0(a) & \forall a \in \mathcal{A} \\ M_0(a) \in \mathbb{N}, m_0(a) \in \mathbb{N}, \gamma_a \in \mathbb{R} & \forall a \in \mathcal{A} \\ \varepsilon \in \mathbb{R}^{*+} \text{ very small.} \end{cases}$$

3.3 CSDFG scheduling and throughput computation

This section extends to the CSDFG model the characterization of precedence constraints, the notion of throughput and the three schedules introduced in Section 2.5 for the SDFG model. Section 3.3.1 extends the characterization of the precedence constraints to the CSDFG model. Throughput is discussed in Section 3.3.2. Sections 3.3.3, 3.3.4 and 3.3.5 extend respectively the ASAP, the one-periodic and the K-periodic schedule to the CSDFG model.

3.3.1 Precedence constraints

The characterisation of a set of non-redundant precedence constraints for the SDFG model has been extended to the CSDFG model in [Benazouz et al., 2010]. Consider a CSDFG model $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$. Let $M(a)$ be the marking of arc $a = (t_i, t_j) \in \mathcal{A}$ after task t_i was executed $n_i.\varphi_i + k_i$ times (n_i full cyclic phases followed by k_i phases) and after task t_j was executed $n_j.\varphi_j + k_j$ times. More formally:

$$M(a) = M_0(a) + P_a(k_i, n_i) - C_a(k_j, n_j)$$

The extension of the characterization of the precedence constraints to the CSDFG model gives:

1. After execution $\langle t_i(k_i), n_i \rangle$ the marking allows execution $\langle t_j(k_j), n_j \rangle$:

$$M_0(a) + P_a(k_i, n_i) - C_a(k_j, n_j) \geq 0. \tag{3.1}$$

2. Before execution $\langle t_i(k_i), n_i \rangle$, the marking allows execution $\langle t_j(k_j), n_j \rangle^{-1}$ but not execution $\langle t_j(k_j), n_j \rangle$:

$$c_a(k_j) > M_0(a) + P_a^{-1}(k_i, n_i) - C_a^{-1}(k_j, n_j) \geq 0. \tag{3.2}$$

Combining equations 3.1 and 3.2 gives the following lemma:

Lemma 4. [Benazouz et al., 2010] Consider the CSDFG $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ and an arbitrary arc $a = (t_i, t_j) \in \mathcal{A}$. There is a precedence constraint between execution $\langle t_i(k_i), n_i \rangle$ and $\langle t_j(k_j), n_j \rangle$ if:

$$p_a(k_i) > M_0(a) + P_a(k_i, n_i) - C_a(k_j, n_j) \geq \max(0, p_a(k_i) - c_a(k_j))$$

Figure 3.7 depicts a CSDFG model with a precedence constraint between execu-

tions $\langle t_2(2), 1\rangle$ and $\langle t_1(1), 1\rangle$. Indeed, application of Lemma 4 yields:

$$\begin{aligned} p_a(1) &= 1 \\ M_0(a) + P_a(1,1) - C_a(2,1) &= 1 + 1 - 2 = 0 \\ \max(p_a(1) - c_a(2), 0) &= \max(1 - 1, 0) = 0 \end{aligned}$$

and since the values satisfy $1 > 0 \geq 0$, the model does have that precedence constraint.

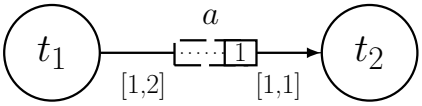


Figure 3.7 – A CSDFG with a precedence constraint between executions $\langle t_2(2), 1\rangle$ and $\langle t_1(1), 1\rangle$.

3.3.2 Throughput and iteration period

Throughput is a measure of performance of a schedule. The formal definition of task throughput naturally extends the definition introduced for the SDFG model:

$$\lambda(t_i) = \lim_{n\to\infty} \frac{n}{s\langle t_i(\varphi_i), n\rangle}.$$

The execution of a CSDFG task amounts to executing all its phases once. As for a SDFG, we define an iteration of a CSDFG as an execution of its tasks as many times as indicated by the repetition vector. The iteration period of a consistent live CSDFG is the interval of time between the beginnings of two consecutive iterations.

To compute the maximum throughput of a CSDFG a common technique consists in performing a symbolic execution, a technique that has been extended to the CSDFG model in [Stuijk et al., 2008]. Expansion techniques, based on the transformation of an SDFG into the equivalent HSDFG followed by the computation of its maximum mean cycle time, are extended to CSDFG by using the functionally equivalent SDFG. As for the SDFG model, both techniques are exact but inefficient, with non-polynomial computation time.

3.3.3 ASAP scheduling

We now describe the ASAP schedule for a CSDFG. The definition formally expresses the property that each task is executed as soon as possible, as permitted by the precedence constraints and task execution times:

Definition 4. *Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a CSDFG and $s\langle t(k), n\rangle$ be the start time of the n^{th} execution of the k^{th} phase of the task t according to schedule s. The schedule s is ASAP if*

- $\forall t_i \in \mathcal{T}$, $\forall k_i \in \{1, \cdots, \varphi_i\}$, $s\langle t_i(k_i), n_i\rangle$ *is minimum with*
  $s\langle t_i(1), 1\rangle \geq 0$,
  for every $k_i \in \{1, \cdots, \varphi_i - 1\}$,
  $s\langle t_i(k_i + 1), n_i\rangle \geq s\langle t_i(k_i), n_i\rangle + \ell_i(k_i)$
  and, for every $n_i \in \mathbb{N}^*$,
  $s\langle t_i(1), n_i + 1\rangle \geq s\langle t_i(\varphi_i), n_i\rangle + \ell_i(\varphi_i)$
- *and,* $\forall a = (t_i, t_j) \in \mathcal{A}$,
  $s\langle t_j(k_j), n_j\rangle \geq s\langle t_i(k_i), n_i\rangle + \ell_i(k_i)$,
  for all precedence constraints between $\langle t_i(k_i), n_i\rangle$ *and* $\langle t_j(k_j), n_j\rangle$, *where* $n_j \in \mathbb{N}^*$.

Figure 3.8 shows the ASAP schedule for a simple CSDFG. The transient phase of the schedule is composed of the executions $\langle t_3(1), 1\rangle$, $\langle t_3(2), 1\rangle$ and $\langle t_3(3), 1\rangle$, starting at times 0, 0.33 and 2, respectively. The periodic phase of the schedule follows; two system iterations are shown. The iteration period is 9, the maximum system throughput is therefore $\Lambda = \frac{1}{9}$.

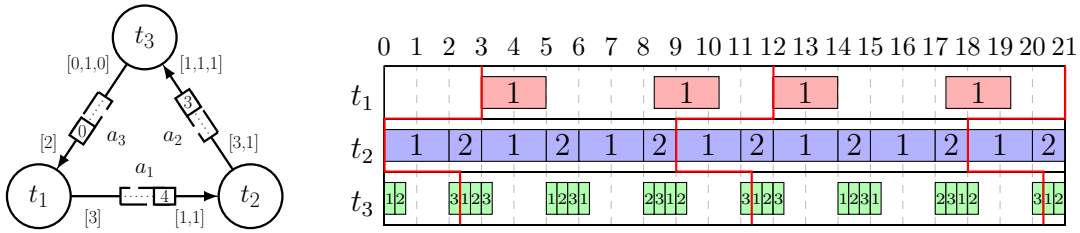


Figure 3.8 – (a) Live CSDFG with $R = [2, 3, 4]$. (b) Associated ASAP schedule for $\ell_1(1) = 2$, $\ell_2(1) = 2$, $\ell_2(2) = 1$ and $\ell_3(1) = \ell_3(2) = \ell_3(3) = 0.33$. The transient phase of the schedule is composed of three executions of t_3 with start times $s\langle t_3(1), 1\rangle = 0$, $s\langle t_3(2), 1\rangle = 0.33$ and $s\langle t_3(3), 1\rangle = 2$. System iterations are delimited by the red lines; two periods are shown.

3.3.4 One-Periodic scheduling

The one-periodic schedule has been extended to the CSDFG model in [Bodin et al., 2013]. The formal definition of the one-periodic schedule for the CSDFG model closely resembles that of the SDFG model:

Definition 5. *Let* $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ *be a CSDFG model,* s *a schedule and* $\langle t_i(k), n\rangle$ *the* n^{th} *execution of phase* $k \in \{1, \cdots, \varphi_i\}$ *of task* t_i. *The schedule* s *is one-periodic if for every phase* k *of every task* $t_i \in \mathcal{T}$ *the start time of the execution* $t_i(k)$ *satisfies*

$$s\langle t_i(k), n\rangle = s\langle t_i(k), 1\rangle + (n-1).w_i$$

with $w_i = \dfrac{T}{R_i} \geq \displaystyle\sum_{k=1}^{\varphi_i} \ell_i(k)$ *the period of task* t_i.

Theorem 2 recalled in Section 2.5.5 has been extended in [Bodin et al., 2013] under the form of Theorem 6. Theorem 6 is a necessary and sufficient condition of existence of a one-periodic schedule for a CSDFG. We denote $gcd_a = gcd(p_a, c_a)$ and:

$$\alpha_a^{max}(k_i, k_j) = \lceil M_0(a) + P_a(k_i, 1) - C_a(k_j, 1) - max\{0, p_a(k_i) - c_a(k_j)\} \rceil_{gcd_a},$$

$$\alpha_a^{min}(k_i, k_j) = \lfloor M_0(a) + P_a^{-1}(k_i, 1) - C_a(k_j, 1) + 1 \rfloor_{gcd_a}.$$

Theorem 6. [Bodin et al., 2013] *Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized CSDFG with an initial marking satisfying the useful tokens assumption. A schedule s is a feasible one-periodic schedule of iteration period T iff,*

- *for every task $t_i \in \mathcal{T}$, the initial starting time satisfies*
  $$s\langle t_i(1), 1\rangle \geq 0$$
  and the period $w_i = \dfrac{T}{R_i}$, where R_i is the repetition factor of t_i, verifies
  $$w_i \geq \sum_{k=1}^{\varphi_i} \ell_i(k)$$
- *and, for every arc $a = (t_i, t_j)$ and every phase k_i and k_j, the inequality*
  $$s\langle t_j(k_j), 1\rangle - s\langle t_i(k_i), 1\rangle \geq \ell_i(k_i) - \frac{T}{Z_j.R_j}.\alpha_a^{max}(k_i, k_j)$$
  with $\alpha_a^{min}(k_i, k_j) \leq \alpha_a^{max}(k_i, k_j)$ holds.

Figure 3.9 illustrates a one-periodic schedule with iteration period $T = 9$ and start times $s\langle t_1(1), 1\rangle = 3.5$, $s\langle t_2(1), 1\rangle = 0$, $s\langle t_2(2), 1\rangle = 2$, $s\langle t_1(1), 1\rangle = 1.5$, $s\langle t_1(2), 1\rangle = 2.25$, $s\langle t_1(3), 1\rangle = 2$. The task periods are $w_1 = 9$, $w_2 = 6$, $w_3 = 4.5$. The system throughput is $\Lambda = \frac{1}{9}$. Note that this is the same throughput as for the ASAP schedule illustrated in Figure 3.8, however, generally, the best one-periodic schedule does not guarantee a maximum throughput.

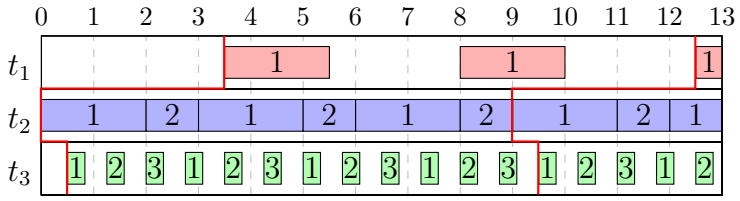


Figure 3.9 – One-periodic schedule for the CSDFG of Figure 3.8(a). Since the repetition vector of the graph is $R = [2, 3, 4]$ and the iteration period is $T = 9$, the tasks have periods $w_1 = 4.5$, $w_2 = 3$ and $w_3 = 2.25$. The red lines delimit the first iteration.

3.3.5 K-Periodic scheduling

The K-periodic schedule was presented for the SDFG model in Section 2.5.6. It has been extended to the CSDFG model in [Bodin et al., 2016]. In this paper, a K-periodic schedule with periodicity vector $K = [R_1, \cdots, R_{|\mathcal{T}|}]$ was proven to give

a schedule with optimal throughput.

We recall that the number of interleaved one-periodic sequences of a task t_i is called its periodicity factor and is denoted K_i. The period of task t_i, denoted w_i, is the time interval between its n^{th} and $(n+K_i)^{th}$ execution. Thus $w_i = T.\frac{K_i}{R_i}$.

We now define a K-periodic schedule for the CSDFG model:

Definition 6. [Bodin et al., 2016] *Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a CSDFG model and $K = [K_1, \cdots, K_{|\mathcal{T}|}]$ with $K_i \in \mathbb{N}^*$ be a periodicity vector . A schedule s is K-periodic of iteration period T if periods $w_i = T.\frac{K_i}{R_i} \geq \sum_{k=1}^{\varphi_i} \ell_i(k)$ exist for all $t_i \in \mathcal{T}$, such that*

$$s\langle t_i(k), n.K_i + m_i\rangle = s\langle t_i(k), m_i\rangle + n.w_i$$

for every $k \in \{1, \cdots, \varphi_i\}$, every $m_i \in \{1, \cdots, K_i\}$, and every $n \in \mathbb{N}^*$.

To present the theorem allowing to verify the feasibility of a K-periodic schedule, we need some additional notations. Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a CSDFG, $K = [K_1, \cdots, K_{|\mathcal{T}|}]$ be a periodicity vector and $\mathcal{G}'_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}', \mathcal{C}', \mathcal{M}, \mathcal{L}')$ be the CSDFG equivalent to $\mathcal{G}_{csdf}$, with the same structure and with production and consumption vectors duplicated K_i times for every $t_i \in \mathcal{T}$. $\mathcal{G}'_{csdf}$ is said to be K-equivalent CSDFG to $\mathcal{G}_{csdf}$.

For example for $a = (t_1, t_2) \in \mathcal{A}$, $K_1 = 2$ and $K_2 = 3$, the vectors $[p_a(1), \cdots, p_a(\varphi_1)]$, $[c_a(1), \cdots, c_a(\varphi_2)]$ become in model $\mathcal{G}'_{csdf}$ $[p_a(1), \cdots, p_a(\varphi_1), p_a(1), \cdots, p_a(\varphi_1)]$ and $[c_a(1), \cdots, c_a(\varphi_2), c_a(1), \cdots, c_a(\varphi_2), c_a(1), \cdots, c_a(\varphi_2)]$. This duplication is also applied to the vectors of execution times.

Figure 3.10 depicts a CSDFG and its K-equivalent for $K = R$. The notation $[x]^y$ represents a vector with x repeated y times. For instance $[1]^6 = [1, 1, 1, 1, 1, 1]$.

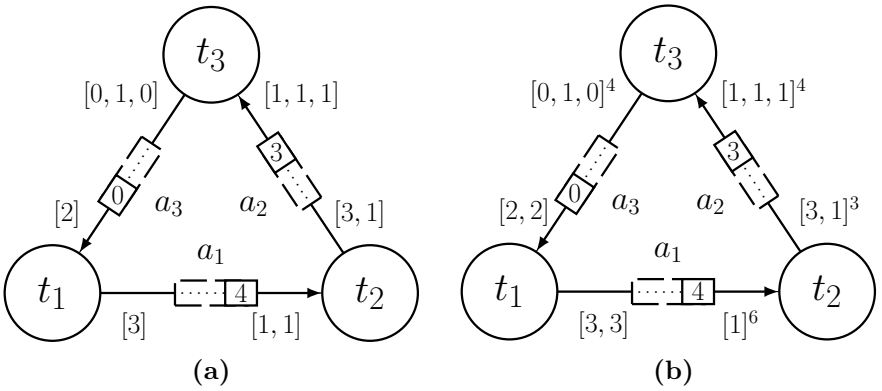


Figure 3.10 – (a) CSDFG with $R = [2, 3, 4]$; (b) K-equivalent CSDFG for $K = [R_1, R_2, R_3] = [2, 3, 4]$.

The notation $\alpha_a^{min}(k_i, k_j)$ and $\alpha_a^{max}(k_i, k_j)$ introduced in the previous section is used for the next theorem. The idea behind this theorem is to exploit Theorem 6 and characterize a K-periodic schedule of a CSDFG in terms of the one-periodic

schedule of the K-equivalent CSDFG:

Theorem 7. [Bodin et al., 2016] *Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized CSDFG with a periodicity vector $K = [K_1, \cdots, K_{|\mathcal{T}|}]$ and $\mathcal{G}'_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}', \mathcal{C}', \mathcal{M}, \mathcal{L}')$ be the K-equivalent CSDFG. A feasible K-periodic schedule s for $\mathcal{G}_{csdf}$ of iteration period T is the same as a one-periodic schedule of iteration period T for its K-equivalent $\mathcal{G}'_{csdf}$ satisfying the conditions expressed in Theorem 6*

Figure 3.11 illustrates the K-periodic schedule of the CSDFG of Figure 3.10(a) with $K = [2, 3, 4]$ and with an iteration period $T = 9$. The K-periodic schedule is identical to the one-periodic schedule of the K-equivalent CSDFG model of Figure 3.10(b). The task periods w_i, $t_i \in \mathcal{T}$, are all equal to the iteration period T since $K = [R_1, R_2, R_3]$. This schedule also gives the maximum system throughput $\Lambda = \frac{1}{9}$, and thus attains the throughput of the ASAP schedule.

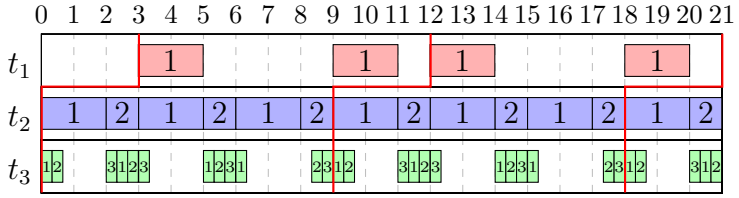


Figure 3.11 – K-periodic schedule for the CSDFG model of Figure 3.10(a) with iteration period $T = 9$. The task periods are $w_1 = w_2 = w_3 = 9$, since $K = [R_1, R_2, R_3] = [2, 3, 4]$. The first two iterations are delimited by the red lines.

3.4 Extension to the Phased Computation Graph model

This section presents a contribution of this thesis. It extends many notions from the CSDFG model to the PCG model. While the PCG model is not as commonly used as the SDFG or the CSDFG model, it is being used by the company Kalray to test their Massively Parallel Processor Array (MPPA).

Consistency and normalization are discussed in the next section. Section 3.4.2 extends to the PCG model the notation P_a, C_a from Section 3.2.5. The characterization of non-redundant precedence constraints is extended to the PCG model in Section 3.4.3. The useful tokens property is extended to the PCG model in Section 3.4.4 and is justified by proving that leaving out non-useful tokens does not impact the precedence constraints. Finally, the sufficient conditions **SCL1** and **SCL2** of Section 3.2.6 are extended to the PCG model in Section 3.4.5.

3.4.1 Consistency and normalization

This section extends consistency and normalization to the PCG model. We first recall some notations previously defined for the PCG in Section 2.2.7. Let $\mathcal{G}_{pcg} =$

$(\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ be a PCG model, and $p_a(k)$ and $c_a(k)$ the production and consumption weights of the phase k on arc a. A task t_i has $\sigma_i + \varphi_i$ phases numbered from $1-\sigma_i$ to φ_i. Phases numbered from $1-\sigma_i$ to 0 are the initial phases while phases numbered from 1 to φ_i are the cyclical phases. The threshold of a consumption phase k, denoted $\theta_a(k)$, satisfies $\theta_a(k) \geq c_a(k)$.

Consistency and normalization are easily extended to the PCG model, without considering initial phases nor thresholds. Denoting $p_a = \sum_{k=1}^{\varphi_i} p_a(k)$ and $c_a = \sum_{k=1}^{\varphi_i} c_a(k)$, the topology matrix and the computation of the normalization vector $N \in (\mathbb{N}^*)^{|\mathcal{A}|}$ for the PCG model are the same as for the CSDFG model.

Figure 3.12 illustrates the normalization of a PCG model. All weights associated to an arc a are multiplied by N_a during normalization, including thresholds and initial phase weights.

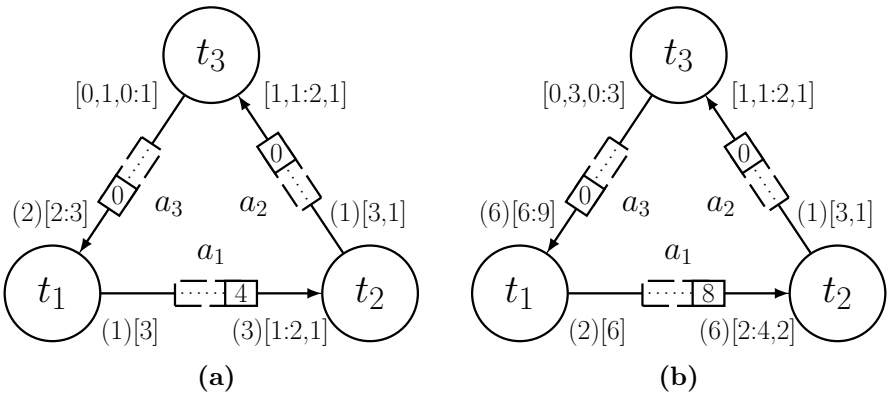


Figure 3.12 – (a) PCG model; (b) equivalent normalized PCG model. The normalization vector is $N = [2, 1, 3]$.

3.4.2 Notation extensions

The notations introduced in Section 3.2.5 for the CSDFG model are now extended to the PCG model.

Execution and predecessor notations

Consider a PCG $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$. A task $t_i \in \mathcal{T}$ has σ_i initial phases and φ_i cyclic phases. The n^{th} execution of phase k of task t_i is denoted by $\langle t_i(k), n\rangle$, where n is a positive integer. If $n = 1$, k belongs to $\delta_t = \{1-\sigma_i, \cdots, 0\} \cup \{1, \cdots, \varphi_i\}$ and the non-positive values of k relate to the initial phases of t_i. Otherwise, $n > 1$ and $k \in \{1, \cdots, \varphi_i\}$ relate to its cyclical normal phases.

The phase execution preceding $\langle t_i(k), n\rangle$, denoted $\langle t_i(k), n\rangle^{-1}$ is formally defined

as:

$$\langle t_i(k), n\rangle^{-1} = \begin{cases} \langle t_i(k-1), n\rangle & \text{if } k > 1 \text{ and } n > 1 \\ \langle t_i(\varphi_i), n-1\rangle & \text{if } k = 1 \text{ and } n > 1 \\ \langle t_i(k-1), 1\rangle & \text{if } k \geq 1 - \sigma_i \text{ and } n = 1 \end{cases} \tag{3.3}$$

Execution $\langle t(-\sigma_i), 1\rangle$ is fictitious and precedes $\langle t(1-\sigma_i), 1\rangle$.

The phase execution preceding $\langle t_i(k), n\rangle^{-1}$ is denoted $\langle t_i(k), n\rangle^{-2}$ and is determined by applying 3.3 recursively and noting that executions $\langle t_i(k), 1\rangle$ with $k \leq -\sigma_i$ are fictitious.

Notations c_a, p_a and θ_a

The execution notation is carried over to the notations c_a, p_a and θ_a. To the execution $\langle t(k), n\rangle$ are associated the weights $c_a(k,n) = c_a(k)$, $p_a(k,n) = p_a(k)$ and the threshold $\theta_a(k,n) = \theta_a(k)$. The weights and threshold associated to the execution $\langle t(k), n\rangle^{-1}$ preceding $\langle t(k), n\rangle$ are accordingly denoted $c_a^{-1}(k,n)$, $p_a^{-1}(k,n)$ and $\theta_a^{-1}(k,n)$.

To illustrate the notation, consider the PCG model of Figure 3.13. The production rates p_a satisfy:

- $p_a(0,0) = 1$, $p_a(1,0) = 3$ and $p_a(1,2) = 3$,
- $p_a^{-1}(0,0) = 0$ as it is fictitious, $p_a^{-1}(1,0) = p_a(0,0) = 1$ and $p_a^{-1}(1,2) = p_{a_1}(1,1) = 3$,
- $p_a^{-2}(0,0) = 0$ and $p_a^{-2}(1,0) = 0$ as they are fictitious, and $p_a^{-2}(1,2) = p_{a_1}(1,0) = 3$.

Similarly, the consumption rates c_a satisfy:

- $c_a(0,0) = 3$, $c_a(1,0) = 1$ and $c_a(2,2) = 1$,
- $c_a^{-1}(0,0) = 0$ as it is fictitious, $c_a^{-1}(1,0) = c_a(0,0) = 3$ and $c_a^{-1}(2,2) = c_a(1,2) = 1$,
- $c_a^{-2}(0,0) = 0$ and $c_a^{-2}(1,0) = 0$ as they are fictitious, and $c_a^{-2}(2,2) = c_a(2,1) = 1$.

Finally we consider the thresholds θ_a. Remember that $\theta_a(k) \geq c_a(k),\ \forall k \in \delta(t)$ and that $\theta_a(k) = c_a(k)$ when $\theta_a(k)$ is not explicitly defined:

- $\theta_a(0,0) = 3$, $\theta_a(1,0) = 2$ and $\theta_a(2,2) = 1$,
- $\theta_a^{-1}(0,0) = 0$ as it is fictitious, $\theta_a^{-1}(1,0) = \theta_a(0,0) = 3$ and $\theta_a^{-1}(2,2) = \theta_a(1,2) = 2$,
- $\theta_a^{-2}(0,0) = 0$ and $\theta_a^{-2}(1,0) = 0$ as they are fictitious, and $\theta_a^{-2}(2,2) = \theta_a(2,1) = 1$.

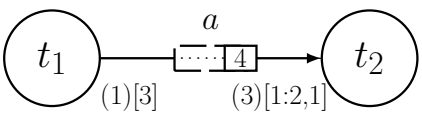


Figure 3.13 – PCG model with a producer and a consumer task.

Notation P_a

Consider execution $\langle t_i(k), n\rangle$ of phase k in the n^{th} execution of task t_i and denote by $P_a(k,n)$ the total number of data items produced by t_i on a from phase $1-\sigma_i$ of its n^{th} execution to the end of phase k.

