Analyse d'Applications Flot de Données pour la Compilation Multiprocesseur

Bruno Bodin

THÈSE DE DOCTORAT DE
l'UNIVERSITÉ PIERRE ET MARIE CURIE

Spécialité

Informatique

École doctorale Informatique, Télécommunications et Électronique (Paris)

Présentée par

Bruno BODIN

Pour obtenir le grade de

DOCTEUR de l'UNIVERSITÉ PIERRE ET MARIE CURIE

Sujet de la thèse :

Analyse d'Applications Flot de Données pour la Compilation Multiprocesseur

soutenue le 20 décembre 2013

devant le jury composé de :

M. Dritan NACE	Rapporteur
M. Jean-François NEZAN	Rapporteur
M. Frédéric BONIOL	Examinateur
M. Albert COHEN	Examinateur
Mme. Safia KEDAD-SIDHOUM	Examinateur
M. Renaud SIRDEY	Examinateur
Mme. Alix MUNIER-KORDON	Directeur de thèse
M. Benoît DUPONT DE DINECHIN	Encadrant de thèse

… pour vos loyaux services
On vous laisse conserver
Un unique échantillon
Comotive ou zoizillon
Tout reprendre à son début
Tous ces lourds secrets perdus
Toute science abattue
Si je laisse la machine
Mais ses plumes sont si fines
Et son coeur battrait si vite
Que je garderais l'oiseau.

Boris Vian

Remerciements

Je tiens tout d'abord à remercier Mme la professeure Alix Munier-Kordon. Merci pour son accueil et sa gentillesse, merci pour ses nombreux encouragements et conseils, et merci pour son indéfectible implication dans ce projet. Sans elle, je n'aurais pas été en mesure de présenter ces travaux avec autant de fierté.

Mes remerciements vont également à M. Benoît Dupont de Dinechin. Malgré la distance, il a su m'encadrer avec sérieux et compétence. Chacune de nos rencontres, et chacun de nos entretiens furent indéniablement bénéfiques pour l'avancée de ce projet. C'est avec sincérité que je tiens à lui exprimer toute ma gratitude.

Je remercie aussi l'ensemble des membres de mon jury de thèse. Merci à M. le professeur Dritan Nace et à M. le professeur Jean-François Nezan d'avoir accepté d'être les rapporteurs de cette thèse, de même que pour leur participation au Jury. Merci également à M. le professeur Frédéric Boniol, M. Albert Cohen, Mme Safia Kedad-Sidhoum et M. Renaud Sirdey d'avoir participé à mon jury de thèse en tant qu'examinateurs. J'ai été touché par l'intérêt que chacun a porté pour mon travail de thèse ; je les en remercie profondément.

Enfin, ces remerciements ne seraient pas complets sans mentionner toutes les personnes qui m'ont accompagnée et soutenue tout au long de ces trois années :

Merci à Andrea, Frédéric, et Margot : partager un bureau avec moi n'était pas tous les jours facile, mais ils ont toujours su m'apporter leur aide et leur soutien. Merci à Benjamin, Benoît, Frédéric, Jérome, Nicolas, Riadh, Samuel, Stéphane et Xavier : chacun a pris de son temps pour échanger avec moi, merci pour leur aide incontestable.

Je tiens véritablement à mentionner le plaisir que j'ai eu à travailler au sein de la société Kalray. J'ai eu la chance d'y côtoyer des personnes d'une immense gentillesse et qui ont su instaurer une ambiance de travail chaleureuse.

Merci aussi à tous les membres du LIP6 avec qui j'ai pu partager, échanger et passer des moments agréables. Mes visites à Jussieu n'ont jamais été nombreuses, mais ils ont toujours fait preuve d'attention à mon égard. Tout particulièrement, merci à Andy, Emmanuelle, Jean-Lou, Karine, Lilia, Olivier, et Roselyne. Enfin, un grand merci à Mohamed et Thomas.

Mes vifs remerciements à Louis Mandel et Adrien Guatto de l'ENS Ulm ; merci pour ces échanges propices et captivants.

Pour finir, J'adresse un grand merci à toutes les personnes qui, par leur soutien, leur amitié ou leur amour, m'ont permis de mener cette thèse à bien. Merci à mes parents, et à mes frères et soeurs, qui malgré la distance ont toujours été présents pour moi. Merci à ma femme, qui a supporté jusqu'au bout cette étrange période de ma vie. Et enfin, merci à ma fille qui n'a jamais manqué une opportunité de me réveiller en pleine nuit pour me permettre de travailler un peu plus.