The cumulative production $P_a(k,n)$ satisfies the recurrence equation

$$P_a(k,n) = P_a^{-1}(k,n) + p_a(k)$$

with $P_a(-\sigma_i, 1) = 0$.

Denoting by p_a the cumulative production $p_a(\varphi_i)$ of one execution of the cyclic phases for any execution $\langle t_i(k), n\rangle$ with $k \in \delta_i$, the cumulative production $P_a(k,n)$ may be expressed as

$$P_a(k,n) = P_a(k,1) + p_a.(n-1). \tag{3.4}$$

For instance, for $a = (t_1, t_2)$ in Figure 3.14, $\varphi_1 = 2$, $\sigma_1 = 1$, $p_a = 4$, $P_a(0,1) = 1$, $P_a(1,1) = 4$ and $P_a(2,1) = 5$. For any positive integer n and any execution $\langle t_1(k), n\rangle$ of t_1, $P_a(k,n) = P_a(k,1) + 4(n-1)$.

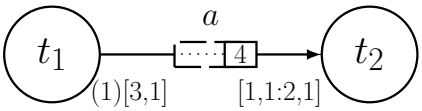


Figure 3.14 – Example producer-consumer PCG model.

Lemma 5 is a technical result about the total number of data items produced on an arc.

Lemma 5. *The cumulated production P_a on an arc $a = (t_i, \cdot) \in \mathcal{A}$ satisfies, for any execution $\langle t_i(k), n\rangle$ of t_i, with n a positive integer,*

$$P_a^{-1}(k,n) = P_a^{-1}(k,1) + p_a.(n-1).$$

Proof. Three cases must be considered:

- If $k > 1$ and $n > 1$, then according to equation 3.3 $\langle t_i(k), 1\rangle^{-1} = \langle t_i(k-1), 1\rangle$ and $\langle t_i(k), n\rangle^{-1} = \langle t_i(k-1), n\rangle$. The expression follows from equation 3.4.
- If $k = 1$ and $n > 1$, equation 3.3 implies $\langle t_i(1), 1\rangle^{-1} = \langle t_i(0), 1\rangle$ and $\langle t_i(1), n\rangle^{-1} = \langle t_i(\varphi_i), n-1\rangle$.
  - if $\sigma_i = 0$ (no initial phase) then $P_a(0,1) = 0$,
  - if $\sigma_i > 0$ (σ_i initial phase) then $P_a(0,1) = \sum_{k=1-\sigma_i}^{0} p_a(k)$.

  Thus, using equation 3.4,

$$P_a^{-1}(1,n) = P_a(\varphi_i, n-1) = P_a(0,1) + p_a.(n-1).$$

- Lastly, if $k \geq 1 - \sigma_i$ and $n = 1$, then $\langle t_i(k), 1\rangle^{-1} = \langle t_i(k-1), 1\rangle$ and the equation is true. ■

Notation C_a

Consider execution $\langle t_i(k), n\rangle$ of task t_i and denote by $C_a(k, n)$ the total number of data items consumed by t from a from phase $1 - \sigma_i$ of its n^{th} execution to the end of phase k. The cumulative consumption $C_a(k, n)$ satisfies the recurrence equation

$$C_a(k, n) = C_a^{-1}(k, n) + c_a(k) \tag{3.5}$$

with $C_a(-\sigma_i, 1) = 0$.

Denoting by c_a the cumulative consumption $c_a(\varphi_i)$ of one execution of the cyclic phases, for any execution $\langle t_i(k), n\rangle$ with $k \in \delta_i$ the cumulative consumption $C_a(k, n)$ may be expressed as

$$C_a(k, n) = C_a(k, 1) + c_a.(n - 1). \tag{3.6}$$

Continuing with the example of Figure 3.14, $\varphi_2 = 3$, $\sigma_2 = 0$, $c_a = 3$, $C_a(1, 1) = 1$, $C_a(2, 1) = 2$ and $C_a(3, 1) = 3$. For any positive integer n and any execution $\langle t_2(k), n\rangle$ of t_2, $C_a(k, n) = C_a(k, 1) + 3(n - 1)$.

Lemma 6 is the analogue for consumption weights of Lemma 5. Its proof is therefore omitted.

Lemma 6. *The cumulative consumption C_a from arc $a = (\cdot, t_i) \in \mathcal{A}$ satisfies, for any execution $\langle t_i(k), n\rangle$ of t_i with n a positive integer,*

$$C_a^{-1}(k, n) = C_a^{-1}(k, 1) + c_a.(n - 1).$$

3.4.3 Precedence constraints

Consider a PCG $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$. The marking $M(a)$ on arc $a = (t_i, t_j)$ after the execution of n_i cycles followed by k_i phases of t_i and n_j cycles followed by k_j phases of t_j satisfies:

$$M(a) = M_0(a) + P_a(k_i, n_i) - C_a(k_j, n_j).$$

Extending the characterization of the precedence constraints from the CSDFG model to the PCG model we can state that execution $\langle t_i(k_i), n_i\rangle$ (directly) precedes execution $\langle t_j(k_j), n_j\rangle$ if the following two conditions are met:

1. After execution $\langle t_i(k_i), n_i\rangle$ the marking is sufficient for execution $\langle t_j(k_j), n_j\rangle$:

$$M_0(a) + P_a(k_i, n_i) - C_a^{-1}(k_j, n_j) \geq \theta_a(k_j). \tag{3.7}$$

2. Before execution $\langle t_i(k_i), n_i\rangle$ the marking is insufficient for execution $\langle t_j(k_j), n_j\rangle$

but sufficient for the execution of the phase before that of $\langle t_j(k_j), n_j \rangle$:

$$\theta_a(k_j) > M_0(a) + P_a^{-1}(k_i, n_i) - C_a^{-2}(k_j, n_j) - \theta_a^{-1}(k_j, n_j) \geq 0. \tag{3.8}$$

Combining these inequalities gives the following lemma:

Lemma 7. *Consider a PCG model* $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ *and an arc* $a = (t_i, t_j) \in \mathcal{A}$*. There is a precedence constraint between execution* $\langle t_i(k_i), n_i \rangle$ *and* $\langle t_j(k_j), n_j \rangle$ *if:*

$$\begin{aligned}\theta_a(k_j) + \theta_a^{-1}(k_j, n_j) > M_0(a) + P_a^{-1}(k_i, n_i) - C_a^{-2}(k_j, n_j) \\ \geq \max(\theta_a^{-1}(k_j, n_j), \theta_a(k_j) - p_a(k_i) + c_a^{-1}(k_j, n_j))\end{aligned}$$

Proof. On one hand equation 3.7 is equivalent to:

$$M_0(a) + P_a^{-1}(k_i, n_i) - C_a^{-1}(k_j, n_j) \geq \theta_a(k_j) - p_a(k_i)$$

which can be rewritten as

$$M_0(a) + P_a^{-1}(k_i, n_i) - C_a^{-2}(k_j, n_j) \geq \theta_a(k_j) - p_a(k_i) + c_a^{-1}(k_j, n_j).$$

On the other hand equation 3.8 is equivalent to

$$\theta_a(k_j) + \theta_a^{-1}(k_j, n_j) > M_0(a) + P_a^{-1}(k_i, n_i) - C_a^{-2}(k_j, n_j) \geq \theta_a^{-1}(k_j, n_j).$$

Combining these rewritings of equations 3.7 and 3.8 gives the lemma. ■

Figure 3.15 illustrates a precedence constraint between executions $\langle t_1(1), 1 \rangle$ and $\langle t_2(2), 1 \rangle$ as the execution $\langle t_2(2), 1 \rangle$ requires two items because of the threshold and the availability of just one item. Applying the formula of Lemma 7:

$$\begin{aligned}\theta_a(2) + \theta_a^{-1}(2, 1) &= 2 + 1 = 3 \\ M_0(a) + P_a^{-1}(1, 1) - C_a^{-2}(2, 1) &= 1 + 1 - 0 = 2 \\ \max(\theta_a^{-1}(2, 1), \theta_a(2) - p_a(1) + c_a^{-1}(2, 1)) &= \max(1, 2 - 3 + 1) = 1\end{aligned}$$

and, since $3 > 2 \geq 1$, the existence of the precedence constraint is confirmed.

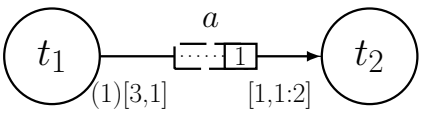


Figure 3.15 – PCG with a precedence constraint between executions $\langle t_1(1), 1 \rangle$ and $\langle t_2(2), 1 \rangle$.

3.4.4 Useful tokens

We now extend the useful tokens property to the PCG model. As seen in Section 2.4.2 this property, introduced in [Marchetti and Munier-Kordon, 2009a] for the

SDFG model, was extended to the CSDFG model in [Benazouz et al., 2010] (Section 3.3.1).

The property states that the residual number of tokens, which cannot be further reduced on each arc, does not affect the precedence constraints. For the CSDFG model this meant that the initial marking $M_0(a)$ of an arc $a = (t_i, t_j)$ could be rounded down to $\lfloor M_0(a) \rfloor_{step_a}$ with $step_a = \gcd(p_a(1), \cdots, p_a(\varphi_i), c_a(1), \cdots, c_a(\varphi_j))$. The formula for the PCG model is the same except that $step_a$ now takes also the initial phases and thresholds into account. Thus $step_a = \gcd(p_a(1-\sigma_i), \cdots, p_a(\varphi_i), c_a(1-\sigma_j), \cdots, c_a(\varphi_j), \theta_a(1-\sigma_j), \cdots, \theta_a(\varphi_j))$.

Lemma 8 formalizes the useful tokens property for the PCG model:

Lemma 8. *Let $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ be a PCG model and $\mathcal{G}^{\star}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ be the PCG model obtained by replacing, for every arc $a = (t_i, t_j) \in \mathcal{A}$, the initial marking $M_0(a)$ by*

$$M^{\star}_0(a) = \lfloor M_0(a) \rfloor_{step_a}$$

with $step_a = gcd(p_a(1-\sigma_i), \cdots, p_a(\varphi_i), c_a(1-\sigma_j), \cdots, c_a(\varphi_j), \theta_a(1-\sigma_j), \cdots, \theta_a(\varphi_j))$. The models $\mathcal{G}_{pcg}$ and $\mathcal{G}^{\star}_{pcg}$ have the same set of precedence constraints.

Proof. According to the definition of $step_a$,

$$\begin{aligned}
&c_a(k_j) + p_a(k_i) - C_a^{-1}(k_j, n_j),\\
&P_a(k_i, n_i) - C_a^{-1}(k_j, n_j) \text{ and}\\
&\max\left(\theta(k_j), \theta^{-1}(k_j, n_j) + p_a(k_i) - C_a^{-1}(k_j, n_j)\right)
\end{aligned}$$

are multiples of $step_a$. The operation $\lfloor M_0(a) \rfloor_{step_a}$ reduces the initial marking to $M^{\star}_0(a) = M_0(a) - \Delta$ with $\Delta < step_a$, in other words $M_0(a) = M^{\star}_0(a) + \Delta$ with $M^{\star}_0(a) = q.step_a$ where q is the quotient in the euclidean division of $M_0(a)$ by $step_a$ and Δ is the remainder.

Thus, the equation for precedence constraints has the form $a.step_a > b.step.a + q.step_a + \Delta \geq c.step_a$ with a, b, c integers and $\Delta < step_a$. When replacing $M_0(a)$ by $M^{\star}_0(a)$ we still have $a.step_a > b.step.a + q.step_a \geq c.step_a$, thus showing that the precedence constraints of $\mathcal{G}_{pcg}$ and $\mathcal{G}^{\star}_{pcg}$ are identical. ■

Figure 3.16 illustrates the useful tokens property on a PCG. We have $step_a = 3$, thus the initial marking can be rounded down to $M_0(a) = 3$ without interfering on the precedence constraints of the model.

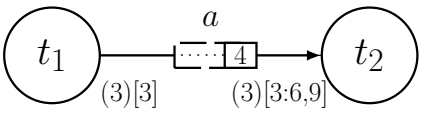


Figure 3.16 – A PCG with $step_a = 3$. The useful tokens property implies that $M_0(a) = 4$ may be replaced by $M_0(a)^{\star} = \lfloor 4 \rfloor_3 = 3$, with no effect on the precedence constraints.

3.4.5 Extension of the sufficient condition of liveness

This section extends to the PCG model the two sufficient conditions of liveness of the CSDFG model, **SCL1** and **SCL2**, expressed by Theorems 4 and 5 in Section 3.2.6.

The first part extends the sufficient condition **SCL1**. The second part extends the sufficient condition **SCL2**. The third part proves the equivalence between **SCL1** and **SCL2**.

Sufficient Condition 1 for the PCG model

At the core of the proof is the characterization of a deadlock. A cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ of a PCG $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ is said to be blocked if there exists a set of executions $E = \{\langle t_i(k_i), n_i\rangle, t_i \in \mu\}$ and a sequence of task executions allowing all the executions $\langle t_i(k_i), n_i\rangle^{-1}$, $t_i \in \mu$, such that no execution from E can be fired. The initial marking of $\mathcal{G}_{pcg}$ is live if no cycle may be blocked.

Since $\mathcal{G}_{pcg}$ is normalized, the total number of tokens remains constant in every cycle. The following lemma provides an upper bound on the total number of tokens in a blocked cycle. We recall that $\delta_i = \{1 - \sigma_i, \cdots, 0\} \cup \{1, \cdots, \varphi_i\}$.

Lemma 9. *Let $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ be a normalized PCG. If a cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ is blocked, then there exists $(k_1, \cdots, k_m) \in \delta_1 \times \cdots \times \delta_m$ such that*

$$\sum_{i=1}^{m} M_0(a_i) + \sum_{i=1}^{m} \left[P_{a_i}^{-1}(k_i, 1) - C_{a_{i-1}}^{-1}(k_i, 1) - \theta_{a_{i-1}}(k_i)\right] \leq -\sum_{i=1}^{m} step_{a_i}$$

where the index values are taken modulo m (if $i = 1$, $i - 1 = m$).

Proof. Suppose that cycle μ is blocked, then there exists a sequence of task executions such that each actor t_i, $i \in \{1, \cdots, m\}$, has reached execution $\langle t_i(k_i), n_i\rangle^{-1}$ but cannot execute $\langle t_i(k_i), n_i\rangle$. Thus, according to equation 3.7, for any arc $a_i = (t_i, t_{i+1})$ of μ, with $t_{m+1} = t_1$,

$$M_0(a_i) + P_{a_i}^{-1}(k_i, n_i) - C_{a_i}^{-1}(k_{i+1}, n_{i+1}) - \theta_{a_i}(k_{i+1}) < 0.$$

Now it follows from Lemmas 5 and 6 that

$$P_{a_i}^{-1}(k_i, n_i) = P_{a_i}^{-1}(k_i, 1) + p_a.(n_i - 1) \text{ and}$$

$$C_{a_i}^{-1}(k_{i+1}, n_{i+1}) = C_{a_i}^{-1}(k_{i+1}, 1) + c_a.(n_{i+1} - 1)$$

so that the above inequality may be rewritten as

$$M_0(a_i) + P_{a_i}^{-1}(k_i, 1) + p_a.(n_i - 1) - C_{a_i}^{-1}(k_{i+1}, 1) - c_a.(n_{i+1} - 1) - \theta_{a_i}(k_{i+1}) < 0.$$

By Lemma 8, the initial marking is supposed to be divisible by $step_a$, and so are all

the terms of the last inequality. This inequality may be rewritten as

$$M_0(a_i)+P_{a_i}^{-1}(k_i,1)+p_a.(n_i-1)-C_{a_i}^{-1}(k_{i+1},1)-c_a.(n_{i+1}-1)-\theta_{a_i}(k_{i+1}) \leq -step_{a_i}.$$

As $\mathcal{G}_{pcg}$ is normalized, $p_a.(n_i-1) = c_a.(n_{i+1}-1)$. Summing the previous inequalities yields

$$\sum_{i=1}^{m} M_0(a_i) + \sum_{i=1}^{m} \left[P_{a_i}^{-1}(k_i,1) - C_{a_{i-1}}^{-1}(k_i,1) - \theta_{a_{i-1}}(k_i)\right] \leq -\sum_{i=1}^{m} step_{a_i},$$

the result which was to be proven. ■

The next theorem is based on Lemma 9 and expresses a first general sufficient condition of liveness of a PCG.

Theorem 8 (SCL1). *A normalized PCG $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ is live if for every cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ and for every m-uple $(k_1, \cdots, k_m) \in \delta_1 \times \cdots \times \delta_m$:*

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} W_{a_i}^{pcg}(k_i, k_{i+1}),$$

where indexes are taken modulo m and $W_{a_i}^{pcg}(k_i, k_{i+1}) = C_{a_i}^{-1}(k_{i+1}, 1) + \theta_{a_i}(k_{i+1}) - P_{a_i}^{-1}(k_i, 1) - step_{a_i}$.

Proof. By taking the contraposition of Lemma 9, if we suppose that for any m-uple $(k_1, \cdots, k_m) \in \delta_1 \times \cdots \times \delta_m$ of any cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ of $\mathcal{G}_{pcg}$, the following inequality is true :

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} \left[C_{a_{i-1}}^{-1}(k_i,1) + \theta_{a_{i-1}}(k_i) - P_{a_i}^{-1}(k_i,1)\right] - \sum_{i=1}^{m} step_{a_i},$$

then $\mathcal{G}_{pcg}$ is live. Since:

$$\sum_{i=1}^{m} \left[C_{a_{i-1}}^{-1}(k_i,1) + \theta_{a_{i-1}}(k_i) - P_{a_i}^{-1}(k_i,1)\right] = \sum_{i=1}^{m} \left[C_{a_i}^{-1}(k_{i+1},1) + \theta_{a_i}(k_{i+1}) - P_{a_i}^{-1}(k_i,1)\right]$$

the previous inequality is equivalent to

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} \left[C_{a_i}^{-1}(k_{i+1},1) + \theta_{a_i}(k_{i+1}) - P_{a_i}^{-1}(k_i,1)\right] - \sum_{i=1}^{m} step_{a_i}$$

and the theorem follows. ■

We illustrate **SCL1** with Figure 3.17. As the theorem states, for each arc, we need to test all phase combinations. We have $\sum_{i=1}^{m} M_0(a_i) = 15$ and $\sum_{i=1}^{m} step_{a_i} = 3$.

We need to test the following set of phases ($[k_1, k_2, k_3]$): $[0,1,1]$, $[1,1,1]$, $[0,1,2]$ and $[1,1,2]$.

- $[0,1,1]$ gives $15 > 13 - 3$ with:
  - $C_{a_1}^{-1}(1,1) + \theta_{a_1}(1) - P_{a_1}^{-1}(0,1) = 0 + 5 - 0 = 5$
  - $C_{a_2}^{-1}(1,1) + \theta_{a_2}(1) - P_{a_2}^{-1}(1,1) = 0 + 2 - 0 = 2$
  - $C_{a_3}^{-1}(0,1) + \theta_{a_3}(0) - P_{a_3}^{-1}(1,1) = 0 + 6 - 0 = 6$
- $[1,1,1]$ gives $15 > 17 - 3$ with:
  - $C_{a_1}^{-1}(1,1) + \theta_{a_1}(1) - P_{a_1}^{-1}(1,1) = 0 + 5 - 2 = 3$
  - $C_{a_2}^{-1}(1,1) + \theta_{a_2}(1) - P_{a_2}^{-1}(1,1) = 0 + 2 - 0 = 2$
  - $C_{a_3}^{-1}(1,1) + \theta_{a_3}(1) - P_{a_3}^{-1}(1,1) = 6 + 6 - 0 = 12$
- $[0,1,2]$ gives $15 > 13 - 3$ with:
  - $C_{a_1}^{-1}(1,1) + \theta_{a_1}(1) - P_{a_1}^{-1}(0,1) = 0 + 5 - 0 = 5$
  - $C_{a_2}^{-1}(2,1) + \theta_{a_2}(2) - P_{a_2}^{-1}(1,1) = 2 + 1 - 0 = 3$
  - $C_{a_3}^{-1}(0,1) + \theta_{a_3}(0) - P_{a_3}^{-1}(2,1) = 0 + 6 - 1 = 5$
- $[1,1,2]$ gives $15 > 17 - 3$ with:
  - $C_{a_1}^{-1}(1,1) + \theta_{a_1}(1) - P_{a_1}^{-1}(1,1) = 0 + 5 - 2 = 3$
  - $C_{a_2}^{-1}(2,1) + \theta_{a_2}(2) - P_{a_2}^{-1}(1,1) = 2 + 1 - 0 = 3$
  - $C_{a_3}^{-1}(1,1) + \theta_{a_3}(1) - P_{a_3}^{-1}(2,1) = 6 + 6 - 1 = 11$

Thus, the sufficient condition of liveness is fulfilled for any m-uple $(k_1, \cdots, k_m) \in \delta_1 \times \cdots \times \delta_m$ of $\mathcal{G}_{pcg}$.

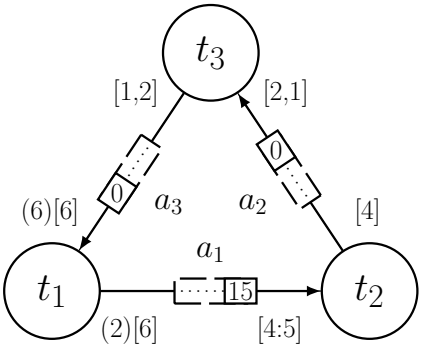


Figure 3.17 – A normalized PCG model live according to **SCL1**.

Sufficient Condition 2 for the PCG model

We now extend the second sufficient condition of liveness of Theorem 5. This condition does not call for an enumeration of the phases:

Theorem 9 (SCL2)**.** *A normalized PCG* $\mathcal{G}_{pcg} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$ *is live if for*

every cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ *of* $\mathcal{G}$

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} W^{pcg}_{a_{i-1},a_i},$$

where $W^{pcg}_{a_{i-1},a_i} = \max_{k\in\delta(t_i)}[C^{-1}_{a_{i-1}}(k,1) + \theta_{a_{i-1}}(k) - P^{-1}_{a_i}(k,1)] - step_{a_i}$ *and indexes are taken modulo* m.

Proof. Suppose that the condition expressed by Theorem 8 is true for any cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ and m-uple $(k_1, \cdots, k_m) \in \delta_1 \times \cdots \times \delta_m$, then we have :

$$\begin{aligned}\sum_{i=1}^{m} M_0(a_i) &> \sum_{i=1}^{m} W^{pcg}_{a_i}(k_i, k_{i+1}) \\ &> \sum_{i=1}^{m} \left(C^{-1}_{a_i}(k_{i+1},1) + \theta_{a_i}(k_{i+1}) - P^{-1}_{a_i}(k_i,1) - step_{a_i}\right) \\ &> \sum_{i=1}^{m} \left(C^{-1}_{a_{i-1}}(k_i,1) + \theta_{a_{i-1}}(k_i) - P^{-1}_{a_i}(k_i,1) - step_{a_i}\right).\end{aligned}$$

Since

$$\begin{aligned}\sum_{i=1}^{m} W^{pcg}_{a_{i-1},a_i} &\geq \sum_{i=1}^{m} \left(C^{-1}_{a_i}(k_{i+1},1) + \theta_{a_i}(k_{i+1}) - P^{-1}_{a_i}(k_i,1) - step_{a_i}\right) \\ &\geq \sum_{i=1}^{m} W^{pcg}_{a_i}(k_i, k_{i+1}),\end{aligned}$$

if $\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} W^{pcg}_{a_i}(k_i, k_{i+1})$ is satisfied then $\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} W^{pcg}_{a_{i-1},a_i}$ is satisfied.

Therefore **SCL1** $\implies$ **SCL2** and $\mathcal{G}_{pcg}$ is live. ■

Pursuing with the example of Figure 3.17. As the theorem states, we need to test each phase of each task of the graph. The formula

$$\max_{k\in\delta_i}[C^{-1}_{a_{i-1}}(k,1) + \theta_{a_{i-1}}(k) - P^{-1}_{a_i}(k,1)]$$

gives:

- for t_1, take the maximum of:
  - for $k_1 = 0$: $C^{-1}_{a_3}(0,1) + \theta_{a_3}(0) - P^{-1}_{a_1}(0,1) = 0 + 6 - 0 = 6$
  - for $k_1 = 1$: $C^{-1}_{a_3}(1,1) + \theta_{a_3}(1) - P^{-1}_{a_1}(1,1) = 6 + 6 - 2 = 10$
- for t_2, take the maximum of:
  - for $k_2 = 1$: $C^{-1}_{a_1}(1,1) + \theta_{a_1}(1) - P^{-1}_{a_2}(1,1) = 0 + 5 - 0 = 5$
- for t_3, take the maximum of:
  - for $k_3 = 1$: $C^{-1}_{a_2}(1,1) + \theta_{a_2}(1) - P^{-1}_{a_3}(1,1) = 0 + 2 - 0 = 2$
  - for $k_3 = 2$: $C^{-1}_{a_2}(2,1) + \theta_{a_2}(2) - P^{-1}_{a_3}(2,1) = 2 + 1 - 1 = 2$

The condition gives $15 > 17 - 3$ and the PCG is checked to be live.

Proof of equivalence between SCL1 and SCL2

The next theorem shows the equivalence between Theorem 8 and Theorem 9.

Theorem 10. *Conditions **SCL1** and **SCL2** are equivalent.*

Proof. As Theorem 9 already shows that **SCL1** $\implies$ **SCL2** we only need to show that **SCL2** $\implies$ **SCL1**.

Suppose that the PCG $\mathcal{G}_{pcg}$ verifies **SCL1** and let $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ be a cycle of $\mathcal{G}_{pcg}$, we have:

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} W_{a_i}^{pcg}(k_i, k_{i+1}),$$

for any m-uple $(k_1, \cdots, k_m) \in \delta_1 \times \cdots \times \delta_m$. Then, denoting by E the set of m-uples $(k_1, \cdots, k_m)$, the previous inequality is equivalent to:

$$\sum_{i=1}^{m} M_0(a_i) > \max_{E} \left(\sum_{i=1}^{m} W_{a_i}^{pcg}(k_i, k_{i+1}) \right),$$

and, since

$$\max_{E} \left(\sum_{i=1}^{m} W_{a_i}^{pcg}(k_i, k_{i+1}) \right) = \sum_{i=1}^{m} W_{a_{i-1},a_i}^{pcg},$$

SCL2 $\implies$ **SCL1** and the sufficient conditions are equivalent. ■

3.5 Conclusion

In this chapter important notions for the SDFG model have been extended to the CSDFG model. These notions, namely consistency, normalization, precedence constraints, useful tokens and two equivalent sufficient conditions of liveness have been further extended to the PCG model. The extensions to the PCG model are one of the contributions of this thesis. All notions presented on SDFG, CSDFG and PCG models have been implemented and experimented in a random graph dataflow generator named `Turbine`. The generator is presented in the next chapter.