Résumé

Les systèmes embarqués sont des équipements électroniques et informatiques, soumis à de nombreuses contraintes et dont le fonctionnement doit être continu. Pour définir le comportement de ces systèmes, les modèles de programmation dataflows sont souvent utilisés. Ce choix de modèle est motivé d'une part, parce qu'ils permettent de décrire un comportement *cyclique*, nécessaire aux systèmes embarqués ; et d'autre part, parce que ces modèles s'apprêtent à des *analyses* qui peuvent fournir des garanties de fonctionnement et de performance essentielles.

La société Kalray propose une architecture embarquée, le MPPA. Il est accompagné du langage de programmation ΣC. Ce langage permet alors de décrire des applications sous forme d'un modèle dataflow déjà très étudié, le modèle *Cyclo-Static Dataflow Graph* (CSDFG). Cependant, les CSDFG générés par ce langage sont souvent trop complexes pour permettre l'utilisation des techniques d'analyse existantes.

L'objectif de cette thèse est de fournir des outils algorithmiques qui résolvent les différentes étapes d'analyse nécessaires à l'étude d'une application ΣC, mais dans un temps d'exécution raisonnable, et sur des instances de grande taille. Nous étudions trois problèmes d'analyse distincts : le test de vivacité, l'évaluation du débit maximal, et le dimensionnement mémoire. Pour chacun de ces problèmes, nous fournissons des méthodes algorithmiques rapides, et dont l'efficacité a été vérifiée expérimentalement.

Les méthodes que nous proposons sont issues de résultats sur les ordonnancements périodiques ; elles fournissent des résultats approchés et sans aucune garantie de performance. Pour pallier cette faiblesse, nous proposons aussi de nouveaux outils d'analyse basés sur les ordonnancements K-périodiques. Ces ordonnancements généralisent nos travaux d'ordonnancement périodiques et nous permettrons dans un avenir proche de concevoir des méthodes d'analyse bien plus efficaces.

Mots-clefs

multiprocesseurs ; compilation ; flot de données ; *Cyclo-Static Dataflow Graphs* ; ordonnancement ; périodique ; k-périodique ; vivacité ; évaluation du débit ; dimensionnement mémoire ; méthode approchée

Abstract

Embedded systems are hardware and software based equipment. They are subject to many constraints and must run without stopping. To define the behavior of these systems, dataflow programming models are often used. On one hand, this choice is motivated by the fact that dataflow models allow the description of cyclic behavior, which is needed for embedded systems ; secondly because the analysis of these models can provide essential guarantees of correctness and performance.

The Kalray company provides an embedded architecture : the MPPA. It is accompanied by the ΣC programming language. This language allows to describe applications in the form of a well-known dataflow model : the Cyclo-Static Dataflow Graph (CSDFG). However, the dataflow graphs that are generated by this language are often too complex to be analysed with existing techniques.

The objective of this thesis will be to provide algorithmic tools that solve the various stages of ΣC application analysis, but within a reasonable execution time, as on large instances. We study three different problems : the liveness, the throughput evaluation, and the buffer sizing. For each of these problems, we provide fast algorithmic methods, and we have experimentally verified their efficiency.

The proposed methods are based on periodic scheduling. Therefore they provide approximate results without any guarantee of optimality. To overcome this weakness, we also offer new analysis tools based on K-periodic scheduling. This result generalizes our previous work and will allow us in the near future to design more efficient analyzing methods.

Keywords

multiprocessor ; compilation ; dataflow ; Cyclo-Static Dataflow Graphs ; scheduling ; periodic ; K-periodic ; liveness ; throughput evaluation ; buffer sizing ; approximate method

Table des matières

Liste des tableaux

Table des figures

Chapitre 1

Introduction

Les systèmes embarqués

Un système embarqué est un équipement électronique et informatique spécialisé dans une tâche précise. Il est autonome et limité en taille, en consommation électrique, et en dissipation thermique. Les premiers systèmes embarqués firent leur apparition dans les années soixante ; ils étaient particulièrement coûteux, et n'étaient utilisés que dans les domaines de l'aérospatiale et de l'armement. Par la suite, ces systèmes furent appliqués à des domaines bien plus vastes, aujourd'hui ils sont incontournables. Qu'il s'agisse d'un simple réveil-matin ou d'une automobile, d'un téléphone ou d'un téléviseur, tous ces équipements contiennent un ou plusieurs systèmes embarqués ou, plus communément, *System on Chip* (SoC).