Chapter 4

Dataflow graph generation

Contents

4.1 Introduction

The first year of the thesis was dedicated to establishing the state of the art and developing the implementation of a dataflow graph generator called `Turbine` available at `https://github.com/bbodin/turbine`. One purpose of this dataflow graph generator was to overcome the lack of publicly available instances of large dataflow graphs. At the end of the thesis `Turbine` has become a multi-functional tool and provides fast generation of dataflow graphs and multiple tools to manipulate them.

Section 4.2 introduces `Turbine` and its two competitors. Section 4.3 illustrates the three generators through several experiments. Section 4.4 is the conclusion.

4.2 Dataflow graph generators

Before presenting `Turbine`, we first introduce two other dataflow graph generators, SDF3 and PREESM. Section 4.2.1 introduces these dataflow graph generators and Section 4.2.2 is dedicated to `Turbine`.

4.2.1 SDF3 and PREESM

This section presents SDF3 [Stuijk et al., 2006] and the DFtool part of the PREESM framework [Pelcat et al., 2014].

SDF For Free

SDF3 is an open source popular software written in *C++* and available at `http://www.es.ele.tue.nl/sdf3/`. It provides a random dataflow generator and handles SDFG and CSDFG models. SDF3 also includes extensive libraries to analyze and transform SDFGs and CSDFGs. The library includes, among other things, tools to evaluate consistency, compute repetition vectors and compute all possible optimal trade-offs between the storage-space allocated to the channels of an SDFG and the maximal throughput.

SDF3 handles *sdf3* (xml) files format. This format is commonly used by the dataflow community.

PREESM

PREESM is an open source rapid prototyping tool written in Java and available at `http://preesm.sourceforge.net/website/`. Given an architecture and an application it simulates DSP applications and generates source-code to execute them on heterogeneous multi/many-core embedded systems.

The software allows the parallelization on a multicore system. It proposes, among other things, throughput optimization using software pipelining, evaluation and op-

timization of the memory footprint, evaluation of actors execution time, and also allows SDFG generation. PREESM further provides an *sdf3* parser.

4.2.2 `Turbine`

`Turbine` is the generator implemented during this thesis. The generator is implemented in Python and uses NetworkX (`https://networkx.github.io`) to handle the graph data structure. It manipulates simple files (non-xml) in order to facilitate self-made instances, however the xml format of *sdf3* is also supported.

`Turbine` generates live SDFGs, CSDFGs and PCGs up to 10,000 tasks in less than 30 seconds. Random generation is divided in three steps described in the following section. The first step generates a random graph with nodes and arcs, the second step computes consumption and production weights on the graph. The last step computes a live initial marking.

The rest of the section describes the three steps of the random dataflow graph generation.

Graph generation

The graph generator can generate general or acyclic graphs. The user decides the number of tasks and a min/max interval for the task degree. The generator respects this unless it is not possible.

Both general and acyclic graph are generated in two steps. The first step generates a random connected graph (with the acyclic restriction for the acyclic case). Then, arcs are added until the average degree, $(min + max)/2$, is reached. To simplify the acyclic graph generation, the graph is first seen as a path and arcs are added in the same direction as the path. The user can decide if the generation can generates self-loops (only for general graphs) and multi-arcs. If the multi-arc parameter is set, multiple arcs between the same source and destination may be generated.

Graph generation operates similarly in SDF3 and PREESM.

Rate Generation

In the rate generation phase, `Turbine` generates the production and consumption weights for each arc of the graph and returns a normalized graph. To guarantee consistency, the repetition factor of each task is decided first. The repetition vector is computed by first setting its sum and then using a combination of an exponential and a multinomial distribution to create its components.

Rate generation depends on the model (SDFG, CSDFG, PCG). For the CSDFG and PCG models, the size of the consumption and production vectors is decided randomly using a uniform distribution with min/max parameters. Then, for each task t_i

of the graph, Z_i is deduced from the repetition factor according to $Z_i = \frac{lcm(R_1,\cdots,R_{|\mathcal{T}|})}{R_i}$. For the CSDFG and PCG models, cyclic phase production and consumption vectors are deduced from Z_i. In the case of a PCG, initial phases and thresholds are generated uniformly on an interval with a min/max parameters.

SDF3 considers two cases for rate generation. If the repetition factor parameters are set, the components of R are computed randomly according to the parameters. Rates are then derived from the components of R. If the parameters R are not set, weights are selected randomly and a depth-first search algorithm is used to modify some of the weights in order to make the graph consistent. PREESM generates repetition factors randomly on an interval, defined by min/max parameters and deduces weights from the repetition vector.

Initial marking computation

The computation of the initial marking is the most critical step since no exact polynomial-time condition of liveness exists. The computation of the initial marking in `Turbine` is based on the sufficient conditions **SCL1** and **SCL2** expressed before. These sufficient conditions of liveness were expressed in Section 2.4.4 for the SDFG model, in Section 3.2.6 for the CSDFG model and in Section 3.4.5 for the PCG model.

The sufficient conditions for the three dataflow models are grouped in two linear programs, **SCL1** and **SCL2**:

Mixed-Integer Linear Program 4: liveness (SCL1)

$$\text{minimize} \quad \sum_{a\in\mathcal{A}} M_0(a)$$

$$\text{subject to} \begin{cases} \gamma_{t_j} - \gamma_{t_i} + M_0(a_j) - \varepsilon \geq & W_a^{scl}(k_i,k_j) & \forall a=(t_i,t_j)\in\mathcal{A}, \\ & & \forall k_i \in \{1,\cdots\varphi_i\}, \forall k_j \in \{1,\cdots\varphi_j\} \\ M_0(a) = & step_a \cdot m_0(a) & \forall a \in \mathcal{A} \\ \gamma_{t_i} \in \mathbb{R} & & \forall t_i \in \mathcal{T} \\ M_0(a) \in \mathbb{N}, m_0(a) \in \mathbb{N} & & \forall a \in \mathcal{A} \\ \varepsilon \in \mathbb{R}^{*+} \text{ very small} & & \end{cases}$$

with

$$W_a^{scl}(k_i,k_j) = \begin{cases} c_a - gcd_a & \text{if } \mathcal{G} \text{ is an SDFG} \\ W_a^{csdf}(k_i,k_j) & \text{if } \mathcal{G} \text{ is a CSDFG} \\ W_a^{pcg}(k_i,k_j) & \text{if } \mathcal{G} \text{ is a PCG} \end{cases}.$$

Note that for the SDFG model k_i and k_j do not appear in the formula. In fact the program for the SDFG is that of a CSDFG with only one phase per task ($\varphi_i = 1$) and $k_i = k_j = 1$.

Let $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ be a dataflow model. We define $\Gamma^+(t)$ (*resp.* $\Gamma^-(t)$) the set of input (*resp.* output) arcs of the task $t \in \mathcal{T}$. Let $\mathcal{D}$ the set of pairs of arcs (a_i, a_j) such that $a_i \in \Gamma^+(t)$ and $a_j \in \Gamma^-(t)$, for $t \in \mathcal{T}$.

Mixed-Integer Linear Program 5: liveness (SCL2)

$$\begin{array}{ll} \text{minimize} & \sum_{a\in\mathcal{A}} M_0(a) \\ \text{subject to} & \begin{cases} \gamma_{a_j} - \gamma_{a_i} + M_0(a_j) - \varepsilon \geq W^{sc2}_{a_i,a_j} & \forall (a_i,a_j) \in \mathcal{D} \\ M_0(a) = step_a \cdot m_0(a) & \forall a \in \mathcal{A} \\ M_0(a) \in \mathbb{N}, m_0(a) \in \mathbb{N}, \gamma_a \in \mathbb{R} & \forall a \in \mathcal{A} \\ \varepsilon \in \mathbb{R}^{*+} \text{ very small} \end{cases} \end{array}$$

with

$$W^{sc2}_{a_i,a_j} = \begin{cases} c_{a_j} - gcd_{a_j} & \text{if } \mathcal{G} \text{ is an SDFG} \\ W^{csdf}_{a_i,a_j} & \text{if } \mathcal{G} \text{ is a CSDFG} \\ W^{pcg}_{a_j,a_j} & \text{if } \mathcal{G} \text{ is a PCG} \end{cases}.$$

`Turbine` solves the linear program either by using a linear approximation or by calling the mixed integer program solver of Gurobi. The linear approximation is rounded up according to the useful tokens assumption. By default `Turbine` uses the linear approximation. Among **SCL1** and **SCL2** it chooses the program with smallest number of constraints.

Live markings are generated in SDF3 and PREESM using a simple sufficient condition of liveness:

Theorem 11. *Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a consistent SDFG. $\mathcal{G}_{sdf}$ is live if there is in every cycle $\mu = (t_1, a_1, \cdots, t_m, a_m, t_1)$ of $\mathcal{G}_{sdf}$ at least one arc $a = (t_i, \cdot)$ with $M_0(a) = R_i.p_a$.*

The computation of a live initial marking consists in selecting a subset of arcs $\mathcal{A}' \in \mathcal{A}$ such that the subgraph $\mathcal{G}' = (\mathcal{T}, \mathcal{A} - \mathcal{A}')$ has no cycle and, then, setting the initial marking to:

$$M_0(a) = \begin{cases} R_i.p_a \text{ if } a = (t_i, \cdot) \in \mathcal{A}' \\ 0 \quad \text{otherwise} \end{cases}$$

SDF3 computes the initial marking by selecting arcs in $\mathcal{A}'$ using a depth-first algorithm. PREESM first selects a set of tasks $\mathcal{T}' \in \mathcal{T}$ such that the subgraph $\mathcal{G}'' = (\mathcal{T} - \mathcal{T}', \mathcal{A})$ is acyclic, then sets $\mathcal{A}' = \{a = (t_i, t_j) \in \mathcal{A}, t_i = \mathcal{T}'\}$. In both cases, the procedure does not attempt to minimize the initial marking. Theorem 11 is easily extended to the CSDFG model using the corresponding definition of p_a and is therefore not repeated for that model.

Other features offered by `Turbine`

Besides being a graph generator, `Turbine` offers multiple tools to manipulate and perform computation on dataflow graphs such as normalization/de-normalization or live initial marking computation. The initial marking computation is implemented in both linear and mixed-integer versions. The linear implementation is used by

default and employs the solver GLPK. The mixed-integer version results in a smaller total number of tokens, but is not scalable since it uses the mixed-integer solver of the Gurobi solver. Initial marking computation under throughput constraint is also available in both linear approximation and mixed-integer version. Finally, repetition factor computation is possible.

Many other features have been implemented, such as consistency tests and a necessary and sufficient condition of liveness using the native symbolic execution.

`Turbine` handles two file formats. Its own, the *.tur* format and the *.sdf3* format. The *.tur* file format is simple and easy to write in, giving files 10 times smaller than the *.sdf3* files.

Technical details on `Turbine`

The installation of `Turbine` requires NetworkX, GLPK and swiglpk. NetworkX is a Python library in open source license and proposes many features about graphs. `Turbine` uses it to handle the graph data structure. GLPK is a solver for linear programs or mixed-integer programs released under GNU license. It is written in *C* and is interfaced with `Turbine` by Swiglpk. Swiglpk allows flexible usage of GLPK and handles many different versions. The solver Gurobi was added only for the purpose of experiment since the GLPK mixed-integer solver cannot compete—in terms of computation time—with industrial quality software like Gurobi or CPLEX. Gurobi requires a license (free for academic purpose).

4.3 Experimentation

In this section we provide experimental results to compare `Turbine` with SDF3 and PREESM. The next section describes the experimental conditions. Section 4.3.2 compares generating times. In Section 4.3.3 we compare the number of tokens of the initial markings computed by the three generators.

4.3.1 Experimental conditions

Most experiments were performed on graphs with four different sizes: *tiny*, *small*, *medium* and *large*, having respectively 10, 100, 1,000 and 10,000 tasks.

The experiments were executed on a four-core *Intel Core I5* 660 at 3.33 Ghz with 6GB of RAM running under Linux and with Python 2.7. Generated dataflow graphs have the same parameters: average repetition factor $R_i = 5$ and average degree of 3. They are generated with cycles and with no multi-arc or self-loop.

4.3.2 Comparison of generation times

To compare performance of PREESM, SDF3 and `Turbine`, 100 instances of each size were generated by each generator. The last versions of SDF3, PREESM and `Turbine` were used (24 July 2014 for SDF3 and v2.2.3 for PREESM). Note that PREESM only generates SDFGs and SDF3 does not handle the PCG model.

Table 4.1(a) presents the average generation time of SDFGs for each graph size and for each generator. SDF3 was not able to generate *large* graphs (the generator was stopped after 24h). The generation of *large* graphs with PREESM is very memory consuming (up to 32GB), the $\star$ indicates that time was measured on a two-Xeon server with 48GB of RAM. Table 4.1(b) presents the average generation time of CSDFGs of each size for `Turbine` and SDF3. SDF3 was not able to generate *large* graphs.

$\|\mathcal{T}\|$	`Turbine`	SDF3	PREESM
Tiny	5ms	19ms	9ms
Small	44ms	313ms	58ms
Medium	592ms	1h24mn	7.2s
Large	20.7s	-	1h39mn*

(a) SDFG

$\|\mathcal{T}\|$	`Turbine`	SDF3
Tiny	7ms	23ms
Small	62ms	315ms
Medium	806ms	1h27mn
Large	20.4s	-

(b) CSDFG

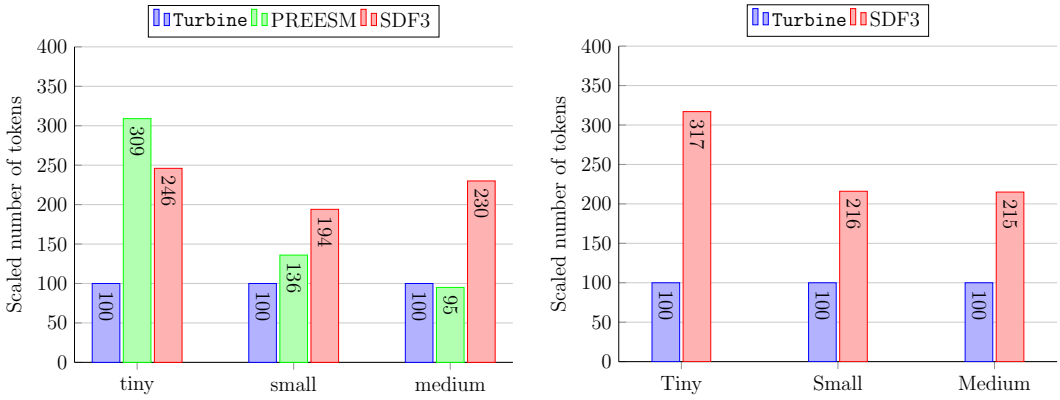
Table 4.1 – (a) SDFG generation time with `Turbine`, SDF3 and PREESM. (b) CSDFG generation time with `Turbine` and SDF3.

Since the first release, the generation time of `Turbine` was greatly improved. For example, the generation time for *large* SDFGs which took an average of 10 minutes, now takes about 20 seconds, hence is 30 times shorter. This is due to several improvements in the graph generation, especially for the arcs and the use of a more recent version of GLPK. The solver outperforms the other generators for both SDFGs and CSDFGs.

4.3.3 Comparison of the initial markings

The computed initial marking must ensure liveness and also be minimal such that the total number of tokens be close to what happens in real applications and leaves room for the user to add tokens. In this section we present experiments to compare the initial marking computed by each generator.

Initial marking computation is part of dataflow graph generation. It calls for modifying the code of SDF3 and PREESM in order to test the algorithm on pre-generated graphs. The total number of tokens is compared among 100 instances of tiny, small and medium-size graphs pre-generated with `Turbine`. Large instances were not tested since PREESM gives an error because of a restriction on integer size. Figure 4.1 shows the average number of tokens obtained for SDFGs and CSDFGs using `Turbine` as reference.



(a) SDFG **(b)** CSDFG

Figure 4.1 – Total number of tokens for the initial markings computed for (a) SDFGs and (b) CSDFGs of tiny, small and medium size. The number of tokens is scaled to 100 for `Turbine`, which is used as reference.

The number of tokens is smaller for `Turbine` on tiny and small instances. However, PREESM gives a smaller number on medium graphs. For acyclic graphs, the initial markings generated by `Turbine` do not contain any token, in contrast with those of SDF3 and PREESM. None of the graphs generated in the experiments whose results are shown in Figure 4.1 is acyclic.

4.3.4 Performance of `Turbine`

This section presents various experiments on `Turbine` to illustrate its scalability. They cover PCG generation time, comparison of **SCL1** and **SCL2** computation times, and comparison of initial markings generated using optimal mixed-integer programming and a linear approximation.

PCG generation time

Table 4.2 shows average generation times for PCGs of size defined previously.

$\lvert\mathcal{T}\rvert$	`Turbine`
Tiny	10ms
Small	99ms
Medium	1.1s
Large	24.7s

Table 4.2 – PCG generation times using `Turbine`.

The generation times are close to those for CSDFGs of the same size and have been greatly improved since the first release: at first the generation of a PCG of *large* size took two hours, it now takes less than 25 seconds.

Comparison between computation times using liveness conditions SCL1 and SCL2

In this section performance of **SCL1** and **SCL2** with two versions of `Turbine` is discussed. We compare the first version of `Turbine`, released in 2013, and the last version, released in 2016. The 2013 version uses GLPK v4.48 while the last release uses GLPK v4.58. Experiments have been performed on SDFGs, CSDFGs and PCGs. Although v4.58 was already available in 2013, the interface Python-glpk in use at the time restricted us to v4.48. Version v4.58 is now used, as we switched to *swiglpk*.

The graphs are pre-generated, so that the generation time measured in the experiments is restricted to the initial marking computation. The experiments include 100 graphs of each size. The presolver in ON for both version of GLPK in order to improve the linear program by removing unused variables and redundant constraints. The presolver has been significantly improved between version v4.48 and v4.58.

Table 4.3 shows average computation times for the initial markings using **SCL1** and **SCL2** with GLPK v4.48.

	SDF		CSDF		PCG	
$\|\mathcal{T}\|$	SCL1	SCL2	SCL1	SCL2	SCL1	SCL2
Tiny	0.004s	0.007s	0.01s	0.007s	0.05s	0.01s
Small	0.06s	0.2s	0.3s	0.2s	1.7s	0.2s
Medium	2.3s	15.4s	59.4s	14.7s	1h10mn	15.2s

Table 4.3 – Initial marking computation times averaged on 100 instances using **SCL1** and **SCL2** on SDFGs, CSDFGs and PCGs with the 2013 `Turbine` release. The experiments, made in 2013, did not test *large* graphs.

Table 4.4 gives the results of the same experiments using now version v4.58 of GLPK.

	SDF		CSDF		PCG	
$\|\mathcal{T}\|$	SCL1	SCL2	SCL1	SCL2	SCL1	SCL2
Tiny	0.004s	0.003s	0.005s	0.003s	0.006s	0.003s
Small	0.014s	0.018s	0.047s	0.018s	0.084s	0.02s
Medium	0.214s	0.225s	1.99s	0.233s	4.301s	0.255s
Large	12.72s	8.182s	6mn5s	8.334s	14mn3s	8.677s

Table 4.4 – Initial marking computation times averaged on 100 instances using **SCL1** and **SCL2** on SDFGs, CSDFGs and PCGs with the 2016 `Turbine` release.

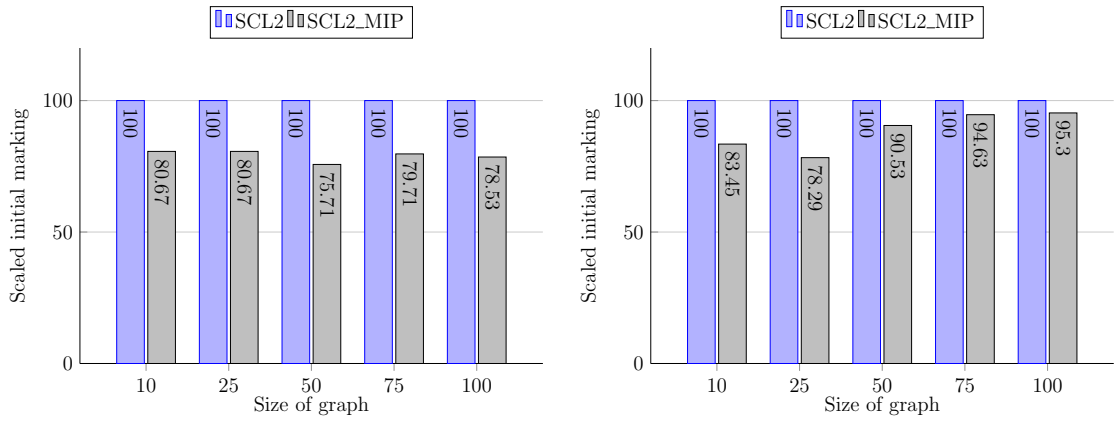
Tables 4.3 and 4.4 show how initial marking computation time improved between the first and the last release of `Turbine`. The usefulness of the **SCL2** implementation for large graphs appears clearly in Table 4.4. **SCL2**'s performance is very close to **SCL1**'s on SDFGs of *tiny*, *small* and *medium* size and is better in every other case.

This higher efficiency is largely due to the pre-solver improvement. Despite that, the implementation of **SCL1** is still of interest for SDFGs with high average degree.

Comparison between SCL2 and SCL2_MIP

Figure 4.2 illustrates the difference between the number of tokens of the initial markings obtained using the linear approximation version of **SCL2**—rounding up to the next useful token—and its mixed-integer linear version denoted **SCL2_MIP**, which finds an optimal solution. The linear approximation is solved with GLPK while the mixed-integer linear version is solved by Gurobi. Figure 4.2 shows the number of tokens of initial markings computed with **SCL2_MIP** compared to **SCL2**. The results are averages on 100 instances generated with `Turbine`.

During the experiment the computation time of **SCL2_MIP** is limited to 5 minutes; the instances requiring more time are ignored. Except for 100-task CSDFGs, for which only 46 instances were solved, all other graphs required less than 5 minutes.



(a) SDFG **(b)** CSDFG

Figure 4.2 – Initial marking computation times using **SCL2** and **SCL2_MIP** on (a) SDFGs and (b) CSDFGs with 10 to 100 tasks.

The initial markings obtained using the linear approximation are closer to the optimum for CSDFGs than SDFGs. Also, the difference appears to decrease with the size of the CSDFGs.

4.4 Conclusion

This chapter presented the dataflow graph generator `Turbine` and two of its competitors, SDF3 and PREESM. The dataflow generators have been compared in terms of computation time and number of tokens of the initial marking for both SDFGs and CSDFGs. The improvement in performance between releases of `Turbine` has also been illustrated. Finally, an experiment compared the linear approximation to the exact solution for the initial marking in terms of the total number of tokens.

This chapter concludes the work done on the PCG model and the related implementation. The next chapters are dedicated to the second part of the thesis, on a dataflow graph mapping problem on distributed architectures.

Chapter 5

Mapping problem with memory constraints

Contents

5.1 Introduction

Due to the multiplication of processing elements on a single chip, the mapping problem is considered one the most urgent problems for implementing embedded systems [Marwedel et al., 2011]. The new approach proposed in this thesis offers a fast evaluation of the memory consumption of mappings that guarantee liveness or satisfy a constraint on throughput .

Our work on the mapping problem is presented in two chapters. This chapter considers theoretical aspects of the problem and elaborates techniques of evaluation of the memory footprint of a Synchronous Dataflow Graph (SDFG) or a Cyclo-Static Dataflow Graph (CSDFG) buffer. Two versions of the evaluation are considered, one assuming a bounded buffer and another with the addition of a throughput constraint.

The following chapter considers practical aspects of the problem and proposes algorithms and experiments using a new model to evaluate the memory footprint of an application.

Section 5.2 presents the mapping problem. Section 5.3 demonstrates a technique to evaluate memory consumption of an application with live or throughput constraints using the SDFG model. Section 5.4 extends the technique to the CSDFG model. Section 6.10 concluded.

5.2 Mapping Problem

The mapping problem is known to be NP-hard [Singh et al., 2013]. It consists in assigning functions of an application to the processing elements of a specific architecture to optimize performance. Many mapping problem variants exist depending on the target architecture and the selected performance objective.

Section 5.2.1 describes the mapping problem studied in this thesis. Section 5.2.2 presents the target architecture. Finally, Section 5.2.3 describes the state of the art for liveness and throughput evaluation.

5.2.1 Problem description

The mapping problem studied in this thesis consists in finding an allocation for the tasks of a dataflow graph on a distributed architecture. The architecture is composed of many clusters linked by means of a Network on Chip (NoC). Each cluster contains processing elements and a memory shared between these processing elements. The goal of task allocation is to minimize the number of clusters used while respecting the memory constraint.

The affectation problem under memory constraint is approached with a new insight. The memory consumption other than program and data storage comes from buffer communication between actors and this consumption depends on the affectation. Indeed, due to the distributed architecture, if two communicating tasks are

affected to different clusters, a copy is performed from one cluster to another resulting in a delay during the communication and an increase in memory consumption.

The evaluation comes in two versions, one with liveness guarantee, the other with throughput constraint. The liveness guarantee ensures that the application will function without dead-lock. The throughput constraint imposes that the throughput be the same as if there were no communication delays. The goal of this work is to find an efficient evaluation of the mapping while reaching high scalability: thousands of tasks mapped on an architecture composed of many clusters.

5.2.2 Targeted architecture

The targeted architecture is inspired from the Massively Parallel Processor Array (MPPA) developed by Kalray [Aubry et al., 2013]. An MPPA chip is an array of 16 homogeneous clusters connected by a high-speed NoC (up to 12 GB/s). A cluster is composed of 16 processing cores and 2MB of shared memory.

The targeted architecture for the mapping problem illustrated in Figure 5.1 is a simplification of the MPPA. We define a set of clusters $\mathcal{S}$ with a limited amount of memory M_{max}. The NoC has a latency depending on the size of the data packets and a constant bandwidth. Every cluster is supposed to communicate with any other with the same latency and the load of the NoC does not affect the bandwidth. To simplify the approach, the routing part is abstracted.

We assume wormhole routing so that network latency comprises two parts [Nikitin and Cortadella, 2009]: a constant term d, accounting for header propagation time and contention delay, and a term proportional to packet size, accounting for data transmission time in a pipelined manner.

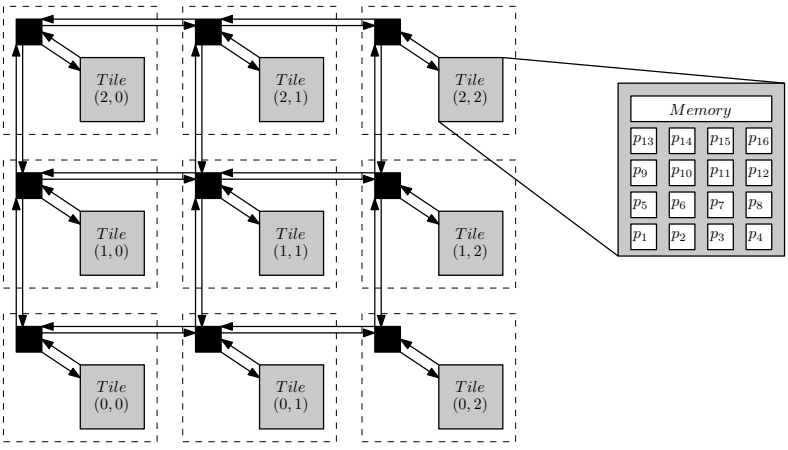


Figure 5.1 – The targeted architecture for the mapping problem.

5.2.3 State of the art

The mapping problem, which consists in assigning functions of an application on a minimal number of processing elements of an architecture (shared by several ap-

plications) while satisfying budget (such as memory or power) and performance (such as throughput or latency) constraints, is known to be NP-hard [Singh et al., 2013]. Its solution requires evaluations of the budget and performance characteristics of many candidate mappings. When an application is modeled by an SDFG, no polynomial—in terms of that weighted-graph description—algorithm has been found for throughput evaluation, making the mapping problem even more challenging.

Symbolic execution is used in [Stuijk et al., 2007] to evaluate SDFG liveness and throughput. Tasks are executed as many times as necessary, leading to an exponential—in terms of the SDFG parameters—number of executions. Integer Linear Programming and Constraint Programming are used respectively in [Lin et al., 2012] and [Zhu et al., 2010] to model an SDFG mapping problem and obtain feasible schedules with a time complexity comparable to the symbolic execution. The mapping problem for a Homogeneous Synchronous Dataflow Graph (HSDFG), for which polynomial evaluations exist is considered in [Bonfietti et al., 2010] and [Zhou et al., 2013]. An SDFG may be transformed into an HSDFG at the expense of a possibly exponential growth of the size of the description, making polynomial techniques on HSDFGs exponential for the underlying SDFGs.

All the previous methods are not scalable and cannot handle large industrial instances. By using sufficient conditions—hence conservative to some extent—liveness and throughput of a mapping may be evaluated polynomially. Bekooij *et al.* [Bekooij et al., 2006] use periodic evaluation to obtain throughput guarantees for applications modeled by acyclic SDFGs. `Turbine`, presented in [Bodin et al., 2014], develops and tests a more general approach introduced in [Benabid-Najjar et al., 2012] to generate random live SDFGs with cycles.

5.3 Memory evaluation with an SDFG model

This section presents a technique to evaluate the memory consumption of an application modeled by an SDFG and assigned to a distributed architecture. The results focus on a single buffer and provide a minimum live initial marking for a buffer shared between two clusters and a sufficient initial marking which guarantees a given throughput.

The next section introduces the boundedness property used to evaluate the memory consumption of a buffer. Section 5.3.2 describes how the memory footprint is evaluated if a buffer is shared between two clusters. Section 5.3.4 addresses the liveness problem in the case of a buffer between two clusters. Section 5.3.6 presents a solution for a live initial marking of a buffer between two clusters. Section 5.3.7 addresses the liveness problem with throughput constraint in the case of a buffer between two clusters. Finally, Section 5.3.8 presents a solution with a throughput constraint.

5.3.1 Boundedness: a property of the dataflow models

A dataflow graph is bounded if its marking is bounded when it is executed. We identify two levels of evaluation of boundedness for dataflow graphs, the exact evaluation which gives an exact marking at a specific time for a given schedule, and a global evaluation which supposes that the capacity of the buffers is bounded. Exact evaluation requires the execution of the schedule hence is not scalable. As this work focuses on scalability the evaluation is performed globally—resulting in a global upper bound on the marking instead of a local upper bound for each buffer—and gives guarantee instead of an exact evaluation.

The bounded buffer assumption is modeled by adding to each arc $a = (t_i, t_j) \in \mathcal{A}$ a backward arc (also called back-pressure arc [Bekooij et al., 2006]) $\overline{a} = (t_j, t_i)$ with $c_{\overline{a}} = p_a$ and $p_{\overline{a}} = c_a$ as illustrated in Figure 5.2. The size of the buffer a is then $\sigma_a = M_0(a) + M_0(\overline{a})$. In the sequel a bounded arc a implies the presence of a backward arc $\overline{a}$.

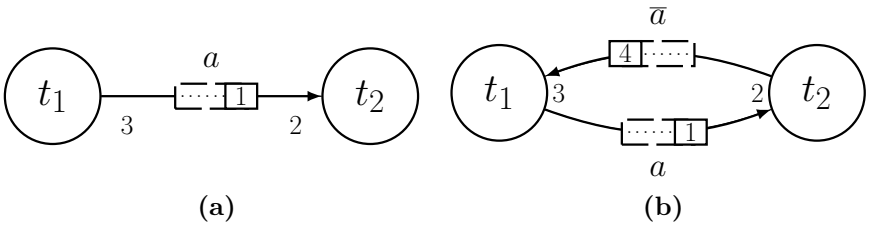


Figure 5.2 – (a) A buffer $a = (t_1, t_2)$ with $p_a = 3$, $c_a = 2$ and $M_0(a) = 1$; (b) the bounded version of buffer a with $\sigma_a = 5$.

5.3.2 Communication memory footprint

This section explains how the memory footprint of a communication is evaluated. The explanation uses the SDFG model but is easily extended to the CSDFG model. We define $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ an application modeled by a SDFG and $\mathcal{S}_c$ a set of clusters with M_{max} the amount of memory for a cluster $c \in \mathcal{S}$.

Let $a = (t_i, t_j) \in \mathcal{A}$ be a bounded buffer. If t_i and t_j are allocated in the same cluster, the memory footprint is σ_a in this cluster. If t_i and t_j are in two different clusters, a new task t_c (c for communication) is added between t_i and t_j to model the communication through the NoC. The execution time of t_c is $\ell_c = d + \frac{Z_c}{B}$ with d a constant latency, Z_c the data packet size and B the bandwith of a NoC channel. Thus, the bounded buffer $a = (t_i, t_j)$ is composed of two bounded buffers $a_i = (t_i, t_c)$ and $a_j = (t_c, t_j)$. The memory footprint of the communication between t_i and t_j is then σ_{a_i} in t_i's cluster and σ_{a_j} in t_j's cluster.

Figure 5.3 illustrates a bounded buffer with the two tasks is connects on (a) the same cluster and (b) two different clusters. Note that the graph is normalized thus, instead of a production and a consumption weight for each arc, only one weight per task is pictured.

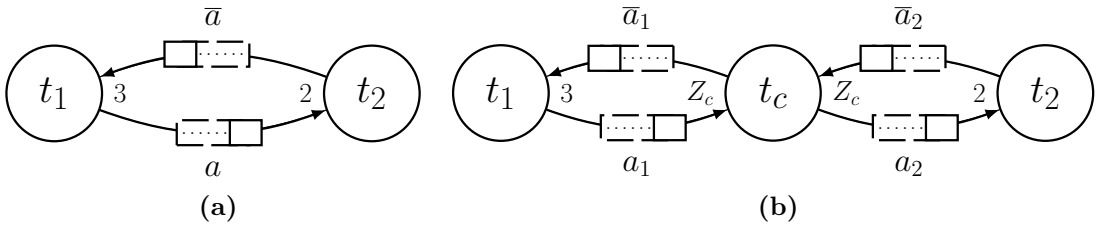


Figure 5.3 – (a) A bounded buffer $a = (t_1, t_2)$; (b) the same bounded buffer a between two clusters with task t_c (c for communication) representing data transfers through the NoC and Z_c the transfer rate.

Figure 5.3 could have shown a CSDFG since this does not change the communication model, the memory footprint depends on the initial marking and not on the production and consumption rates.

Figure 5.4 depicts two different mappings for an application composed of four tasks.

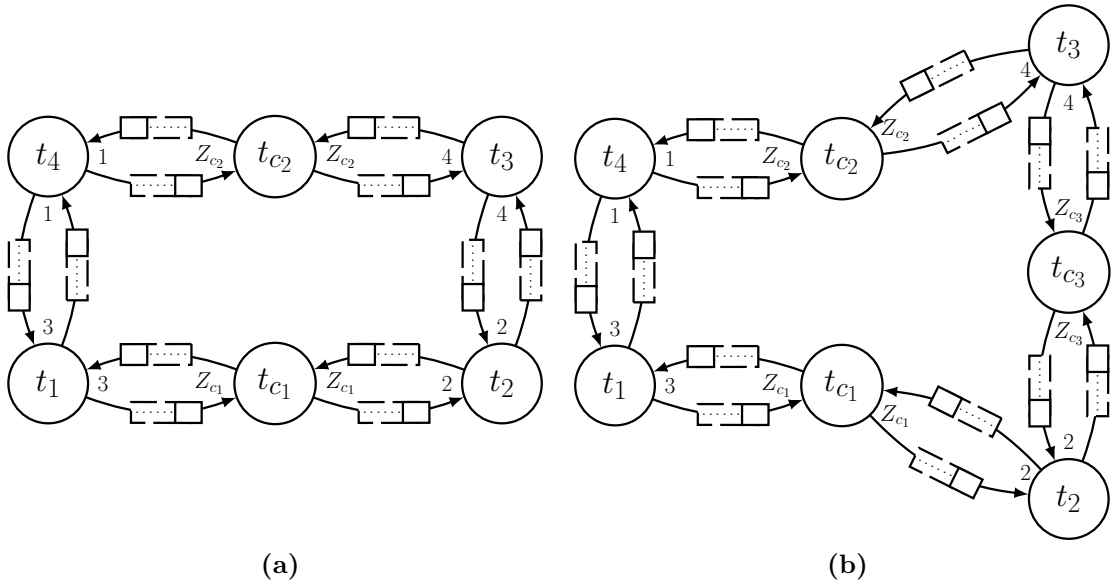


Figure 5.4 – (a) A mapping with t_1 and t_4 on one cluster and t_2 and t_3 on another. (b) The same application with t_1 and t_4 mapped on one cluster, t_2 on a second cluster and t_3 on a third.

Note that the communication tasks in Figure 5.4 have distinct consumption and production rates since they are independent from each other.

5.3.3 Height of an SDFG

The notion of height of an arc will be useful for the analysis that follows. Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized SDFG with $a = (t_i, t_j) \in \mathcal{A}$. The height

of the arc a is denoted $H(a) = M_0(a) + gcd_a - Z_j$ and the height of a cycle μ is $H(\mu) = \sum_{a \in \mu} H(a)$.

Note that the sufficient condition of liveness for an SDFG, **SCL**, expressed by Theorem 1 in Section 2.4.4 is equivalent to $H(\mu) > 0$, $\forall \mu \in \mathcal{G}_{sdf}$.

5.3.4 Liveness guarantee for an inter-cluster buffer

Let $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a bounded SDFG that satisfies the sufficient condition of liveness **SCL**. Suppose that $a = (t_i, t_j)$ is a bounded buffer such that t_i and t_j are in different clusters. We model the communication through the NoC by splitting the buffer in two and inserting an additional task denoted t_c, which performs (virtually) the transfer of the data. This new task is linked to t_i and t_j using two bounded buffers, a_i and a_j, as pictured in Figure 5.5. Z_c is the normalized weight of task t_c and its value will be set later.

Our aim is to determine Z_c and the initial markings of the couple of bounded buffers, a_i and a_j illustrated in Figure 5.5, so that the memory be minimized. The memory corresponding to σ_{a_i} (*resp.* σ_{a_j}) is associated to the bounded buffer a_i (*resp.* a_j) and is located in the cluster of t_i (*resp.* t_j).

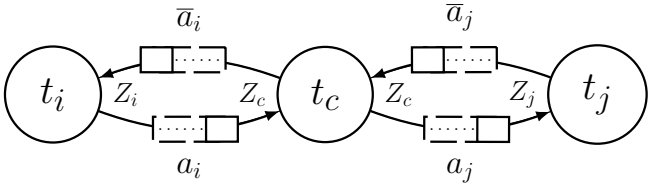


Figure 5.5 – Model of communication between two clusters via a bounded buffer. Task t_c models data transfers through the NoC.

Let $\mathcal{G}'_{sdf}$ be the graph $\mathcal{G}_{sdf}$ with the bounded buffer $a = (t_i, t_j)$ from $\mathcal{G}_{sdf}$ replaced by the two bounded buffers $a_i = (t_i, t_c)$ and $a_j = (t_c, t_j)$. We denote $\overline{a}_i$, $\overline{a}_j$ the backward arcs of a_i and a_j, respectively.

Lemma 10. *Let $\mathcal{G}'_{sdf}$ be the SDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{sdf}$ by a communication task and its four associated arcs according to Figure 5.5. If $\mathcal{G}_{sdf}$ verifies **SCL** and the following four conditions*

$$\begin{cases} H(a_i) + H(\overline{a}_i) > 0 \\ H(a_j) + H(\overline{a}_j) > 0 \\ H(a_i) + H(a_j) \geq H(a) \\ H(\overline{a}_i) + H(\overline{a}_j) \geq H(\overline{a}) \,, \end{cases}$$

*then $\mathcal{G}'_{sdf}$ verifies **SCL**.*

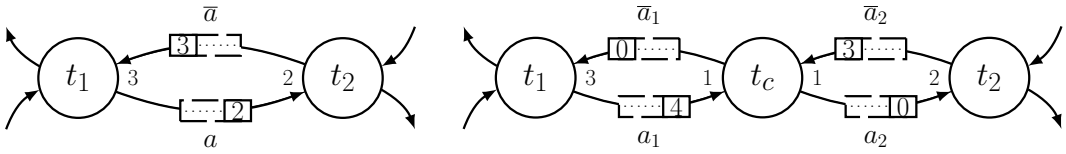
Proof. If $\mathcal{G}_{sdf}$ verifies **SCL**, a sufficient condition of liveness of $\mathcal{G}'_{sdf}$ is that the following four additional conditions be met:

- Cycles (t_i, t_c, t_i) and (t_j, t_c, t_j) of $\mathcal{G}'_{sdf}$ must verify **SCL**, thus $H(a_i)+H(\overline{a}_i)>0$ and $H(a_j)+H(\overline{a}_j)>0$.
- To every cycle μ of $\mathcal{G}$ passing through a (resp. $\overline{a}$) is associated a unique cycle of $\mathcal{G}'_{sdf}$ passing through a_i and a_j (resp. $\overline{a}_i$ and $\overline{a}_j$). **SCL** is then ensured by imposing $H(a_i)+H(a_j)\geq H(a)$ and $H(\overline{a}_i)+H(\overline{a}_j)\geq H(\overline{a})$. ■

Figure 5.6 illustrates a live marking according to Lemma 10 since we have:

$$H(a)=2-2+1=1 \qquad H(\overline{a})=3-3+1=1$$

$$H(a_1)=4-1+1=5 \qquad H(\overline{a}_1)=30-3+1=-2$$
$$H(a_2)=0-2+1=-1 \qquad H(\overline{a}_2)=3-1+1=3.$$



(a) **(b)**

Figure 5.6 – (a) A bounded buffer $a=(t_1,t_2)$; (b) the same bounded buffer a betwen two clusters with a live initial marking according to Lemma 10.

5.3.5 Optimal transfer rate for live minimum memory footprint

This section shows that the value $Z_c=\gcd_a$ yields the smallest value for the total initial marking satisfying the sufficient condition of liveness **SCL**.

Lemma 11. *Consider a couple* $(a,b)\in\mathbb{N}^*\times\mathbb{N}^*$. *The function* $f:\mathbb{N}^*\to\mathbb{Z}$ *defined as*

$$f_{a,b}(x)=x-\gcd(x,a)-\gcd(x,b)$$

is minimum for $x=\gcd(a,b)$.

Proof. We shall substitute f for $f_{a,b}$ in the proof.

If x divides neither a nor b, then $f(x)\geq 0$ since $\gcd(x,a)\leq\frac{x}{2}$ and $\gcd(x,b)\leq\frac{x}{2}$. Otherwise:

- if x divides a, then $\gcd(x,a)=x$ and $f(x)=-\gcd(x,b)=-\gcd(x,a,b)\geq-\gcd(a,b)$,
- if x divides b, then $f(x)=-\gcd(x,a,b)\geq-\gcd(a,b)$.

Thus if x divides either a or b, $f(x)=-\gcd(x,a,b)$, and the minimum $f(x)=-\gcd(a,b)$ is attained if x is a multiple of $\gcd(a,b)$. Moreover the smallest value of

x is obtained for $x = \gcd(a, b)$. ■

Theorem 12. *For any bounded buffer $a = (t_i, t_j)$ between two clusters, the value $Z_c = \gcd_a$ minimizes the total memory required for the buffer according to the sufficient condition of liveness **SCL**. The conditions expressed by Lemma 10 become then*

$$\begin{cases} M_0(a_i) + M_0(\overline{a}_i) \geq Z_i \\ M_0(a_j) + M_0(\overline{a}_j) \geq Z_j \\ M_0(a_i) + M_0(a_j) \geq M_0(a) \\ M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq M_0(\overline{a}) \,. \end{cases}$$

Proof. Replacing H by its definition, the first two inequalities of Lemma 10 become:

$$M_0(a_i) + M_0(\overline{a}_i) > Z_i + f_{Z_i,Z_i}(Z_c),$$
$$M_0(a_j) + M_0(\overline{a}_j) > Z_j + f_{Z_j,Z_j}(Z_c).$$

Similarly, the last two inequalities become

$$M_0(a_i) + M_0(a_j) \geq M_0(a) + gcd_a + f_{Z_i,Z_j}(Z_c),$$
$$M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq M_0(\overline{a}) + gcd_a + f_{Z_i,Z_j}(Z_c).$$

Since $f_{Z_i,Z_i}(Z_c) + f_{Z_j,Z_j}(Z_c) = 2f_{Z_i,Z_j}(Z_c)$, we obtain when adding up these four inequalities the following condition on the total memory required for buffer a:

$$\begin{aligned} &2(M_0(a_i) + M_0(\overline{a}_i) + M_0(a_j) + M_0(\overline{a}_j)) > \\ &\quad Z_i + Z_j + M_0(a) + M_0(\overline{a}) + 2gcd_a + 4f_{Z_i,Z_j}(Z_c). \end{aligned}$$

By Lemma 11, the right hand side of this inequality is minimum for $f_{Z_i,Z_j}(Z_c) = -\gcd(Z_i, Z_j) = -gcd_a$. By selecting $Z_c = gcd_a$, we have moreover $f_{Z_i,Z_i}(Z_c) = f_{Z_j,Z_j}(Z_c) = -gcd_a$.

The first two inequalities become then $M_0(a_i) + M_0(\overline{a}_i) > Z_i - gcd_a$ and $M_0(a_j) + M_0(\overline{a}_j) > Z_j - gcd_a$, giving the first two conditions of the theorem. The last two conditions follow directly from the other pair of inequalities by substituting $-gcd_a$ for $f_{Z_i,Z_i}(Z_c)$, thus concluding the proof of the theorem. ■

5.3.6 Live minimum memory footprint computation

The following theorem presents a solution satisfying the conditions from Theorem 12 that minimizes the overall memory needed for the communication of a bounded buffer between two clusters.

Theorem 13. *Let $\mathcal{G}'_{sdf}$ be the SDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{sdf}$ by a communication task and its four associated arcs according to Figure 5.5. Let also*

$Z_c = \gcd_a$. *Setting* $\delta = \min(M_0(a), Z_j)$, *the values*

$$\begin{cases} M_0(a_i) = M_0(a) - \delta \\ M_0(\overline{a}_i) = \delta + \max(M_0(\overline{a}) - Z_j, Z_i - M_0(a)) \\ M_0(a_j) = \delta \\ M_0(\overline{a}_j) = Z_j - \delta \end{cases}$$

constitute a feasible solution minimizing the total memory required for communication.

Proof. We denote $\sigma_a = M_0(a) + M_0(\overline{a})$, $\sigma_{a_i} = M_0(a_i) + M_0(\overline{a}_i)$ and $\sigma_{a_j} = M_0(a_j) + M_0(\overline{a}_j)$. We easily check that M_0 satisfies the inequalities from Theorem 12. Two cases must be considered to minimize $\sigma_{a_i} + \sigma_{a_j}$:

1. If $M_0(\overline{a}) - Z_j \geq Z_i - M_0(a)$, then $\sigma_{a_i} + \sigma_{a_j} = \sigma_a$ and the last two inequalities are tight.
2. Otherwise, $\sigma_{a_i} + \sigma_{a_j} = Z_i + Z_j$ and the first two inequalities are tight.

The consequence is that $\sigma_{a_i} + \sigma_{a_j}$ is minimum. ■

Figure 5.7 illustrates a bounded buffer between two clusters with a minimum live marking according to Theorem 13. We have:

$$\begin{array}{l} Z_1 = 3, Z_2 = 2 \\ M_0(a) = 2, M_0(\overline{a}) = 3 \\ \delta = \min(M_0(a), Z_2) = 2 \\ \max(M_0(\overline{a}) - Z_2, Z_1 - M_0(a)) = 1 \end{array} \quad \text{thus} \begin{cases} M_0(a_1) = 0 \\ M_0(\overline{a}_1) = 3 \\ M_0(a_2) = 2 \\ M_0(\overline{a}_2) = 0 \end{cases}$$

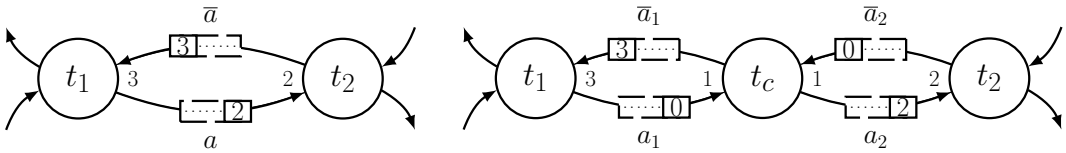


Figure 5.7 – (a) A bounded buffer $a = (t_1, t_2)$; (b) the same bounded buffer a between two clusters with a minimum live initial marking according to Theorem 13.

5.3.7 Throughput guarantee for an inter-cluster buffer

This section deals with the minimization of the memory footprint of an inter-cluster buffer with a throughput constraint. The throughput constraint uses a sufficient condition which comes from the one-periodic schedule characterization of Theorem 2 in Section 2.5.5. The one-periodic schedule is preferred over the K-periodic or the As Soon As Possible (ASAP) schedule as it can be computed efficiently and hence evaluated in a reasonable amount of time for the large dataflow graphs considered

in this thesis. We now refer to this theorem as a sufficient condition of throughput (**SCT**).

Assuming that $\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ is an SDFG with an initial marking sufficient to admit a one-periodic schedule of fixed period T, let $\mathcal{G}'_{sdf}$ be the graph $\mathcal{G}_{sdf}$ with the bounded buffer $(a, \overline{a})$ replaced by two bounded buffers $(a_i, \overline{a}_i)$ and $(a_j, \overline{a}_j)$. Let ℓ_c be the time needed to transfer Z_c data items between the two clusters.

We show that the initial marking of the SDFG obtained by adding a task t_c of weight $Z_c = gcd_a$ can be computed optimally to reach a one-periodic feasible schedule of period T. Theorem 14 presents a set of sufficient conditions ensuring that conditions **SCT** are fulfilled for a fixed period. We denote the reduced period $\widetilde{T} = \frac{T}{Z_i.R_i}, \forall t_i \in \mathcal{T}$.

Theorem 14. *Let $a = (t_i, t_j)$ and $\overline{a} = (t_j, t_i)$ be a bounded buffer in $\mathcal{G}_{sdf}$ and let $\mathcal{G}'_{sdf}$ be the SDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{sdf}$ by a communication task, t_c, with $Z_c = gcd_a$. Letting*

$$r_i = Z_i - Z_c + \left\lceil \frac{\ell_i + \ell_c}{\widetilde{T}} \right\rceil_{gcd_a},$$

$$u_a = M_0(a) + \left\lceil \frac{\ell_c}{\widetilde{T}} \right\rceil_{gcd_a},$$

where $\lceil x \rceil_z$ denotes the smallest multiple of z greater than x, and assuming that the initial marking of $\mathcal{G}'_{sdf}$ satisfies the following inequalities

$$\begin{cases} M_0(a_i) + M_0(\overline{a}_i) \geq r_i \\ M_0(a_j) + M_0(\overline{a}_j) \geq r_j \\ M_0(a_i) + M_0(a_j) \geq u_a \\ M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq u_{\overline{a}}, \end{cases}$$

then, the initial marking is feasible for a one-periodic schedule of reduced period $\widetilde{T}$ for $\mathcal{G}'_{sdf}$.

Proof. The four inequalities are proven separately.

- Cycle (t_i, t_c, t_i) must verify condition **SCT**, which is equivalent to $\ell_i + \ell_c - \widetilde{T}.(H(a_i) + H(\overline{a}_i)) \leq 0$ that is $H(a_i) + H(\overline{a}_i) \geq \frac{\ell_i + \ell_c}{\widetilde{T}}$.

  Now $H(a_i) + H(\overline{a}_i) = M_0(a_i) + M_0(\overline{a}_i) - Z_i + gcd_a$ so that, as $M_0(a_i)$ and $M_0(\overline{a}_i)$ are multiples of gcd_a, the first inequality holds. The second inequality can be proved similarly.

- Now, to every cycle μ of $\mathcal{G}$ passing through a is associated a unique cycle of $\mathcal{G}'$ passing through a_i and a_j. A sufficient condition to ensure **SCT** is then

  $$\ell_i - \widetilde{T}.H(a) \geq \ell_i + \ell_c - \widetilde{T}.(H(a_i) + H(a_j)),$$

that is $H(a_i) + H(a_j) \geq H(a) + \frac{\ell_c}{\widetilde{T}}$. Now, as $H(a) = M_0(a) - Z_j + gcd_a$ and

$$\begin{aligned} H(a_i) + H(a_j) &= M_0(a_i) + M_0(a_j) - Z_c - Z_j + 2.gcd_a \\ &= M_0(a_i) + M_0(a_j) - Z_j + gcd_a, \end{aligned}$$

we get

$$M_0(a_i) + M_0(a_j) \geq M_0(a) + \frac{\ell_c}{\widetilde{T}}.$$

Since all these markings are multiples of $Z_c = gcd_a$, the third inequality holds. The last inequality is proved using the same arguments. ■

Theorem 14 formalizes the guarantee of a throughput for a bounded buffer between two clusters. Note that the throughput guarantee ensures liveness; since $r_i > Z_i$ and $u_a > M_0(a)$. If the conditions of Theorem 14 are satisfied then the conditions of Theorem 12 are satisfied however large is the period T. The next section presents a solution for a minimum initial marking while ensuring a given throughput for a bounded buffer between two clusters.

5.3.8 Minimum memory footprint computation under throughput constraint

The following theorem is a characterization of a locally optimal initial marking ensuring the existence of a periodic schedule of period T for $\mathcal{G}'_{sdf}$.

Theorem 15. *Consider a bounded buffer of $\mathcal{G}_{sdf}$ with arc $a = (t_i, t_j)$ and backward arc $\overline{a} = (t_j, t_i)$. Let $\mathcal{G}'_{sdf}$ be the SDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{sdf}$ by a communication task t_c with $Z_c = gcd_a$. Setting $\delta = \min(u_a, r_j)$, the values*

$$\begin{cases} M_0(a_i) = u_a - \delta \\ M_0(\overline{a}_i) = \delta + \max(u_{\overline{a}} - r_j, r_i - u_a) \\ M_0(a_j) = \delta \\ M_0(\overline{a}_j) = r_j - \delta \end{cases}$$

are a solution ensuring the existence of a periodic schedule of period T minimizing the total memory $\sigma_{a_i} + \sigma_{a_j}$ required for the communication.

Proof. The proof uses Theorem 14. As it is similar to that of Theorem 13 it is omitted. ■

We now illustrate the throughput constraint property with the example of Figure 5.8 where $\ell_1 = \ell_2 = 1$. Without considering the rest of the SDFG this bounded buffer is able to comply with a one-periodic schedule with period $T = 6$.

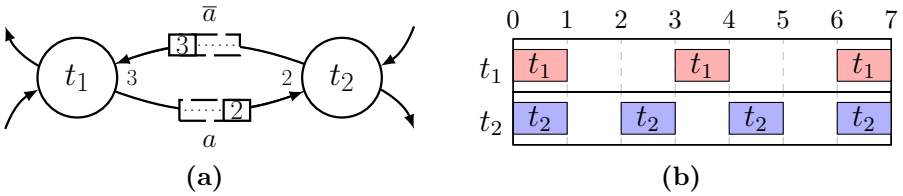


Figure 5.8 – (a) A bounded buffer; (b) One-periodic schedule for the bounded buffer with period $T = 6$.

Assume that $\ell_c = 1$. We have $T = 6$ and $Z_1.R_1 = Z_2.R_2 = 6$ thus $\widetilde{T} = 1$. For the bounded buffer between two clusters, Theorem 15 gives:

$$
\begin{aligned}
r_1 &= 3 - 1 + 2 = 4 \\
r_2 &= 2 - 1 + 2 = 3 \\
u_a &= 2 + 1 = 3 \\
u_{\overline{a}} &= 3 + 1 = 4 \\
\delta &= \min(3, 3) = 3
\end{aligned}
\qquad \text{yielding to:} \quad
\begin{cases}
M_0(a_1) = 0 \\
M_0(\overline{a}_1) = 4 \\
M_0(a_2) = 3 \\
M_0(\overline{a}_2) = 0
\end{cases}
$$

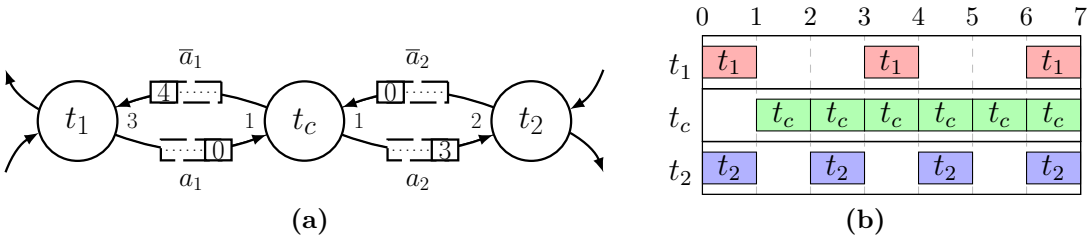


Figure 5.9 – (a) The bounded buffer $a = (t_1, t_2)$ of Figure 5.8(a) when distributed between two clusters. (b) One-periodic schedule for the bounded buffer between two clusters with period $T = 6$.

Figures 5.8 and 5.9 show how to compute an initial marking while ensuring liveness and throughput for a bounded buffer between two clusters. As pictured in Figure 5.8(b) and 5.9(b) the throughput is maintained.

5.4 Memory evaluation for a CSDFG model

This section extends the liveness guarantee and the minimal throughput guarantee for a bounded buffer to the CSDFG model. The extension of the memory evaluation to the CSDFG model uses the same technique as for the SDFG model.

Section 5.4.1 extends the notion of height to the CSDFG model. Section 5.4.2 formalizes the liveness problem for a single bounded buffer in a CSDFG. Section 5.4.3 determines the optimal transfer rate for the communication task to minimize the live marking of the bounded buffer. Section 5.4.4 determines the minimum live

initial marking for a throughput guarantee of a CSDFG bounded buffer. Section 5.4.5 formalizes the throughput guarantee problem for a bounded buffer and, finally, Section 5.4.6 determines an initial marking with a throughput guarantee.

5.4.1 Height of a CSDFG

The height notation is extended to the CSDFG model. Let $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ be a normalized CSDFG. There are $\varphi_i \times \varphi_j$ heights for an arc $a = (t_i, t_j)$, denoted $H_{k_i,k_j}(a) = M_0(a) + P_a^{-1}(k_i, 1) - C_a(k_j, 1) + step_a$ with $k_i \in \{1, \cdots \varphi_i\}$ and $k_j \in \{1, \cdots \varphi_j\}$.

The sufficient condition of liveness (**SCL1**) expressed by Theorem 4 in Section 3.2.6 is equivalent to $\sum_{a\in\mu} H_{k_i,k_j}(a) > 0$, $\forall k_i \in \{1, \cdots \varphi_i\}$, $\forall k_j \in \{1, \cdots \varphi_j\}$, $\forall a = (t_i, t_j) \in \mu$ for every cycle μ.

5.4.2 Liveness guarantee for an inter-cluster buffer

This section extends the liveness guarantee for an inter-cluster buffer to the CSDFG model. Let $\mathcal{G}_{csdf}$ be a bounded CSDFG that satisfies the sufficient condition of liveness **SCL1**. Suppose that $a = (t_i, t_j)$ is a bounded buffer such that t_i and t_j are in different clusters.

Lemma 12. *Let $\mathcal{G}'_{csdf}$ be the CSDFG obtained by replacing an arc a and the associated backward $\overline{a}$ in $\mathcal{G}_{csdf}$ by a communication task and its four associated arcs according to Figure 5.5. If $\mathcal{G}_{csdf}$ verifies **SCL1** and the following conditions*

$$\begin{cases} H_{k_i,k_c}(a_i) + H_{k_c,k_i}(\overline{a}_i) > 0 \\ H_{k_c,k_j}(a_j) + H_{k_j,k_c}(\overline{a}_j) > 0 \\ H_{k_i,k_c}(a_i) + H_{k_c,k_j}(a_j) \geq H_{k_i,k_j}(a) \\ H_{k_c,k_i}(\overline{a}_i) + H_{k_j,k_c}(\overline{a}_j) \geq H_{k_j,k_i}(\overline{a}) \end{cases}$$

*$\forall k_i \in \{1, \cdots, \varphi_i\}$, $\forall k_j \in \{1, \cdots, \varphi_j\}$ and $\forall k_c \in \{1, \cdots, \varphi_c\}$, then $\mathcal{G}'_{csdf}$ verifies **SCL1**.*

Proof. If $\mathcal{G}_{csdf}$ verifies **SCL1**, a sufficient condition of liveness of $\mathcal{G}'_{csdf}$ is that the following four additional conditions be met $\forall k_i \in \{1, \cdots, \varphi_i\}$, $\forall k_j \in \{1, \cdots, \varphi_j\}$ and $\forall k_c \in \{1, \cdots, \varphi_c\}$:

- Cycles (t_i, t_c, t_i) and (t_j, t_c, t_j) of $\mathcal{G}'_{csdf}$ must verify **SCL1**, thus $H_{k_i,k_c}(a_i) + H_{k_c,k_i}(\overline{a}_i)$ and $H_{k_c,k_j}(a_j) + H_{k_j,k_c}(\overline{a}_j) > 0$.
- To every cycle μ of $\mathcal{G}_{csdf}$ passing through a (resp. $\overline{a}$) is associated a unique cycle of $\mathcal{G}'_{csdf}$ passing through a_i and a_j (resp. $\overline{a}_i$ and $\overline{a}_j$). **SCL1** is then ensured by imposing $H_{k_i,k_c}(a_i) + H_{k_c,k_j}(a_j) \geq H_{k_i,k_j}(a)$ and $H_{k_c,k_i}(\overline{a}_i) + H_{k_j,k_c}(\overline{a}_j) \geq H_{k_j,k_i}(\overline{a})$.

■

5.4.3 Optimal transfer rate for a live minimum memory footprint

The next theorem shows that the transfer rate $r_c = step_a$ with $\varphi_c = 1$ is optimal in order to minimize the live marking of the shared bounded buffer. Note that task t_c models the data transfer through the NoC, therefore consumption and production vectors on each arc are identical.

Theorem 16. *For any bounded buffer $a = (t_i, t_j)$ between two clusters, the value $r_c = step_a$ minimizes the total memory required for the buffer according to the sufficient condition of liveness **SCL1**. The conditions expressed by Lemma 12 become then*

$$\begin{cases} M_0(a_i) + M_0(\overline{a}_i) \geq \max_{k\in\{1,\cdots,\varphi_i\}}(c_{\overline{a}_i}(k)) \\ M_0(a_j) + M_0(\overline{a}_j) \geq \max_{k\in\{1,\cdots,\varphi_j\}}(c_{a_j}(k)) \\ M_0(a_i) + M_0(a_j) \geq M_0(a) \\ M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq M_0(\overline{a})\,. \end{cases}$$

Proof. Replacing H by its definition, the first two inequalities of Lemma 12 give:

$$\begin{aligned} &M_0(a_i) + M_0(\overline{a}_i) + P^{-1}_{a_i}(k_i,1) + P^{-1}_{\overline{a}_i}(k_c,1) - C_{a_i}(k_c,1) - C_{\overline{a}_i}(k_i,1) + 2step_{a_i} > 0 \\ &M_0(a_j) + M_0(\overline{a}_j) + P^{-1}_{a_j}(k_c,1) + P^{-1}_{\overline{a}_j}(k_j,1) - C_{a_j}(k_j,1) - C_{\overline{a}_j}(k_c,1) + 2step_{a_j} > 0. \end{aligned}$$

Since $P_{a_i}(k_i,1) = C_{\overline{a}_i}(k_i,1)$, $P^{-1}_{a_i}(k_i,1) - C_{\overline{a}_i}(k_i,1) = -c_{\overline{a}_i}(k_i)$and,
since $P_{\overline{a}_i}(k_c,1) = C_{a_i}(k_c,1)$, $P^{-1}_{\overline{a}_i}(k_c,1) - C_{a_i}(k_c,1) = -c_{a_i}(k_c)$.

Similarly

$$\begin{aligned} &C_{a_j}(k_j,1) = P_{\overline{a}_j}(k_j,1) \text{ implies } P^{-1}_{\overline{a}_j}(k_j,1) - C_{a_j}(k_j,1) = -c_{a_j}(k_j) \text{ and,} \\ &P_{a_j}(k_c,1) = C_{\overline{a}_j}(k_c,1) \text{ implies } P^{-1}_{a_j}(k_c,1) - C_{\overline{a}_j}(k_c,1) = -c_{\overline{a}_j}(k_c). \end{aligned}$$

Using these relations the first two inequalities of Lemma 12 become:

$$\begin{aligned} &M_0(a_i) + M_0(\overline{a}_i) > c_{\overline{a}_i}(k_i) + c_{a_i}(k_c) - 2step_{a_i}, \\ &M_0(a_j) + M_0(\overline{a}_j) > c_{a_j}(k_j) + c_{\overline{a}_j}(k_c) - 2step_{a_j}. \end{aligned}$$

Now, replacing H by its definition, the last two inequalities of Lemma 12 become:

$$\begin{aligned} M_0(a_i) + M_0(a_j) + P^{-1}_{a_i}(k_i,1) + P^{-1}_{a_j}(k_c,1) - C_{a_i}(k_c,1) - C_{a_j}(k_j,1) \\ + step_{a_i} + step_{a_j} \geq M_0(a) + P^{-1}_{a}(k_i,1) - C_{a}(k_j,1) + step_a \\ M_0(\overline{a}_i) + M_0(\overline{a}_j) + P^{-1}_{\overline{a}_j}(k_j,1) + P^{-1}_{\overline{a}_i}(k_c,1) - C_{\overline{a}_j}(k_c,1) - C_{\overline{a}_i}(k_i,1) \\ + step_{\overline{a}_i} + step_{\overline{a}_j} \geq M_0(\overline{a}) + P^{-1}_{\overline{a}}(k_j,1) - C_{\overline{a}}(k_i,1) + step_{\overline{a}}. \end{aligned}$$

Since $C_{a_i}(k_c,1) = P_{a_j}(k_c,1)$, $P^{-1}_{a_j}(k_c,1) - C_{a_i}(k_c,1) = -c_{a_i}(k_c)$ and,
since $P_{\overline{a}_i}(k_c,1) = C_{\overline{a}_j}(k_c,1)$, $P^{-1}_{\overline{a}_i}(k_c,1) - C_{\overline{a}_j}(k_c,1) = -c_{\overline{a}_j}(k_c)$.

Since moreover

$$P_a^{-1}(k_i,1) = P_{a_i}^{-1}(k_i,1),\ C_a(k_j,1) = C_{a_j}(k_j,1) \text{ and,}$$
$$P_{\overline{a}}^{-1}(k_j,1) = P_{\overline{a}_j}^{-1}(k_j,1),\ C_{\overline{a}}(k_i,1) = C_{\overline{a}_i}(k_i,1),$$

the last two inequalities of Lemma 12 become:

$$M_0(a_i) + M_0(a_j) \geq M_0(a) + step_a + c_{a_i}(k_c) - step_{a_i} - step_{a_j} \text{ and,}$$
$$M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq M_0(\overline{a}) + step_{\overline{a}} + c_{\overline{a}_j}(k_c) - step_{\overline{a}_i} - step_{\overline{a}_j}.$$

Denoting $g_i = gcd(p_{a_i}(1), \cdots, p_{a_i}(\varphi_i))$ and $g_j = gcd(c_{a_j}(1), \cdots, c_{a_j}(\varphi_j))$, by applying Lemma 11 we obtain:

$$\begin{cases} M_0(a_i) + M_0(\overline{a}_i) > c_{\overline{a}_i}(k_i) + f_{g_i,g_i}(c_{a_i}(k_c)) \\ M_0(a_j) + M_0(\overline{a}_j) > c_{a_j}(k_j) + f_{g_j,g_j}(c_{\overline{a}_j}(k_c)) \\ M_0(a_i) + M_0(a_j) \geq M_0(a) + step_a + f_{g_i,g_j}(c_{a_i}(k_c)) \\ M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq M_0(\overline{a}) + step_{\overline{a}} + f_{g_i,g_j}(c_{\overline{a}_j}(k_c)). \end{cases}$$

Following the same logic as for Theorem 12 we find that $c_{a_i}(k_c) = c_{\overline{a}_j}(k_c) = step_a$ with $\varphi_c = 1$ is the optimal solution for minimizing the initial marking of the bounded buffer, which gives the theorem. ■

5.4.4 Minimum live memory footprint evaluation

This section extends the live minimal memory footprint results of the SDFG model to the CSDFG model. The proof is omitted since it is similar to the one for the SDFG model.

Theorem 17. *Let $\mathcal{G}'_{csdf}$ be the CSDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{csdf}$ by a communication task and its four associated arcs according to Figure 5.5. The transfer rate of task t_c is $[step_a]$. Denoting $max_{a_i} = \max_{k\in\{1,\cdots,\varphi_i\}}(c_{a_i}(k))$ and setting $\delta = \min(M_0(a), max_{a_j})$, the values*

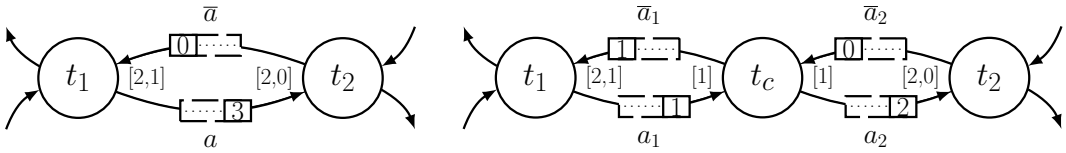
$$\begin{cases} M_0(a_i) = M_0(a) - \delta \\ M_0(\overline{a}_i) = \delta + \max(M_0(\overline{a}) - max_{a_j}, max_{\overline{a}_i} - M_0(a)) \\ M_0(a_j) = \delta \\ M_0(\overline{a}_j) = max_{a_j} - \delta \end{cases}$$

constitute a feasible solution minimizing the total memory required for communication.

Figure 5.10 illustrates a bounded buffer between two clusters with a minimum

live marking according to Theorem 17. We have:

$$\begin{array}{l} M_0(a) = 3 \\ \max_{\overline{a}_1} = 2 \\ \max_{a_2} = 2 \\ \delta = \min(M_0(a), max_{a_2}) = 2 \\ \max(M_0(\overline{a}) - max_{a_2}, max_{\overline{a}_1} - M_0(a)) = -1 \end{array} \quad \text{which gives:} \begin{cases} M_0(a_1) = 1 \\ M_0(\overline{a}_1) = 1 \\ M_0(a_2) = 2 \\ M_0(\overline{a}_2) = 0 \end{cases}$$



(a) **(b)**

Figure 5.10 – (a) A bounded buffer $a = (t_1, t_2)$; (b) the same bounded buffer a distributed between two clusters with a minimum live initial marking according to Theorem 17.

5.4.5 Throughput guarantee for an inter-cluster buffer

This section extends the throughput guarantee for an inter-cluster buffer to the CSDFG model. The throughput guarantee uses the one-periodic schedule already defined in Theorem 6 of Section 3.3.4. We now refer to this theorem as **SCT**.

Theorem 18 presents a set of sufficient conditions ensuring that conditions **SCT** are fulfilled for the fixed period. We denote the reduced period $\widetilde{T} = \frac{T}{p_a.R_i}$ with $a = (t_i, \cdot)$. We recall that $\frac{T}{p_a.R_i} = \frac{T}{c_a.R_j}$, $\forall a = (i, j) \in \mathcal{A}$.

Assuming that $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ is a CSDFG with an initial marking sufficient to build a one-periodic schedule of fixed period T, let $\mathcal{G}'_{csdf}$ be the graph with the bounded buffer $(a, \overline{a})$ in $\mathcal{G}_{csdf}$ replaced by two bounded buffers $(a_i, \overline{a}_i)$ and $(a_j, \overline{a}_j)$. Let ℓ_c be the time needed to transfer $step_a$ data items between the two clusters.

Lemma 13. *Let $\mathcal{G}'_{csdf}$ be the CSDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{csdf}$ by a communication task and its four associated arcs according to Figure 5.5. If $\mathcal{G}_{csdf}$*

verifies ***SCT*** *and the following conditions*

$$\begin{cases} H_{k_i,k_c}(a_i) + H_{k_c,k_i}(\overline{a}_i) \geq \left\lceil \dfrac{\ell_i(k_i) + \ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a} & (5.1) \\ H_{k_c,k_j}(a_j) + H_{k_j,k_c}(\overline{a}_j) \geq \left\lceil \dfrac{\ell_j(k_j) + \ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a} & (5.2) \\ H_{k_i,k_c}(a_i) + H_{k_c,k_j}(a_j) \geq H_{k_i,k_j}(a) + \left\lceil \dfrac{\ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a} & (5.3) \\ H_{k_c,k_i}(\overline{a}_i) + H_{k_j,k_c}(\overline{a}_j) \geq H_{k_j,k_i}(\overline{a}) + \left\lceil \dfrac{\ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a} & (5.4) \end{cases}$$

$\forall k_i \in \{1, \cdots, \varphi_i\}$, $\forall k_j \in \{1, \cdots, \varphi_j\}$ *and* $\forall k_c \in \{1, \cdots, \varphi_c\}$, *then* $\mathcal{G}'_{csdf}$ *verifies* ***SCT***.

Proof. The conditions are derived separately. We recall the notation

$$\alpha_a^{max}(k_i, k_j) = \lfloor M_0(a) + P_a^{-1}(k_i, 1) - C_a(k_j, 1) + 1 \rfloor_{gcd_a}.$$

- By applying **SCT** on cycle (t_i, t_c, t_i) we obtain:

  $$\alpha_{a_i}^{max}(k_i, k_c) + \alpha_{\overline{a}_i}^{max}(k_c, k_i) \geq \frac{\ell_i(k_i) + \ell_c(k_c)}{\widetilde{T}}.$$

  The first condition follows since $H_{k_i,k_j}(a) \geq \alpha_a^{max}(k_i, k_j)$, $\forall a = (t_i, t_j) \in \mathcal{A}$, and since the initial marking is assumed to be composed only of useful tokens so that the right hand side of the inequality can be rounded up to the closest multiple of $step_a$.

  The second condition is obtained similarly.

- To every cycle μ passing through a is associated a unique cycle of $\mathcal{G}'_{csdf}$ passing through a_i and a_j. A sufficient condition to ensure **SCT** is:

  $$\ell_i(k_i) - \widetilde{T}.\alpha_a^{max}(k_i, k_j) \geq \ell_i(k_i) + \ell_c(k_c) - \widetilde{T}(\alpha_{a_i}^{max}(k_i, k_c) + \alpha_{a_j}^{max}(k_c, k_j)),$$

  which is equivalent to $\alpha_{a_i}^{max}(k_i, k_c) + \alpha_{a_j}^{max}(k_c, k_j) \geq \alpha_a^{max}(k_i, k_j) + \frac{\ell_c(k_c)}{\widetilde{T}}$.

  The third condition follows since $H_{k_i,k_j}(a) \geq \alpha_a^{max}(k_i, k_j)$, $\forall a = (i, j) \in \mathcal{A}$, and since the initial marking is assumed to be composed only of useful tokens so that the right hand side of the inequality can be rounded up to the closest multiple of $step_a$.

  The fourth inequality is proved similarly. ■

The following theorem builds upon Lemma 13 to provide conditions on the initial markings.

Theorem 18. *Let* $\mathcal{G}'_{csdf}$ *be the CSDFG obtained by replacing* a *and* $\overline{a}$ *in* $\mathcal{G}_{csdf}$ *by a*

communication task and its four associated arcs according to Figure 5.5. We denote

$$r_i(k_i) = p_a(k_i) + step_a + \left\lceil \frac{\ell_i(k_i) + \ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a}$$

$$u_a(k_i) = M_0(a) + \left\lceil \frac{\ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a}$$

and

$$r_i^m = \max_{k_i \in \{1,\cdots,\varphi_i\}} r_i(k_i)$$
$$u_a^m = \max_{k_i \in \{1,\cdots,\varphi_i\}} u_a(k_i).$$

*If $\mathcal{G}_{csdf}$ verifies **SCT** and the following four conditions*

$$\begin{cases} M_0(a_i) + M_0(\overline{a}_i) \geq r_i^m & (5.5) \\ M_0(a_j) + M_0(\overline{a}_j) \geq r_j^m & (5.6) \\ M_0(a_i) + M_0(a_j) \geq u_a^m & (5.7) \\ M_0(\overline{a}_i) + M_0(\overline{a}_j) \geq u_{\overline{a}}^m, & (5.8) \end{cases}$$

*then $\mathcal{G}'_{csdf}$ verifies **SCT**.*

Proof. The conditions are derived separately.

- Since $P_{a_i}^{-1}(k_i, 1) - C_{\overline{a}_i}(k_i, 1) = -p_{a_i}(k_i)$, $P_{\overline{a}_i}^{-1}(k_c, 1) = 0$ and $C_{a_i}(k_c, 1) = step_a$, condition (5.1) may be rewritten as

  $$M_0(a_i) + M_0(\overline{a}_i) - p_{a_i}(k_1) + step_a \geq \left\lceil \frac{\ell_i(k_i) + \ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a},$$

  which gives the expression (5.5). Expression 5.6 is obtained similarly.

- Since $P_{a_i}^{-1}(k_i, 1) = P_a^{-1}(k_i, 1)$ and $C_{a_j}(k_j, 1) = C_a(k_j, 1)$, condition (5.7) may be rewritten as

  $$M_0(a_i) + M_0(a_j) + step_a \geq M_0(a) + step_a + \left\lceil \frac{\ell_c(k_c)}{\widetilde{T}} \right\rceil_{step_a},$$

  which gives the expression (5.7). Expression (5.8) is obtained similarly. ■

5.4.6 Minimum memory footprint computation under a throughput constraint

We now compute an initial marking to ensure the throughput of a CSDFG when mapped on a distributed architecture.

Theorem 19. *Consider a bounded buffer of a CSDFG $\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$ with arc $a = (t_i, t_j)$ and backward arc $\overline{a} = (t_j, t_i)$. Let $\mathcal{G}'_{csdf}$ be the CSDFG obtained by replacing a and $\overline{a}$ in $\mathcal{G}_{csdf}$ by a communication task t_c with $r_c = \lceil step_a \rceil$.*

Setting $\delta = \min(u_a^m, r_j^m)$*, the values*

$$\begin{cases} M_0(a_i) = u_a^m - \delta \\ M_0(\overline{a}_i) = \delta + \max(u_{\overline{a}}^m - r_j^m, r_i^m - u_a^m) \\ M_0(a_j) = \delta \\ M_0(\overline{a}_j) = r_j^m - \delta \end{cases}$$

are a solution ensuring the existence of a periodic schedule of period T *minimizing the total memory* $\sigma_{a_i} + \sigma_{a_j}$ *required for the communication.*

Proof. The proof uses Theorem 18. As it is similar to that of Theorem 17 it is omitted. ■

We now illustrate the throughput constraint property on the example of Figure 5.11 with $\ell_1 = \ell_2 = [1, 1]$. Without considering the rest of the SDFG this bounded buffer is able to reach a one-periodic schedule with a period $T = 12$ as illustrated in Figure 5.8(b).

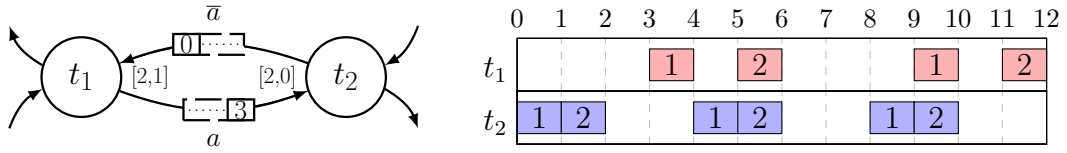


Figure 5.11 – (a) A bounder buffer; (b) the one-periodic schedule of the bounded buffer with a period $T = 12$.

We assume $\ell_c = [1]$. Since $T = 12$ and $p_a.R_1 = c_a.R_2 = 6$, we have $\widetilde{T} = 2$. For the bounded buffer between two clusters Theorem 15 gives:

$$\begin{array}{l} r_1^m = 4 \\ r_2^m = 4 \\ u_a^m = 4 \\ u_{\overline{a}}^m = 2 \\ \delta = \min(4, 4) = 4 \end{array} \qquad \text{the initial marking: } \begin{cases} M_0(a_1) = 0 \\ M_0(\overline{a}_1) = 2 \\ M_0(a_2) = 4 \\ M_0(\overline{a}_2) = 0 \end{cases}.$$

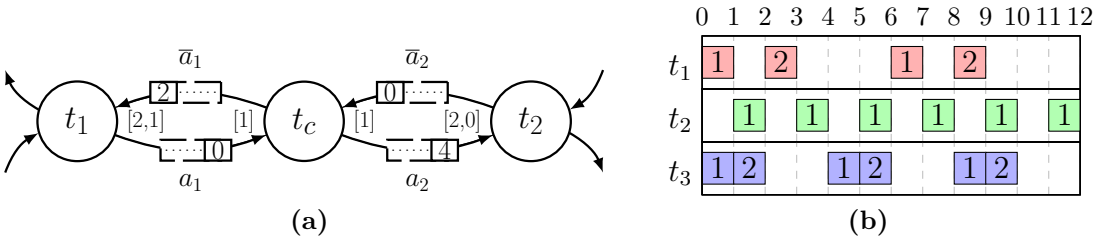


Figure 5.12 – (a) The bounded buffer $a = (t_1, t_2)$ of Figure 5.11(a) when distributed between two clusters; (b) the one-periodic schedule of the bounded buffer between the two clusters with a period $T = 12$.

Figures 5.11 and 5.12 illustrate the computation of an initial marking while ensuring liveness and maintaining a throughput for a bounded buffer between two clusters. As shown in Figures 5.11(b) and 5.12(b) the throughput is maintained.

5.5 Conclusion

This chapter introduced a new method to evaluate the memory consumption of a communication between two tasks. One token of the initial marking on an arc or the associated backward arc represents a data item or a memory space of the same size dedicated to a communication. The targeted architecture is composed of many processing elements grouped in clusters with shared memory linked by a NoC. On this architecture, two types of communications can be distinguished whether two tasks are located in the same cluster or not.

To evaluate communicating memory a backward arc is introduced for each arc of the application to bound the memory consumption of a communication. A communication task is explicitly inserted between two tasks to model the communication through the NoC. This insertion requires a re-evaluation of the initial marking between the two tasks. A sufficient condition of liveness is used to ensure liveness while the one-periodic schedule is used to guarantee satisfaction of a throughput constraint.

All these techniques have been detailed on both SDFG and CSDFG models. The next chapter applies these techniques to quickly evaluate memory usage while ensuring liveness or satisfaction of a throughput constraint with different mapping algorithms and evaluates the performance and the scalability of the approach with experiments.

Chapter 6

Algorithms for the mapping problem

Contents

6.1 Introduction

This chapter concludes the contributions of this thesis and presents an approach to solve the mapping problem described in Section 5.2.1. The approach builds upon the buffer memory footprint evaluation method for SDFGs and CSDFGs presented in Section 5.3 and 5.4. It uses an undirected multivalued graph to model the memory consumption of every buffer in an application graph. This multivalued graph lets make allocations of tasks using a fast memory footprint evaluation method. Liveness and throughput constrained problems are both tackled with the same techniques.

In Section 6.2 the problem of preserving the functional behavior when performing memory footprint evaluation is discussed. Section 6.3 introduces the undirected multivalued graph used to evaluate the memory consumption associated to a mapping. Section 6.4 illustrates memory evaluation using the multivalued graph on the H263 encoder application. The NP-completeness of the problem is proven in Section 6.6. A brief state of the art on mapping resolution techniques is exposed in Section 6.7. Section 6.7 introduces several algorithms to solve the mapping problem. Section 6.9 experiments the mapping algorithms on real life and randomly generated applications.

6.2 Functionality preservation

In order for the implementation of the application to work as specified, functionality must be preserved when refining and mapping dataflow system models.

Normalization, replacing one token by N_a (virtual) tokens manipulated as a block for the purpose of analysis, does not affect system functionality. The addition of the communication task t_c and the backward arcs with the markings selected according to Theorem 13 in Section 5.3.6 preserves functionality as the precedence constraints are preserved and the initial marking $M_0(a)$ is merely decomposed into $M_0(a_1) + M_0(a_2) = M_0(a)$.

In the mapping problem where the memory required is evaluated under a throughput constraint, the initial marking will not be preserved when two tasks belonging to a cycle are mapped on two different clusters, as $M_0(a_1)+M_0(a_2) > M_0(a)$. While tokens (items) will just have to be added on the paths incident to t_1 or t_2 that do not belong to any cycle to preserve functionality, preserving (global) functionality will require extra work for the tasks in a cycle containing a, typically the (local) modification of the function of one of these tasks. For a signal processing application, the function modification may be to employ look-ahead computation (for recursive filtering) [Parhi, 1995]; for a control application, the function modification may amount to modifying some of its parameters (for a controller). Also note that if latency matters between some tasks, one may impose that these tasks be on the same cluster and evaluate memory under that constraint.

6.3 Multivalued undirected graph representation

The multivalued undirected graph is deduced from an SDFG or a CSDFG $\mathcal{G}$ as follows. The set of tasks $\mathcal{T}$ remains the same for both graphs and the set of arcs $\mathcal{A}$ is simplified into a set of edges where each edge corresponds to a bounded buffer. The graph is denoted $\mathcal{U} = (\mathcal{T}, \mathcal{E})$. To any edge $e = \{t_i, t_j\} \in \mathcal{E}$ are associated three weights w_e, w_{ei} and w_{ej}.

1. w_e is the size of the corresponding buffer if t_i and t_j are in the same cluster, thus $w_e = \sigma_a$;
2. w_{ei} and w_{ej} are the amounts of memory in each cluster for the communication between t_i and t_j when the two tasks are on different clusters. We set $w_{ei} = \sigma_{a_i}$ and $w_{ej} = \sigma_{a_j}$ following Theorem 13 of Section 5.3.6 or Theorem 15 of Section 5.3.8 for the SDFG model, or Theorem 17 of Section 5.4.4 or Theorem 19 of Section 5.4.6 for the CSDFG model, depending on the optimization problem considered.

The transformation implies that, for every bounded buffer of $\mathcal{G}$, an equivalent bounded buffer between two clusters is evaluated and the associated initial marking is computed.

Let m be a mapping of $\mathcal{T}$ to the set of clusters $\mathcal{S}_c$. Setting $\mathcal{E}_c = \{e = \{t_i, t_j\} \in \mathcal{E}, m(t_i) = m(t_j) = c\}$ and $\mathcal{E}'_c = \{e = \{t_i, t_j\} \in \mathcal{E}, m(t_i) = c, m(t_j) \neq c\}$, the total memory needed for storage in any cluster $c \in \mathcal{S}_c$ is

$$g(m, c) = \sum_{e \in \mathcal{E}_c} w_e + \sum_{e=(t_i,\cdot) \in \mathcal{E}'_c} w_{ei}.$$

The total memory for the mapping is then $\sum_{s \in \mathcal{S}_c} g(m, c)$.

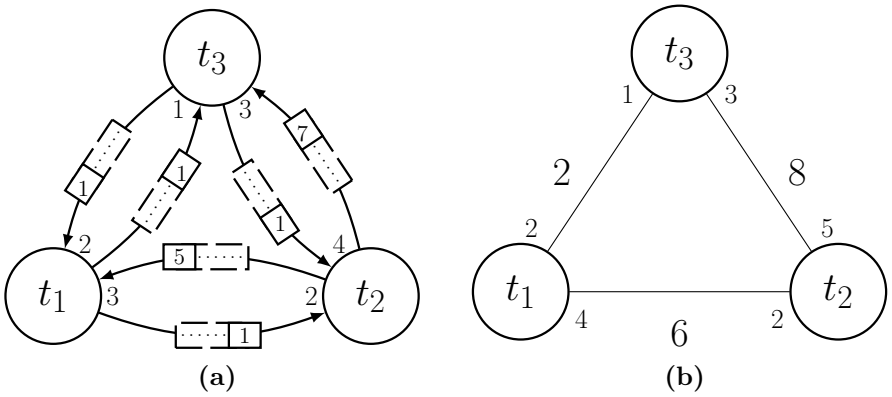


Figure 6.1 – (a) A bounded SDFG; (b) the associated graph $\mathcal{U}$.

Figure 6.1 gives an example of a graph $\mathcal{U}$ obtained from a bounded SDFG for the liveness problem **SCL**. The weights at the extremities of an edge $e = (t_i, t_j)$ correspond to w_{ei} and w_{ej}, the weight in the center is w_e. Note that values on the edge of graph $\mathcal{U}$ come from unnormalized values computed from the bounded and

normalized version of the graph of Figure 6.1(a).

Consider a mapping on two clusters c_1 and c_2 such as $m(t_1) = m(t_2) = c_1$ and $m(t_3) = c_2$. The amounts of memory consumed are $g(m, c_1) = 13$ in cluster c_1 and $g(m, c_2) = 4$ in cluster c_2. The total memory used by the mapping is $g(m, c_1) + g(m, c_2) = 17$.

6.4 Illustration with an H263 encoder

This section illustrates the computation of the memory footprint using the multivalued undirected graph $\mathcal{U}$. We choose a simple application: an H263 video encoder [Oh and Ha, 2002]. The encoder is modeled by the CSDFG of Figure 6.2, already encountered in Section 2.3.1, which comprises 8 tasks: Read From Device (RFD), Motion Estimation (ME), Distributor (Dist), Motion Block Encoding (MBE), Motion Block Decoding (MBD), Motion Compensation (MC), Variable Length Coding (VLC) and Write To Device (WTD), respectively numbered from 1 to 8. The graph repetition vector is $R = [1, 1, 1, 99, 99, 1, 1, 1]$.

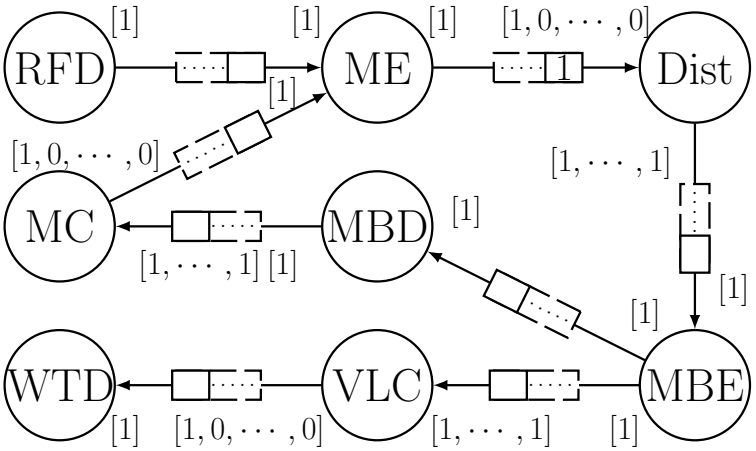


Figure 6.2 – 8-node CSDFG modeling the H263 encoder where the vectors $[1, 0, \cdots, 0]$ and $[1, \cdots, 1]$ have 99 elements.

In Figure 6.3 the CSDFG from Figure 6.2 is bounded and its initial marking ensures liveness.

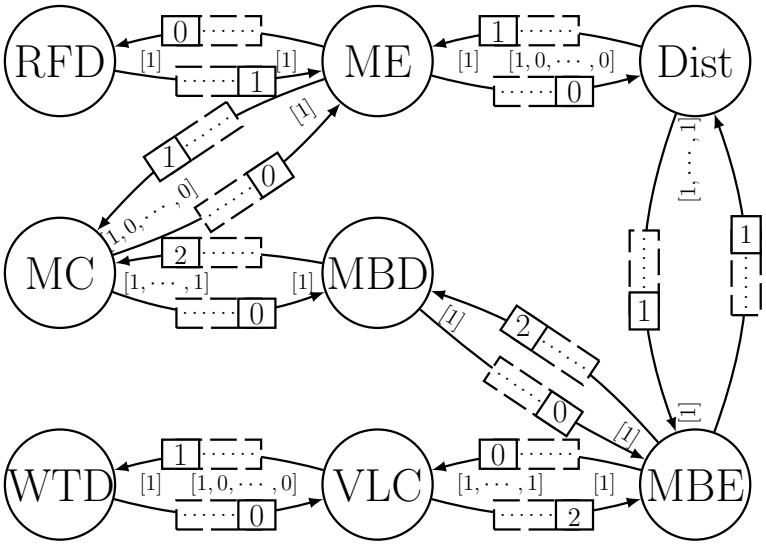


Figure 6.3 – Bounded CSDFG model of the H263 encoder where the vectors $[1, 0, \cdots, 0]$ and $[1, \cdots, 1]$ have 99 elements.

Figure 6.4 is the graph $\mathcal{U}$ deduced from the bounded CSDFG of Figure 6.3 by using Theorem 17 to compute the edge weights. As explained in Section 6.3, for an edge $e = (t_i, t_j)$, the weight next to t_i is $M_0(a_{t_i}) + M_0(\overline{a}_{t_i})$, the one next to t_j is $M_0(a_{t_j}) + M_0(\overline{a}_{t_j})$, and the one in-between is $M_0(a) + M_0(\overline{a})$, from the original bounded arc in Figure 6.3.

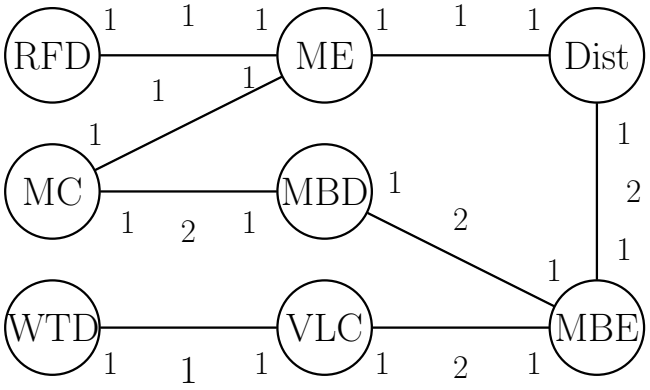


Figure 6.4 – Graph $\mathcal{U}$ deduced from Figure 6.3.

The memory consumption associated to mapping can be evaluated easily using graph $\mathcal{U}$. Figure 6.5 shows a mapping of tasks RFD, ME and Dist on cluster 1, MBD and MC on cluster 2 and MBE, VLC and WTD on cluster 3. The memory consumption in each cluster is the sum of the values in bold of the corresponding color: green for cluster 1, blue for cluster 2 and red for cluster 3.

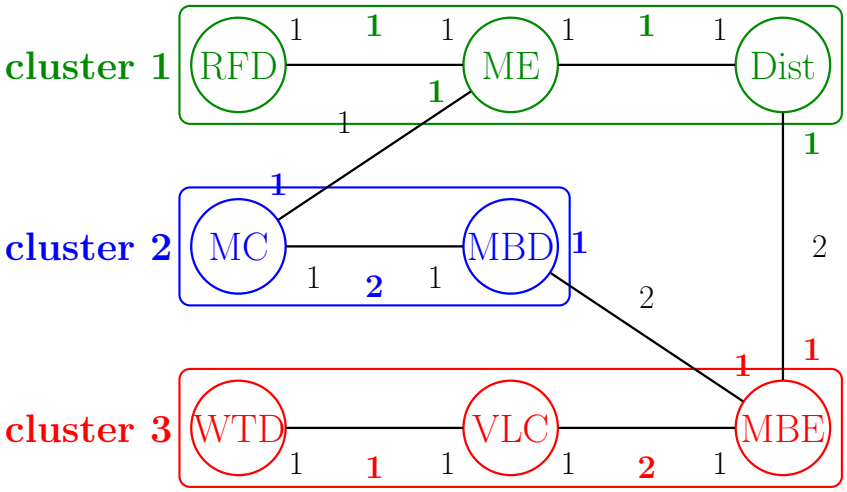


Figure 6.5 – Mapping using $\mathcal{U}$ of the H263 encoder.

The mapping memory consumption is thus: 4 for cluster 1, 4 for cluster 2 and 5 for cluster 3, giving a total memory consumption of 13. Using $\mathcal{U}$, any other mapping can be evaluated in a very short amount of time.

6.5 Formalization of the mapping problem

This section proposes a formal definition of the mapping problem using integer-linear programming. Let $\mathcal{U} = (\mathcal{T}, \mathcal{E})$ be a multivalued undirected graph and $\mathcal{S}$ be the set of clusters. The mapping problem may be expressed as the following optimization problem:

$$\min \sum_{c \in \mathcal{S}_c} z_c$$

such that

$$\begin{aligned}
\sum_{e \in \mathcal{E}} \left(y_{ec}.w_e + \sum_{t_i \in e} (x_{t_i c} - y_{ec}).w_{ei} \right) &\leq M_{max} && \forall c \in \mathcal{S}_c \\
y_{ec} &\geq x_{t_i c} + x_{t_j c} - 1 && \forall e = (t_i, t_j) \in \mathcal{E},\ \forall c \in \mathcal{S}_c, \\
y_{ec} &\leq x_{t_i c} && \forall e \in \mathcal{E},\ \forall t_i \in \mathcal{T},\ \forall c \in \mathcal{S}_c, \\
z_c &\geq x_{t_i c} && \forall t_i \in \mathcal{T},\ \forall c \in \mathcal{S}_c, \\
\sum_{c \in \mathcal{S}_c} x_{t_i c} &= 1 && \forall t_i \in \mathcal{T}, \\
x_{t_i c} &\in \{0, 1\} && \forall t_i \in \mathcal{T},\ \forall c \in \mathcal{S}_c, \\
y_{ec} &\in \{0, 1\} && \forall e \in \mathcal{E},\ \forall c \in \mathcal{S}_c, \\
z_c &\in \{0, 1\} && \forall c \in \mathcal{S}_c,
\end{aligned}$$

where

$$x_{t_i c} = \begin{cases} 1 \text{ if the task } t_i \in \mathcal{T} \text{ is assigned to cluster } c \in \mathcal{S}_c, \\ 0 \text{ otherwise,} \end{cases}$$

$$y_{ec} = \begin{cases} 1 \text{ if } x_{t_i c} = 1 \text{ and } x_{t_j c} = 1 \text{ with } e = (t_i, t_j) \in \mathcal{E}, c \in \mathcal{S}_c, \\ 0 \text{ otherwise,} \end{cases}$$

$$z_c = \begin{cases} 1 \text{ if the cluster } c \in \mathcal{S}_c \text{ is used,} \\ 0 \text{ otherwise.} \end{cases}$$

The objective function to minimize is the number of clusters. The first constraint formalizes the memory constraint on clusters, with M_{max} the memory available in each cluster. The second and the third constraints impose the proper value for the variable y_{ec}. The fourth constraint forces a cluster with at least one task mapped on it to be counted as used. The last constraint forces only one cluster assignment per task.

6.6 NP-completeness of the mapping problem

The mapping problem is proved to be NP-complete. The proof uses a reduction to the problem of bin-packing. We explain the bin-packing problem first and the NP-completeness proof follows.

6.6.1 The bin-packing problem

Bin-packing is an NP-complete problem [Karp, 1972]. It consists in finding the optimal assignment of n elements of varying sizes into bins such that the number of bins be minimized. All the bins have the same size.

More formally we have a set O of n objects, each object $i \in O$ is characterized by its size p_i. Bins have capacity C_{max}. The bin-packing problem is expressed as the following optimization problem:

$$\min \sum_{j=1}^{n} y_j$$

such that

$$\begin{aligned}
\sum_{i=1}^{n} p_i x_{ij} \leq C_{max}.y_j \qquad & \forall j \in \{1,\ldots,n\}, \\
\sum_{j=1}^{n} x_{ij} = 1 \qquad & \forall i \in \{1,\ldots,n\}, \\
x_{ij} \in \{0,1\} \qquad & \forall i \in \{1\ldots n\} \forall j \in \{1\ldots n\}, \\
y_j \in \{0,1\} \qquad & \forall j \in \{1\ldots n\},
\end{aligned}$$

where

$$x_{ij} = \begin{cases} 1 \text{ if the object } i \in O \text{ is assigned to bin } j, \\ 0 \text{ otherwise} \end{cases}$$

and

$$y_j = \begin{cases} 1 \text{ if bin } j \text{ is used}, \\ 0 \text{ otherwise}. \end{cases}$$

6.6.2 NP-completeness of the mapping problem

Now we can prove that our mapping problem is NP-complete:

Theorem 20. *The mapping problem formalized in Section 6.5 is NP-complete.*

Proof. We use the reduction to the bin-packing problem. Let $\mathcal{U} = (\mathcal{T}, \mathcal{E})$ be the multivalued undirected graph. Let Γ_i be the set of edges adjacent to the task t_i. We assume that $\forall e = (t_i, t_j) \in \mathcal{E},\ w_e = w_{ei} + w_{ej}$. This assumption associates to a task a unique weight denoted $w_i = \sum_{e \in \Gamma_i} w_{ei}$.

Now we have p_i equivalent to w_i, C_{max} equivalent to M_{max}, x_{ij} equivalent to $x_{t_i s}$ and y_i equivalent to z_s and, since each task $t_i \in \mathcal{T}$ has a unique weight w_i the mapping problem is equivalent to the bin-packing problem. Thus, the set $\mathcal{T}$ is equivalent to the set O of the bin-packing problem and clusters are equivalent to bins.

Since the bin-packing problem is NP-complete, by reduction, the mapping problem considered in this thesis is also NP-complete. ■

6.7 State of the art for the mapping algorithms

The mapping problem using dataflow models is widely studied. In [Mirza et al., 2014] the authors identify 65 papers published between 1975 and 2014 on this problem. The two most common optimization criteria are data memory size and throughput. To the best of our knowledge, this thesis is the first which proposes a highly scalable approach to minimize the memory size under a throughput constraint. This short state of the art surveys several mapping approaches for dataflow models.

As far as we know, the mapping problem with the most scalable algorithm is proposed by [Berger et al., 2016]. The dataflow model used is equivalent to a HSDFG. The solution provided solves instances of more than 200,000 tasks, however, it does not take a throughput evaluation.

A mapping approach that also tackles scheduling is proposed in [Liu and Xiao, 2016]. The goal is, given a mapping, to find a schedule which respects non-overlapping constraints for tasks assigned to the same processor while ensuring maximum throughput. The model is an HSDFG. A re-timing phase to redistribute the initial marking is performed before determining the periodic schedule that maximizes the throughput. The solution is computed with a branch-and-bound technique which solves applications with up to 70 tasks in 30 minutes.

In [Ahn et al., 2008] the authors present a System-on-Chip Design AcceLerator (SocDAL) which uses the SDFG model to analyze and simulate the execution of an application on the designed system. The mapping is performed using an evolutionary algorithm and is computed in less than 0.1 second for graphs with 60 nodes or less.

A constraint programming methods is presented in [Bonfietti et al., 2013] to allocate and schedule an SDFG onto a multi-core architecture. The schedule is self-timed hence, ASAP. Graphs of up to 15 nodes are handled in 5 minutes.

6.8 Algorithms

We present several algorithms to solve the mapping problem. Most of them are inspired from the bin-packing problem. The algorithms were tested and implemented during the 5-month internship of Ahlam Mouaci at LIP6 as part of her M2.

We chose four algorithms used for bin-packing: First-Fit (FF) and Best-Fit (BF) and their decreasing versions. Two other algorithms have also been implemented: *Sort* and *Matching*.

To ease the resolution of the mapping problem and to avoid any unfeasible solution, when a task t_i is assigned to a cluster c, its footprint in c is always taken to be the worst case. This means that for any unassigned adjacent task t_j of t_i, the memory footprint is increased by w_{ei} even if t_j is not yet assigned to a cluster.

Note that the following descriptions of the algorithms consider an assignment of a task in a cluster to be valid simply if it satisfies the memory constraint.

6.8.1 First-Fit and First-Fit decreasing

The FF algorithm works as follows: for every task, the algorithm iterates on non-empty clusters until the task can be assigned. If the task cannot be assigned in a non-empty cluster, it is assigned to an empty cluster. The decreasing version works similarly except that tasks are sorted in decreasing order of their weight $\sum_{e \in \Gamma_i} w_{ei}$.

More formally:

input: list L of all tasks (sorted by decreasing $\sum_{e \in \Gamma_i} w_{ei}$ for the decreasing version).
$S_u = [\ ]$ (set of used clusters)
for $t_i \in L$ **do**
  $is\_assigned$ = **False**
  for $c \in S_u$ **do**
    if t_i *can be assigned in* c **then**
      assign t_i to c
      $is\_assigned$ = **True**
    end
  end
  if *not* $is\_assigned$ **then**
    assign t_i to a new cluster c.
    $S_u = S_u + c$
  end
end

6.8.2 Best-Fit and Best-Fit decreasing

The BF algorithm assigns the task in the cluster which is the fullest, if possible, or in the next fullest cluster otherwise. If no cluster can be found, the task is assigned to a new cluster. The decreasing version uses the same sorted list as FF decreasing.

Denoting the function which gives the memory utilization of a cluster $g(c)$ with $g(null) = 0$, the algorithm is described more formally:

input: list L of all tasks (sorted by their decreasing $\sum_{e\in\Gamma_i} w_{ei}$ for the decreasing version).
$S_u = [\ ]$ (set of used clusters)
for $t_i \in L$ **do**
  $best\_fit\_cluster = null$
  for $c \in \mathcal{S}_c$ **do**
    if t_i *can be assigned in* c **then**
      if $g(c) > g(best\_fit\_cluster)$ **then**
        $best\_fit\_cluster = c$
      end
    end
  end
  if $best\_fit\_cluster$ *is null* **then**
    $best\_fit\_cluster =$ an empty cluster.
  end
  assign t_i to $best\_fit\_cluster$.
  $S_u = S_u + best\_fit\_cluster$
end

6.8.3 Sorting heuristic

The sorting heuristic was proposed by Ahlam Mouaci during her internship. Let L_t be the list of all tasks sorted in decreasing order of the $\sum_{e\in\Gamma_i} w_{ei}$. The goal of the algorithm is to affect the first task of L_t and its neighbors, and the neighbors of its neighbors, and so on, on the same cluster c until an affectation in c is not possible. Each task added is removed from L_t, the algorithm stops when L_t is empty. This gives more formally:

input: list L_t of all tasks sorted according to the $\sum_{e \in \Gamma_i} w_{ei}$ in decreasing order.
while L_t *is not empty* **do**
  $t_i = L_t.pop$
  assign t_i to an empty cluster c
  continue = **True**
  while *continue* **do**
    $L_n = \Gamma_i$
    L_n is sorted according to the $\sum_{e \in \Gamma(t_i)} w_{ei}$ in decreasing order
    for $t_j \in L_n$ **do**
      if t_j *can be assigned to* c *and* t_j *is not already assigned* **then**
        assign t_j to c
        remove t_j from L_t
      end
      else
        continue = **False**
      end
    end
    $t_i = L_n[0]$
  end
end

The function *pop* returns the first task of a list and removes it from the list.

6.8.4 Matching heuristic

The matching heuristic was also proposed by Ahlam Mouaci. It uses a maximum weighted matching, which consists in finding a matching—a subset of edges in which no node occurs more than once—such that the total weight of the matching is maximal.

The matching algorithm builds a graph $\mathcal{H} = (\mathcal{N}, \mathcal{E})$ as follows. The nodes $n_i \in \mathcal{N}$ are the non-empty clusters of the mapping. Two nodes are linked by an edge when they can be merged without violating the memory constraint. The weight of an edge is the memory footprint of the two merged clusters.

The heuristic is initialized with the allocation of every task to a different cluster. Then, while a matching is not empty (meaning that a merge between two clusters is possible), the algorithm successively computes a matching and merges the matched clusters.

Maximum weighted matching is an algorithm with a time complexity of $|\mathcal{T}|^3$ [Galil, 1986], thus its execution time is not reasonable on large instances. To overcome the time complexity, an alternative consists in initializing the heuristic with a better solution than one task per cluster. Note that the maximum weight matching algorithm [Galil, 1986] is implemented in the library NetworkX already presented in

the Chapter 4. The algorithm is described more formally as:

input: A solution mapping m (by default each task is in a different cluster).
Build the graph $\mathcal{H}$ from m
$match = Maximum\_Weighted\_Matching(\mathcal{H})$
while *match is not empty* **do**
  for $e \in \mathcal{E}$ **do**
    $m =$ merge the corresponding clusters of e
  end
  build the graph $\mathcal{H}$ from m
  $match = Maximum\_Weighted\_Matching(\mathcal{H})$
end

6.9 Experiments

This section illustrates the relevance in terms of performance and scalability of the mapping algorithms described in the previous section. First we describe the experimental conditions. Next we examine the computation times of the preparatory steps required before executing the algorithms. The computation times of the algorithms are measured in the following part and the solutions are compared in terms of quality. Finally, the algorithms are tested on real applications.

6.9.1 Experimental conditions

The following algorithms are tested:

- the First-Fit heuristic and its decreasing order version
- the Best-Fit heuristic and its decreasing order version
- the sorting heuristic
- the matching heuristic
- the matching heuristic starting from the sort solution

The experiments were executed on a four-core *Intel Core I5* 660 at 3.33 Ghz with 6 GB of RAM under Linux and with Python 2.7. We use the same example dataflow graphs as in Chapter 4, of size 10, 100, 1,000 and 10,000. Each experiment is performed on 100 graphs, the average computation times are shown in the tables.

The memory size M_{max} of the clusters depends on the instances and is fixed equal to $\max_{t_i \in \mathcal{T}} \sum_{e \in \Gamma_i} w_e$.

Four preparatory steps precede the execution proper of a mapping algorithm. First the dataflow graph is read from a *.tur* file, then it is bounded by adding backward arcs, the initial marking is computed and finally the graph $\mathcal{U}$ is constructed. These steps are the same for all the mapping algorithms and their computation time is counted separately.

6.9.2 Computation time of the preparatory steps

The computation time to prepare the dataflow graph and construct the graph $\mathcal{U}$ in order to execute a mapping algorithm is shown in Table 6.1. The "read" column is the time to read a *.tur* file, "buffer" column is the buffer bounding time, "marking" is the initial marking computation time and "$\mathcal{U}$" is the computation time of the graph $\mathcal{U}$.

	Preparation Steps			
$\lvert\mathcal{T}\rvert$	read	buffer	marking	$\mathcal{U}$
10	0.00s	0.00s	0.00s	0.00s
100	0.02s	0.01s	0.06s	0.01s
1,000	0.14s	0.11s	2.08s	0.04s
10,000	1.42s	1.09s	13mn39s	0.45s

(a) SDFG

	Preparation Steps			
$\lvert\mathcal{T}\rvert$	read	buffer	marking	$\mathcal{U}$
10	0.01s	0.00s	0.01s	0.00s
100	0.02s	0.01s	0.11s	0.00s
1,000	0.19s	0.12s	9.13s	0.04s
10,000	2.77s	1.60s	56mn37s	0.58s

(b) CSDFG

Table 6.1 – Computation times of the preparatory steps for (a) the SDFG model and (b) the CSDFG model.

The preparatory steps are effected once for each instance, after that, the obtained graph $\mathcal{U}$ is saved into a text file. The computation time is reasonable up to graphs of 1,000 tasks. In comparison with the original unbounded dataflow graphs, the addition of backward arcs multiplies by two the number of constraints of **SCL1** and by four the number of constraints of **SCL2**. It gives a computation time of nearly an hour for CSDFG instances of 10,000 tasks with an average degree of 2.5.

6.9.3 Computation time of the mapping algorithms

This section compares the computation times of the algorithms presented in Section 6.8. The computation times are presented in two tables, Table 6.2 for the SDFG model, and Table 6.3 for the CSDFG model.

Tasks	FF	FFD	BF	BFD	Sort	Match	M+S
10	0.00s	0.00s	0.00s	0.00s	0.00s	0.02s	0.00s
100	0.01s	0.01s	0.04s	0.06s	0.03s	4.83s	0.17s
1,000	0.10s	0.12s	3.68s	4.87s	0.31s	23mn5s	20.21s
10,000	1.67s	1.94s	5mn28s	7m8s	3.22s	X	58mn19s

Table 6.2 – Computation time of the mapping algorithms for SDFGs having from 10 to 10,000 tasks. FFD and BFD are the decreasing versions of FF and BF.

Tasks	FF	FFD	BF	BFD	Sort	Match	M+S
10	0.00s	0.00s	0.00s	0.00s	0.00s	0.02s	0.00s
100	0.01s	0.01s	0.04s	0.05s	0.03s	9.89s	0.16s
1,000	0.10s	0.12s	2.25s	3.12s	0.30s	1h26mn23s	12.59s
10,000	1.07s	1.19s	2mn16s	3mn9s	1.87s	X	48mn17s

Table 6.3 – Computation time of the mapping algorithms for CSDFGs having from 10 to 10,000 tasks.

Except when using the matching algorithm, mappings are obtained faster for CSDFGs than for SDFGs. This is due to the larger gap between the weights of the tasks of the CSDFGs, leading to mappings with fewer clusters than for SDFGs. In a way consistent with the preparatory steps, instances with fewer than 10,000 tasks have a reasonable computation time.

6.9.4 Quality of the mappings

This section compares the solutions uncover by the mapping algorithms in terms of quality, as measured by achived number of clusters. As usual, the algorithms are tested on graphs from 10 to 10,000 tasks. The FF algorithm is used as a general reference. The result shown is the average number of clusters in percentage compared to the FF algorithm. Figures 6.6 and 6.7 present this measure of quality of the algorithms for the SDFG and the CSDFG model.

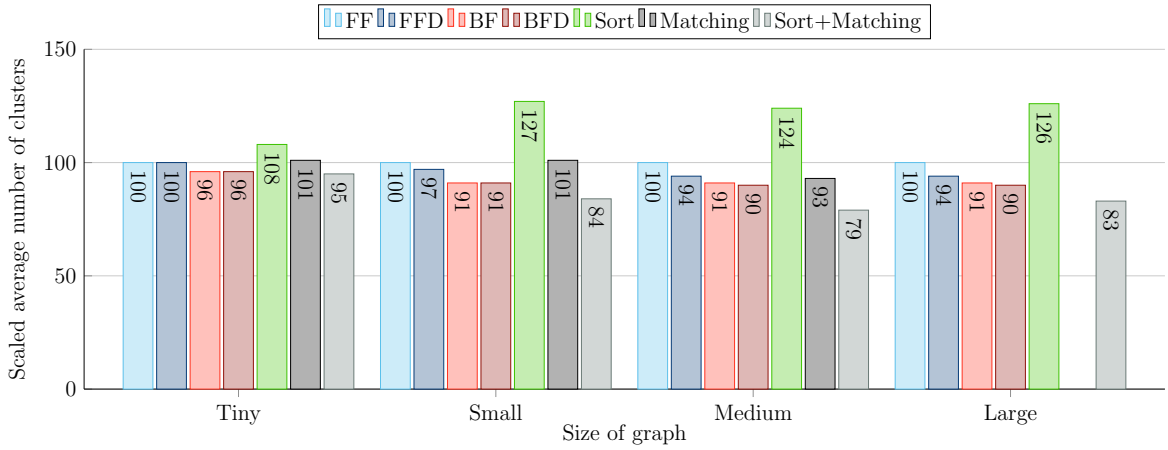


Figure 6.6 – Scaled number of clusters (set to 100 for the FF algorithm) for SDFGs of 10 to 10,000 tasks.

Without surprise the algorithms of bin-packing type give better results with their decreasing order version and BF is better than FF. The matching algorithm gives solutions close to the bin-packing algorithms, but, when combined with the sorting algorithm, gives mapping of better quality.

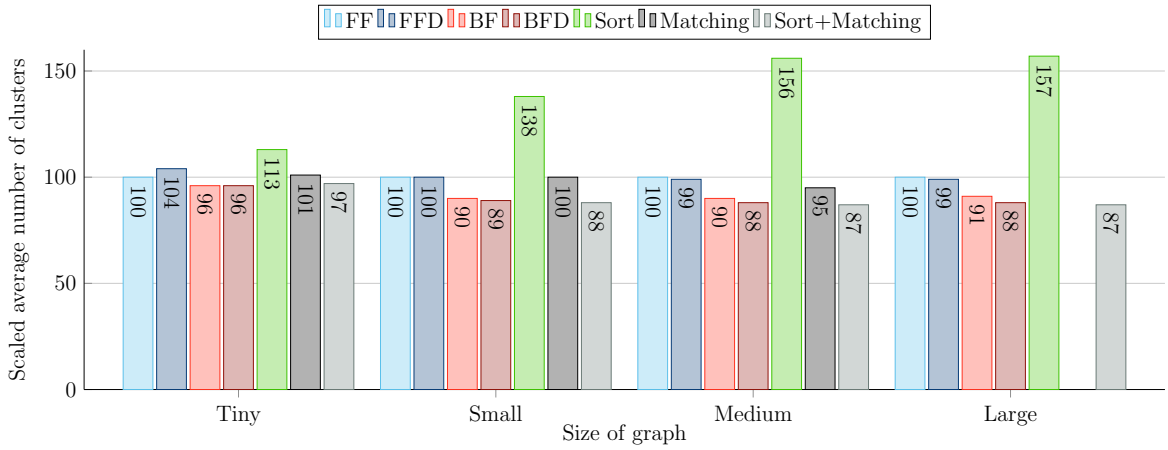


Figure 6.7 – Scaled number of clusters (set to 100 for the FF algorithm) for CSDFGs of 10 to 10,000 tasks.

Figure 6.7 shows that for CSDFGs the combined sort-matching algorithm also gives mappings with better average quality than the best bin-packing algorithm, however, this improvement is not as significant as for SDFGs.

6.9.5 Integer Linear Programming

In this section, we compare the results obtained when the mapping problem is solved using the integer linear program solver Gurobi and when it is solved using the three best algorithms: FF decreasing, BF decreasing and Sorting followed by Matching. The solver is tested on 100 instances of 10, 25, 50, 75 and 100 tasks. Both SDFGs and CSDFGs are tested. The Integer Linear Program (ILP) solver is limited to 10 minutes. Figure 6.8 presents the number of instances solved within the time limit.

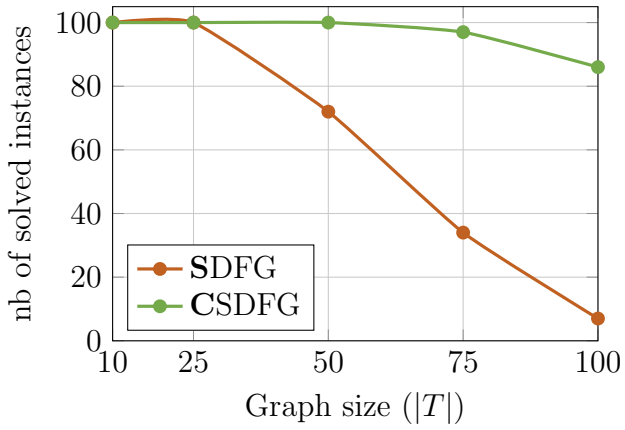
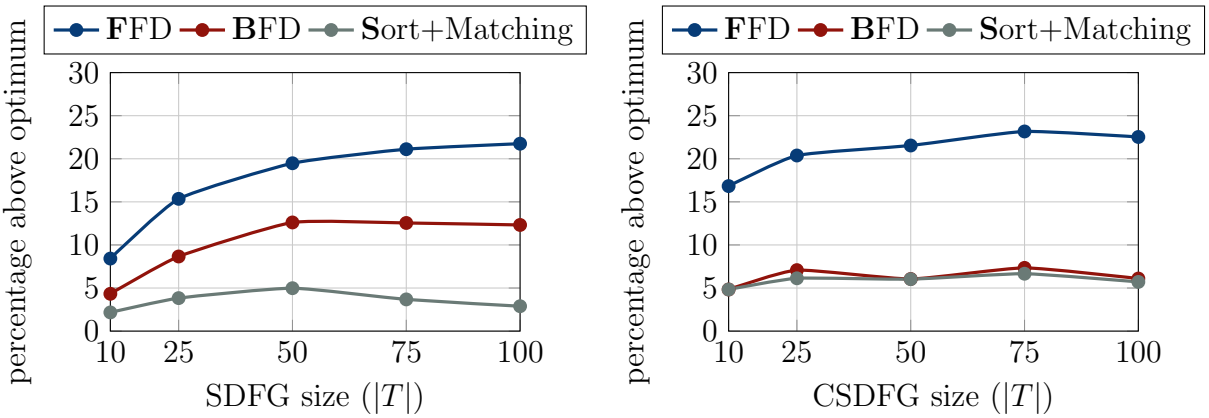


Figure 6.8 – The number of solved instances with the ILP solver for the SDFG and CSDFG model (the time limit is 10mn).

The mapping problem appears easier to solve for CSDFGs than for the SDFGs. For 100-task graphs, the solver solved only 5 SDFG instances while it solved more

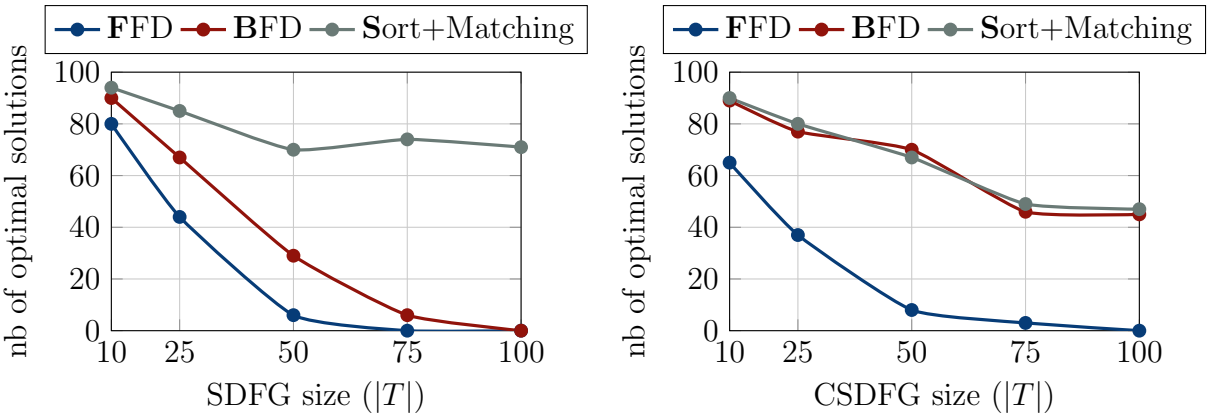
than 80 CSDFG instances within the time limit.

Figures 6.9 and 6.10 compare the quality of the solutions obtained with the FF decreasing, the BF decreasing and the Sorting followed by Matching algorithms to the optimal solution from Gurobi. Figure 6.9 gives the average percentage above the optimal solution. Figure 6.10 gives the percentage of solutions that are optimal.



(a) **(b)**

Figure 6.9 – Percentage above the optimum for (a) the SDFG and (b) the CSDFG models.



(a) **(b)**

Figure 6.10 – Percentage of mappings that are optimal for (a) the SDFG and (b) the CSDFG models.

As already seen, the Sort+Matching algorithm gives better result for SDFGs than for CSDFGs. BF decreasing gives same quality results than Sort+Matching on CSDFGs while being eight times faster.

6.9.6 Experiments on real applications

The throughput constrained version of the mapping problem is illustrated in this section on a selection of real applications. We first present the applications and then illustrate the mapping problem with throughput constraint with experiments. All the applications are described by CSDFGs.

BlackScholes is a financial tool which solves differential equations. This is our smallest instance.

Echo is an audio filter that cancels echoes from an audio source. It is used in cell phones.

H264 is a well known standard dedicated to video encoding/decoding. The application tested is the encoder. The application graph exhibits a high number of cycles and is also the largest.

JPEG2000 is a video encoder according to the JPEG2000 standard, more powerful and more flexible than the JPEG standard.

Pdetect is a an application dedicated to pedestrian detection.

Table 6.4 summarizes the characteristics of the applications described above. The column "Size" is the memory footprint of the application (without the communication buffers), column $||R||$ is the sum of the components of the repetition vector of the application, column $|\varphi|$ is the sum of the number of phases of all the tasks, the presence/absence of cycles in the application graph is given in the last column.

Name	$\|\mathcal{T}\|$	$\|\mathcal{A}\|$	Size	$\|\|R\|\|$	$\|\varphi\|$	Cycles
BlackScholes	41	40	16KB	923	261	No
Echo	38	82	28KB	35003	45	Yes
H264	666	3128	1368KB	762	1375	Yes
JPEG2000	240	703	3807KB	24676	639	No
Pdetect	58	76	3859KB	58	4045	No

Table 6.4 – Characteristics of the real applications mapped with constrained throughput.

All the applications were tested in their CSDFG version and in their functionally equivalent SDFG version (see Section 3.2.1). Table 6.5 presents the impact of the mapping on the application. The column "Model" indicates which model is used. Column "Time" is the computation time of the entire process, which includes the initial marking computation of the original application, the computation of the throughput before and after the mapping, the preparatory steps and the mapping itself. Note that verification of the throughput is done on the dataflow graph built as if the application were mapped: with the t_c tasks and their related arcs with the initial marking evaluated by our method. Column "$|\mathcal{S}|$" is the number of clusters used by the mapping, column "M_{max}" is the memory size per cluster, column "Mem" is the total memory footprint of the application, finally, column "Λ" informs if the original throughput constraint is met. The parameters of the communication task

com are set to $d = 5$ and $B = \frac{1}{2}$. The memory per cluster has been changed for the SDFG version of BlackScholes and JPEG2000 in order to provide feasible solutions (in terms of memory constraints).

	Model	Time	$\|\mathcal{S}\|$	M_{max}	Mem	Λ
BlackScholes	SDFG	0.3s	3	23KB	48KB	No
	CSDFG	0.6s	3	8KB	17KB	Yes
Echo	SDFG	0.3s	4	10KB	30KB	Yes
	CSDFG	0.3s	3	10KB	29KB	Yes
H264	SDFG	1mn3s	18	90KB	1328KB	Yes
	CSDFG	1mn1s	18	90KB	1372KB	Yes
JPEG2000	SDFG	3.8s	2	3532KB	6168KB	Yes
	CSDFG	5.6s	3	1695KB	3807KB	Yes
Pdetect	SDFG	3mn31s	4	928KB	3643KB	No
	CSDFG	5mn11s	4	928KB	3859KB	Yes

Table 6.5 – Results of the mapping using real applications.

As illustrated by Table 6.4 the mapping is more satisfactory for the CSDFG model than the SDFG model. For instance, the memory footprint of the SDFG is twice the size of that of the CSDFG for the application JPEG2000. The throughput constraint is not met for BlackScholes and Pdetect with the SDFG model while every mapping with the CSDFG model provides a feasible solution. Finally, the computation time appears closely related with the number of phases for the CSDFGs. Some solutions with the SDFG model does not respect the throughput constraint, this is due to the slow down cause by the communications between clusters. Due to the linear approximation of the (non-mapped) applications memory size given by their number of tokens, and because the mapping gives an exact evaluation for each buffer between clusters, the memory footprint is sometimes smaller when mapped.

6.10 Conclusion

In this chapter, first the multivalued undirected graph $\mathcal{U}$ was introduced. Then several algorithms, most of them inspired from the bin-packing, were described. Finally, the algorithms were run on random applications to map them on multi-cluster processor arrays under liveness constraint, and with real applications, to map them under throughput constraint.

The first part also exemplified the use of the graph $\mathcal{U}$ on the H263 application. It also discussed the preservation of functional behavior. The proof of functional behavior preservation for the mapping under liveness constraint has been presented. However, functionality is not guaranteed to be preserved when mapping under a throughput constraint.

The second part proposed four mapping algorithms inspired by the bin-packing

problem, First-Fit, First-Fit Decreasing, Best-Fit and Best-Fit Decreasing. Two other algorithm were also presented, a simple algorithm based on sorting and a more elaborate matching algorithm.

The last part presented multiple experiments to evaluate the computation time of the mapping using the proposed algorithms. The solutions were compared to optimal solutions. Finally, the mapping problem with throughput constraint was tested with a set of five real applications.

The mapping evaluations methods used to find a mapping which minimizes the number of clusters used proved to be efficient and fast. Applied with a CSDFG, the mapping, in most cases, required less memory. Between the SDFG and the CSDFG model, the computation time is quite similar, but often in favor of the SDFG model.

The flexibility of the methods also allows a finer evaluation of the mapping, for instance with a more complex model of the NoC or with a K-periodic schedule.

Chapter 7

Conclusion

This thesis proposes two contributions to the dataflow community. The first is the software package `Turbine`, a generator of live dataflow graphs (SDFG, CSDFG and Phased Computation Graph (PCG)) of up to 10,000 tasks in less than 30 seconds. This fast generation is performed by using classic methods of graph generation and an implementation of a sufficient condition of liveness as a linear program to compute a live marking. The objective of Turbine was to provide instances of several thousands of tasks in a reasonable time. During the thesis, the generator has evolved to become a powerful tool to study dataflow graphs. It is now available on github.

The second contribution is an evaluation method of a mapping on a distributed architecture. The method consists in evaluating the memory consumption of communications between the tasks of a dataflow graph. The method was first developed on the SDFG model and then extended to the CSDFG model. A declination of the problem with a throughput constraint was also studied on both dataflow models. The goal was to perform a memory footprint evaluation of instances of several thousand of tasks while keeping the computation time low. The evaluation method has been tested on a mapping problem which minimizes the number of clusters used. With the best algorithm implemented, memory footprint evaluation takes several seconds on instances of 1,000 tasks for both models.

Chapter 2 is an introduction to dataflow model. It presents three static dataflow models used in the thesis and their context. The SDFG is the simplest among the three, the CSDFG model is an extension of the SDFG where task execution is decomposed into phases, the PCG model is an extension of the CSDFG model, including initialization phases and thresholds for the consumption phases. The three models are compared in terms of expressiveness. Finally, the basic properties of the models are introduced using the SDFG: consistency, useful tokens, normalization, liveness and scheduling.

Chapter 3 extends the basic properties introduced using the SDFG model in the previous chapter to the CSDFG and the PCG model. One contribution of the thesis is to extend these properties to the PCG model.

Chapter 4 is dedicated to the dataflow generator `Turbine` implemented during the thesis. The chapter presents Turbine and its competitors. The dataflow graph

generators are compared experimentally in terms of computation time and number of tokens of the computed initial marking.

Chapter 5 introduces a new evaluation method for the memory footprint on distributed memory architectures. The method is declined in two versions, the first guarantees a live marking and the second adds a throughput constraint. The technique uses backward arc to bound the initial marking and a simple modification of the graph to simulate the functioning of a NoC. The modification of the graph implies local re-computation of the initial marking. The liveness is ensured using a sufficient condition of liveness while the throughput constraint is guaranteed using a one-periodic schedule. The chapter demonstrates the methods using the SDFG and CSDFG models.

Chapter 6 proposes a proof of concept for the evaluation method. The method is tested on a mapping problem which objective is to minimize the number of clusters used by the application. The evaluation method is used to create a multivalued undirected graph which summarizes the distribution of the buffer memory of a mapping. The multivalued undirected graph is illustrated with the H263 application. Finally, algorithms to solve the mapping problem are presented and evaluated experimentally using randomly generated and real applications for both declinations of the problem.

7.1 Perspectives

This section presents perspectives and potential future work in the continuation of this thesis.

Extension of the K-periodic schedule for the PCG model

The PCG model is currently used with the ΣC compiler of Kalray to model a dataflow applications. The one-periodic and the ASAP schedule are easily extended to the PCG graph. As shown in [Bodin et al., 2016] the K-periodic schedule provides a maximal throughput under certain conditions and is computed much faster than the ASAP schedule. Thus, an extension to the PCG of K-periodic scheduling appears worth exploring.

Characterization of schedule with overlapping

Reentrant execution allows to express parallelism without duplicating tasks in dataflow graphs. The characterization of the ASAP, one-periodic and K-periodic schedule in Section 2.5 and 3.3 for the SDFG and CSDFG, expressed the non-reentrant behavior of tasks by adding a sequential execution constraint for each task when determining the schedule. The suppression of this constraint makes the schedule reentrant. Intermediate levels of parallelism may be obtained by adding a system to control the number of simultaneous executions of each task.

Dataflow mapping under throughput constraint

As presented in this thesis, the mapping under throughput constraint may introduce tokens such that the system's behavior does not respect the functionality of the application. Several approaches can be used in conjunction to satisfy the throughput constraint while preserving functionality, a sine qua non property for a mapping. The simplest is to place tasks whose communication is critical for the throughput in the same cluster.

Mapping on an architecture with RAM or with heterogeneous cluster

The mapping problem was studied using a specific architecture. However the multivalued undirected graph could be an efficient tool for other architectures. For instance an architecture with a shared fast memory in complement of the small amount of memory in the clusters. Heterogeneous cluster architectures could also be considered.

The mapping problem with routing latency depending on the NoC topology

The solution proposed to take into account the NoC latency for the mapping problem has been presented in a simple fashion. The markings of the normalized graph used for memory and communication delay evaluation must be divided by the normalization factor N_a to get corresponding numbers of data items, which, when multiplied by their size in bits allows to better account for contention. Many different topologies of NoC exist with latency depending on the hop number and the use of through-silicon via. If those different topologies were reunited into a single mathematical formalism our approach to computing throughput could be extended to these NoC topologies.

Publications

[Bodin et al., 2014] Bodin, B., Lesparre, Y., Delosme, J.-M., and Munier-Kordon, A. (2014). Fast and efficient dataflow graph generation. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 40–49. ACM.

[Lesparre et al., 2015] Lesparre, Y., Munier-Kordon, A., and Delosme, J.-M. (2015). Compile-time mapping of dataflow applications with buffer minimization. In *Investigating Dataflow in Embedded computing Architectures (IDEA), 2015 HIPEAC Conference.*

[Lesparre et al., 2016] Lesparre, Y., Munier-Kordon, A., and Delosme, J.-M. (2016). Evaluation of synchronous dataflow graph mappings onto distributed memory architectures. 2016 Euromicro Conference In *Digital System Design (DSD)*, pages 146–153. IEEE.

Acronyms

ASAP : As Soon As Possible
BF : Best-Fit
CG : Computation Graph
CSDFG : Cyclo-Static Dataflow Graph
DSP : Digital Signal Processing
FF : First-Fit
FIFO : First in First out
HSDFG : Homogeneous Synchronous Dataflow Graph
ILP : Integer Linear Program
KPN : Kahn Process Network
MPPA : Massively Parallel Processor Array
MoC : Model of Computation
NoC : Network on Chip
PCG : Phased Computation Graph
SDF3 : SDF For Free
SDFG : Synchronous Dataflow Graph
WCET : Worst Case Execution Time

Notation

	First appearance
General notation	
$\lceil x \rceil_z$: smallest integer larger x multiple of z	24
$\lceil x \rceil$: smallest integer larger than x	24
$\lfloor x \rfloor_z$: largest integer smaller than x multiple of z	24
$\lfloor x \rfloor$: largest integer smaller than x	24
gcd : greatest common divisor	24
lcm : least common multiple	24
$\lvert\mathcal{E}\rvert$: size of the set $\mathcal{E}$	24

	First appearance
Dataflow	
$\mathcal{T}$: set of tasks (or actors)	27
$\mathcal{A}$: set of arcs (or channels)	27
$\mathcal{M}$: set of initial markings	28
$M_0(a)$: initial marking on arc $a = (t_i, t_j)$	28
R_i : repetition factor of task t_i	37
R : repetition vector $R = [R_1, \cdots, R_{\lvert\mathcal{T}\rvert}]$	37
N_a : normalization factor of arc $a = (t_i, t_j)$, $N_a = \frac{lcm(R)}{R_{t_i}.p_a}$ (or $N_a = \frac{lcm(R)}{R_{t_j}.c_a}$)	38
N : normalization vector $N = [N_{a_1}, \cdots, N_{a_{\lvert\mathcal{A}\rvert}}]$	38
$\Gamma^+(t)$: set of input arcs of task t	80
$\Gamma^-(t)$: set of output arcs of task t	80
s : schedule	43
T : period	47
$\lambda(t)$: throughput of task t, $\lim_{n\to\infty} \frac{n}{s\langle t,n\rangle}$	44
Λ : global throughput of a graph $\Lambda = \frac{1}{T}$	44
w_i : period of the task t_i	47
K_i : periodicity factor of task t_i for K-periodic schedule	48
K : periodicity vector, $K = [K_1, \cdots, K_{\lvert\mathcal{T}\rvert}]$	48

	First appearance
SDF	
$\mathcal{G}_{sdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$: Synchronous Dataflow Graph	28
ℓ_i : execution time of task t_i	28, 29
c_a : consumption rate of task t' from arc $a = (t, t')$	28

p_a :	production rate of task t on arc $a = (t, t')$	28
$\mathcal{P}$:	set of production weights	28
$\mathcal{C}$:	set of consumption weights	28
$\mathcal{L}$:	set of execution times	28
gcd_a :	$gcd(c_a, p_a)$	38
Z_i :	normalized weight of task t_i, $Z_i = p_a.N_a$ where $a = (t_i, \cdot)$	38
SCL :	sufficient condition of liveness	40
$s\langle t, n\rangle$:	start time of execution $\langle t, n\rangle$ (n^{th} execution of task t)	43
gcd_{Ka} :	$gcd(K_i.Z_i, K_j.Z_j)$ for arc $a = (t_i, t_j)$	49
$\pi_a^{max}(m_i, m_j)$:	$\lfloor M_0(a) - Z_i + gcd_a + Z_i.m_i - Z_j.m_j \rfloor^{gcd_{Ka}}$	49
$\pi_a^{min}(m_i, m_j)$:	$\lceil M_0(a) - \max\{Z_i - Z_j, 0\} + Z_i.m_i - Z_j.m_j \rceil^{gcd_{Ka}}$	49

First appearance

CSDF

$\mathcal{G}_{csdf} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \mathcal{M}, \mathcal{L})$:	Cyclo-Static Dataflow Graph	30
φ_i :	number of phases of task t_i	30
$p_a(k)$:	production rate of phase k of task t on arc $a = (t, \cdot)$	30
$c_a(k)$:	consumption rate of phase k of task t from arc $a = (\cdot, t)$	30
c_a :	sum of the components of consumption vector of arc a	30
p_a :	sum of the components of production vector of arc a	30
$\ell_i(k)$:	execution time of phase k of task t_i	30
$\mathcal{P}$:	set of production vectors	30
$\mathcal{C}$:	set of consumption vectors	30
$\mathcal{L}$:	set of vectors of execution times	30
$\langle t_i(k), n\rangle$:	n^{th} execution of the k^{th} phase of task t_i	55
$\langle t_i(k), n\rangle^{-1}$:	phase execution preceding $\langle t_i(k), n\rangle$	55
$P_a(k, n)$:	total number of data items produced by t_i on $a = (t_i, \cdot)$ from its first phase to the end of $\langle t_i(k), n\rangle$, $P_a(k, n) = \sum_{k'=1}^{k} p_a(k') + p_a.(n-1)$	55
$P_a^{-1}(k, n)$:	total number of data items produced by t_i on $a = (t_i, \cdot)$ from its first phase to the end of $\langle t_i(k), n\rangle^{-1}$	55
$C_a(k, n)$:	total number of data items consumed by t_i from the buffer $a = (\cdot, t_i)$ until the end of $\langle t_i(k), n\rangle$, $C_a(k, n) = \sum_{k'=1}^{k} c_a(k') + c_a.(n-1)$	56
$C_a^{-1}(k, n)$:	total number of data items consumed by t_i from the buffer $a = (\cdot, t_i)$ until the end of $\langle t_i(k), n\rangle^{-1}$	56
SCL1 :	sufficient condition of liveness 1	56
SCL2 :	sufficient condition of liveness 2	56
$step_a$:	$gcd(p_a(i), \cdots, p_a(\varphi_i), c_a(j), \cdots, c_a(\varphi_j))$ where $a = (t_i, t_j)$	54
$W_a^{csdf}(k_i, k_j)$:	$C_a(k_j, 1) - P_a^{-1}(k_i, 1) - step_a$ with $a = (t_i, t_j)$	58
W_{a_i,a_j}^{csdf} :	$\max_{k \in \{1,\cdots\varphi_e\}} \left[C_{a_i}(k, 1) - P_{a_j}^{-1}(k, 1) - step_{a_j}\right]$ with $a_i = (\cdot, t_e)$ and $a_j = (t_e, \cdot)$	58
$\alpha_a^{max}(k_i, k_j)$:	$\lceil M_0(a) + P_a(k_i, 1) - C_a(k_j, 1) - \max\{0, p_a(k_i) - c_a(k_j)\}\rceil^{gcd_a}$ with $a = (t_i, t_j)$	62
$\alpha_a^{min}(k_i, k_j)$:	$\lfloor M_0(a) + P_a^{-1}(k_i, 1) - C_a(k_j, 1) + 1\rfloor^{gcd_a}$ with $a = (t_i, t_j)$	62

PCG

		First appearance
$\mathcal{G} = (\mathcal{T}, \mathcal{A}, \mathcal{P}, \mathcal{C}, \Theta, \mathcal{M}, \mathcal{L})$:	Phased Computation Graph	32
σ_i :	number of initial phases of the task t_i	31
δ_t :	set of initial phases and cyclic phases of the task t_i, $\delta_i = \{1-\sigma_i, \cdots, 0\} \cup \{1, \cdots, \varphi_i\}$	65
$\theta_a(k)$:	threshold of consumption phase k on arc a	33
c_a :	sum of the rates of the cyclical consumption phases from arc a	65
p_a :	sum of the rates of the cyclical production phases on arc a	65
$\langle t_i(k), n\rangle^{-2}$:	phase execution two executions before $\langle t_i(k), n\rangle$	66
$P_a(k, n)$:	total number of data items produced by t_i on $a = (t_i, \cdot)$ from its first phase to the end of $\langle t(k), n\rangle$, $P_a(k, n) = \sum_{k'=1-\sigma_t}^{k} p_a(k') + p_a.(n-1)$	67
$P_a^{-1}(k, n)$:	total number of data items produced by t_i on $a = (t_i, \cdot)$ from its first phase to the end of $\langle t(k), n\rangle^{-1}$	67
$C_a(k, n)$:	total number of data items consumed by t_i from the buffer a until the end of $\langle t(k), n\rangle$, $C_a(k, n) = \sum_{k'=1-\sigma_t}^{k} c_a(k') + c_a.(n-1)$	68
$C_a^{-1}(k, n)$:	total number of data items consumed by t_i from the buffer a until the end of $\langle t(k), n\rangle^{-1}$	68
$W_{a_i}^{pcg}(k_i, k_{i+1})$:	$C_{a_i}^{-1}(k_{i+1}, 1) + \theta_{a_i}(k_{i+1}) - P_{a_i}^{-1}(k_i, 1) - step_{a_i}$	72
W_{a_{i-1},a_i}^{pcg} :	$\max_{k \in \delta_i}[C_{a_{i-1}}^{-1}(k, 1) + \theta_{a_{i-1}}(k) - P_{a_i}^{-1}(k, 1)] - step_{a_i}$	74

Mapping problem

		First appearance
$\overline{a}$:	backward arc associated to arc a	93
σ_a :	sum of the initial markings on arc a and backward arc $\overline{a}$	93
$\mathcal{S}_c$:	set of clusters	93
M_{max} :	size of cluster memory	93
t_c :	communication task modeling communication between tasks through the NoC	93
$H(a)$:	(SDF) height of an arc, $H(a) = M_0(a) + gcd_a - Z_j$ with $a = (t_i, t_j)$	95
$\widetilde{T}$:	(SDF) reduced system period, $\widetilde{T} = \frac{T}{Z_i.R_i}$ with $a = (t_i, \cdot)$	99
$H_{k_i,k_j}(a)$:	(CSDF) height of an arc, $H_{k_i,k_j}(a) = M_0(a) + P_a^{-1}(k_i, 1) - C_a(k_j, 1) + step_a$	102
$\widetilde{T}$:	(CSDF) reduced system period, $\widetilde{T} = \frac{T}{p_a.R_i}$ with $a = (t_i, \cdot)$	105
$\mathcal{U} = (\mathcal{T}, \mathcal{E})$:	undirected multivalued graph, with $\mathcal{T}$ the set of tasks and $\mathcal{E}$ the set of edges	113
w_e :	memory consumption of buffer $e = (t_i, t_j)$ if t_i and t_j are in the same cluster	113
w_{ei} :	memory consumption of task t_i for buffer $e = (t_i, t_j)$ if t_i and t_j are in different clusters	113
$g(m, c)$:	total memory needed for communication buffers in cluster c for a mapping m, $g(m, c) = \sum_{e \in \mathcal{E}_c} w_e + \sum_{e \in \mathcal{E}'_c} w_{e,t_i}$	113

List of Figures

List of Tables

Bibliography

[Ahn et al., 2008] Ahn, Y., Han, K., Lee, G., Song, H., Yoo, J., Choi, K., and Feng, X. (2008). SoCDAL: System-on-chip design accelerator. *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, 13(1):17.

[Aubry et al., 2013] Aubry, P., Beaucamps, P.-E., Blanc, F., Bodin, B., Carpov, S., Cudennec, L., David, V., Doré, P., Dubrulle, P., De Dinechin, B. D., et al. (2013). Extended cyclostatic dataflow program compilation and execution for an integrated manycore processor. *Procedia Computer Science*, 18:1624–1633.

[Baccelli et al., 1992] Baccelli, F., Cohen, G., Olsder, G. J., and Quadrat, J.-P. (1992). *Synchronization and linearity: an algebra for discrete event systems*. John Wiley & Sons Ltd.

[Bekooij et al., 2006] Bekooij, M., Smit, G., Jansen, P., and Wiggers, M. (2006). Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Proceedings of the 4th International Conference on Hardware/Software Codesign and System Synthesis, 2006. CODES+ ISSS'06.*, pages 10–15. IEEE.

[Benabid-Najjar et al., 2012] Benabid-Najjar, A., Hanen, C., Marchetti, O., and Munier-Kordon, A. (2012). Periodic schedules for bounded timed weighted event graphs. *IEEE Transactions on Automatic Control*, 57(5):1222–1232.

[Benazouz et al., 2010] Benazouz, M., Marchetti, O., Munier-Kordon, A., and Michel, T. (2010). A new method for minimizing buffer sizes for cyclo-static dataflow graphs. In *2010 8th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia)*, pages 11–20. IEEE.

[Benazouz et al., 2013] Benazouz, M., Munier-Kordon, A., Hujsa, T., and Bodin, B. (2013). Liveness Evaluation of a Cyclo-Static DataFlow Graph. In *Design Automation Conference (DAC'13)*, pages 3–7.

[Berger et al., 2016] Berger, K.-E., Le Cun, B., Sirdey, R., et al. (2016). A semi-greedy heuristic for the mapping of large task graphs. In *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International Conference*, pages 817–824. IEEE.

[Bhattacharyya et al., 2000] Bhattacharyya, S., Leupers, R., and Marwedel, P. (2000). Software synthesis and code generation for signal processing systems. *IEEE Transactions on Circuits and Systems II: Analog and Digital Signal Processing*, 47(9):849–875.

[Bilsen et al., 1995] Bilsen, G., Engels, M., Lauwereins, R., and Peperstraete, J. A. (1995). Cyclo-static data flow. In *1995 International Conference on Acoustics, Speech, and Signal Processing, 1995. ICASSP-95.*, volume 5, pages 3255–3258. IEEE.

[Bodin et al., 2014] Bodin, B., Lesparre, Y., Delosme, J.-M., and Munier-Kordon, A. (2014). Fast and efficient dataflow graph generation. In *Proceedings of the 17th International Workshop on Software and Compilers for Embedded Systems*, pages 40–49. ACM.

[Bodin et al., 2012] Bodin, B., Munier-Kordon, A., and De Dinechin, B. D. (2012). K-periodic schedules for evaluating the maximum throughput of a synchronous dataflow graph. In *2012 International Conference on Embedded Computer Systems (SAMOS)*, pages 152–159. IEEE.

[Bodin et al., 2016] Bodin, B., Munier-Kordon, A., and de Dinechin, B. D. (2016). Optimal and fast throughput evaluation of csdf. In *Proceedings of the 53rd Annual Design Automation Conference*, DAC '16, pages 160:1–160:6, New York, NY, USA. ACM.

[Bodin et al., 2013] Bodin, B., Munier-Kordon, A., and Dupont de Dinechin, B. (2013). Periodic Schedules for Cyclo-Static Dataflow. In *11th IEEE Workshop on Embedded Systems for Real-Time Multimedia (ESTIMedia'13)*, pages 105–114.

[Bonfietti et al., 2010] Bonfietti, A., Benini, L., Lombardi, M., and Milano, M. (2010). An efficient and complete approach for throughput-maximal SDF allocation and scheduling on multi-core platforms. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2010*, pages 897–902. IEEE.

[Bonfietti et al., 2013] Bonfietti, A., Lombardi, M., Milano, M., and Benini, L. (2013). Maximum-throughput mapping of SDFGs on multi-core SoC platforms. *Journal of Parallel and Distributed Computing*, 73(10):1337–1350.

[Buck and Lee, 1993] Buck, J. T. and Lee, E. A. (1993). Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *1993 IEEE International Conference on Acoustics, Speech, and Signal Processing, 1993. ICASSP-93.*, volume 1, pages 429–432. IEEE.

[Chakilam and O'Neil, 2009] Chakilam, S. R. A. K. C. and O'Neil, T. W. (2009). Static scheduling for cyclo-static data flow graphs.

[Chrétienne, 1985] Chrétienne, P. (1985). Transient and limiting behavior of timed event graphs. *RAIRO Techniques et Sciences Informatiques*, 4:127–192.

[Cohen et al., 1985] Cohen, G., Dubois, D., Quadrat, J., and Viot, M. (1985). A linear-system-theoretic view of discrete-event processes and its use for performance evaluation in manufacturing. *IEEE Transactions on Automatic Control*, 30(3):210–220.

[Commoner et al., 1971] Commoner, F., Holt, A. W., Even, S., and Pnueli, A. (1971). Marked directed graphs. *Journal of Computer and System Sciences*, 5(5):511–523.

[Davis and Keller, 1982] Davis, A. L. and Keller, R. M. (1982). Data flow program graphs. *Computer 15.2*, pages 26–41.

[de Groote et al., 2012] de Groote, R., Kuper, J., Broersma, H., and Smit, G. J. (2012). Max-plus algebraic throughput analysis of synchronous dataflow graphs. In *38th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pages 29–38. IEEE.

[Desnos et al., 2016] Desnos, K., Pelcat, M., Nezan, J.-F., and Aridhi, S. (2016). On memory reuse between inputs and outputs of dataflow actors. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(2):30.

[Galil, 1986] Galil, Z. (1986). Efficient algorithms for finding maximum matching in graphs. *ACM Computing Surveys (CSUR)*, 18(1):23–38.

[Geilen and Stuijk, 2010] Geilen, M. and Stuijk, S. (2010). Worst-case performance analysis of synchronous dataflow scenarios. In *Proceedings of the eighth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*, pages 125–134. ACM.

[Ghamarian et al., 2006a] Ghamarian, A. H., Geilen, M., Basten, T., Theelen, B. D., Mousavi, M. R., and Stuijk, S. (2006a). Liveness and boundedness of synchronous data flow graphs. In *Formal Methods in Computer Aided Design, 2006. FMCAD'06*, pages 68–75. IEEE.

[Ghamarian et al., 2006b] Ghamarian, A. H., Geilen, M., Stuijk, S., Basten, T., Moonen, A., Bekooij, M. J., Theelen, B. D., and Mousavi, M. (2006b). Throughput analysis of synchronous data flow graphs. In *Sixth International Conference on Application of Concurrency to System Design, 2006. ACSD 2006.*, pages 25–36. IEEE.

[Ito and Parhi, 1994] Ito, K. and Parhi, K. K. (1994). Determining the iteration bounds of single-rate and multi-rate data-flow graphs. In *1994 IEEE Asia-Pacific Conference on Circuits and Systems, 1994. APCCAS'94.*, pages 163–168. IEEE.

[Kahn, 1974] Kahn, G. (1974). The semantics of a simple language for parallel programming. *Information Processing*, 74:471–475.

[Karp, 1972] Karp, R. M. (1972). Reducibility among combinatorial problems. In *Complexity of computer computations*, pages 85–103. Springer.

[Karp and Miller, 1966] Karp, R. M. and Miller, R. E. (1966). Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411.

[Khasawneh, 2007] Khasawneh, S. F. (2007). *Static Scheduling for synchronous data flow graphs*. PhD thesis, University of Akron.

[Lee and Messerschmitt, 1987] Lee, E. A. and Messerschmitt, D. G. (1987). Synchronous dataflow. *Proceedings of the IEEE*, 75(9):1235–1245.

[Lee and Parks, 1995] Lee, E. A. and Parks, T. M. (1995). Dataflow process networks. *Proceedings of the IEEE*, 83(5):773–801.

[Lin et al., 2012] Lin, J., Gerstlauer, A., and Evans, B. L. (2012). Communication-aware heterogeneous multiprocessor mapping for real-time streaming systems. *Journal of Signal Processing Systems*, 69(3):279–291.

[Liu and Xiao, 2016] Liu, W. and Xiao, C. (2016). An efficient technique of application mapping and scheduling on real-time multiprocessor systems for throughput optimization. *ACM Transactions on Embedded Computing Systems (TECS)*, 15(4):65.

[Marchetti and Munier-Kordon, 2009a] Marchetti, O. and Munier-Kordon, A. (2009a). A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2):532–540.

[Marchetti and Munier-Kordon, 2009b] Marchetti, O. and Munier-Kordon, A. (2009b). Cyclic scheduling for the synthesis of embedded systems. *Y. Vivien and R. Frederic, editors, Introduction to scheduling, chapter 6. Chapman and Hall/CRC Press, 2009*, pages 135–164.

[Marwedel et al., 2011] Marwedel, P., Teich, J., Kouveli, G., Bacivarov, I., Thiele, L., Ha, S., Lee, C., Xu, Q., and Huang, L. (2011). Mapping of applications to MP-SoCs. In *Proceedings of the 7th International Conference on Hardware/Software Codesign and System Synthesis*, pages 109–118. ACM.

[Mirza et al., 2014] Mirza, U. M., Arslan, M. A., Cedersjo, G., Sulaman, S. M., and Janneck, J. W. (2014). Mapping and scheduling of dataflow graphs—a systematic map. In *Proceedings of the 48th Asilomar Conference on Signals, Systems and Computers*, pages 1843–1847. IEEE.

[Moreira et al., 2010] Moreira, O., Basten, T., Geilen, M., and Stuijk, S. (2010). Buffer sizing for rate-optimal single-rate data-flow scheduling revisited. *IEEE Transactions on Computers*, 59(2):188–201.

[Nikitin and Cortadella, 2009] Nikitin, N. and Cortadella, J. (2009). A performance analytical model for network-on-chip with constant service time routers. In *Proceedings of the 2009 International Conference on Computer-Aided Design*, pages 571–578. ACM.

[Oh and Ha, 2002] Oh, H. and Ha, S. (2002). Fractional rate dataflow model and efficient code synthesis for multimedia applications. *ACM SIGPLAN Notices*, 37(7):12–17.

[Parhi, 1995] Parhi, K. K. (1995). High-level algorithm and architecture transformations for DSP synthesis. *Journal of VLSI Signal Processing Systems for Signal, Image and Video Technology*, 9(1-2):121–143.

[Pelcat et al., 2014] Pelcat, M., Desnos, K., Heulot, J., Guy, C., Nezan, J.-F., and Aridhi, S. (2014). PREESM: a dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *6th European Embedded Design in Education and Research Conference (EDERC)*, pages 36–40. IEEE.

[Saha et al., 2006] Saha, S., Puthenpurayil, S., and Bhattacharyya, S. S. (2006). Dataflow transformations in high-level DSP system design. In *International Symposium on System-on-Chip, 2006.*, pages 1–6. IEEE.

[Singh et al., 2013] Singh, A. K., Shafique, M., Kumar, A., and Henkel, J. (2013). Mapping on multi/many-core systems: Survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 1:1–1:10, New York, NY, USA. ACM.

[Sriram and Bhattacharyya, 2009] Sriram, S. and Bhattacharyya, S. S. (2009). *Embedded multiprocessors: Scheduling and synchronization*. CRC, second edition.

[Stuijk et al., 2007] Stuijk, S., Basten, T., Geilen, M., and Corporaal, H. (2007). Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs. In *Proceedings of the 44th annual Design Automation Conference*, pages 777–782. ACM.

[Stuijk et al., 2006] Stuijk, S., Geilen, M., and Basten, T. (2006). SDF3: SDF For Free. In *Proceedings of the 6th International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278.

[Stuijk et al., 2008] Stuijk, S., Geilen, M., and Basten, T. (2008). Throughput-buffering trade-off exploration for cyclo-static and synchronous dataflow graphs. *IEEE Transactions on Computers*, 57(10):1331–1345.

[Thies et al., 2002] Thies, W., Lin, J., and Amarasinghe, S. (2002). Phased computation graphs in the polyhedral model. *Technical Report LCS-TM-630, M.I.T. Laboratory for Computer Science*.

[Zhou et al., 2013] Zhou, Z., Desnos, K., Pelcat, M., Nezan, J.-F., Plishker, W., and Bhattacharyya, S. S. (2013). Scheduling of parallelized synchronous dataflow actors. In *Proceedings of the 2013 International Symposium on System on Chip (SoC)*, pages 1–10. IEEE.

[Zhu et al., 2010] Zhu, J., Sander, I., and Jantsch, A. (2010). Constrained global scheduling of streaming applications on MPSoCs. In *Proceedings of the 2010 Asia and South Pacific Design Automation Conference*, pages 223–228. IEEE Press.