Un système embarqué se divise en deux parties, l'une matérielle (l'électronique), l'autre logicielle (l'informatique). Le matériel délimite les interfaces et les performances du système tandis que le logiciel définit la ou les fonctions réalisées par ce système.

La complexité matérielle de ces équipements n'a cessé de croître, et les systèmes embarqués actuels comptent souvent plusieurs unités de calcul leur permettant de traiter des opérations en simultané. Ce sont des *Multi-Processor System on Chip* (MPSoC). La société Kalray propose notamment une architecture embarquée composée de plus de 256 cœurs de calcul et plus d'une vingtaine d'interfaces différentes. Avec une dissipation thermique proche de 5 Watt, ce processeur est alors parmi les plus performants du marché.

Cependant, les outils de conception logicielle actuels ne sont pas adaptés à ces architectures multicœurs. Pour cette raison, la société Kalray a aussi fait le choix de proposer la chaîne de compilation AccessCore. Il s'agit d'un outil de conception adapté à cette architecture. Il est accompagné d'un langage de programmation dit *dataflow* (ou flot de données) appelé le ΣC.

La programmation dataflow

Les modèles de programmation standards sont généralement séquentiels. Ils définissent un programme par une succession d'opérations. Dans le programme de la Figure 1.1 page ci-contre, les opérations `op1`, `op2`, `op3` et `op4` vont ainsi se succéder infiniment. Des dépendances entre ces opérations sont induites par cet ordre d'exécution. L'exécution parallèle de ces opérations n'est donc pas naturelle.

```
1  while (true) {
2    out1   = op1(input);
3    out2   = op2(out1);
4    out3   = op3(out1);
5    output = op4(out2,out3);
6  }
```

FIGURE 1.1 – Un exemple de programme séquentiel dans le langage de programmation C.

La modélisation dataflow pallie ce problème de parallélisation en permettant de spécifier clairement les dépendances de données entre les différentes opérations. Un dataflow modélise une application sous forme d'un graphe dans lequel, chaque nœud est une tâche (ou un ensemble d'opérations), et les arcs sont des canaux de communication entre ces tâches. La Figure 1.2 est un exemple de dataflow. On remarque que ces différentes tâches peuvent alors s'exécuter en parallèle.

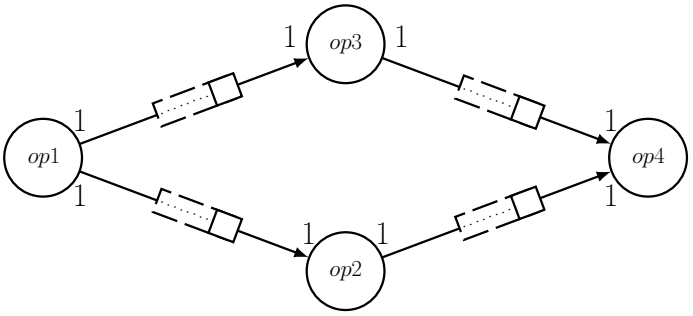


FIGURE 1.2 – Un exemple d'application dataflow.

Pour certains modèles dataflows, les quantités de données produites et consommées dans ces canaux sont prédéterminées : on parle de modèles statiques. Le principal intérêt de ces modèles statiques est de permettre leur analyse sans avoir à effectuer de simulation.

Le langage ΣC (proposé par la société Kalray) permet de décrire des applications sous forme d'un dataflow statique particulier : le modèle *Cyclo-Static Dataflow Graph* (CSDFG). La chaîne de compilation AccessCore est ainsi en mesure d'évaluer *a priori* les performances d'un programme ou l'espace mémoire nécessaire à son exécution.

Cependant, les CSDFG extraits à partir des applications ΣC sont trop grands pour nous permettre d'utiliser des techniques d'analyse exactes. La complexité de ces méthodes est trop élevée.

L'objectif principal de cette thèse est d'apporter des solutions d'analyse efficaces pour la chaîne de compilation AccessCore. Compte tenu de l'ampleur de ce travail, nous avons concentré nos efforts sur trois axes distincts :

– le test de vivacité,

– l'évaluation du débit maximal,

– et le dimensionnement mémoire.

La vivacité

Lorsqu'il n'y a pas suffisamment de données pour exécuter les tâches d'un dataflow, celui-ci peut atteindre un état de blocage. Si cet état de blocage n'existe pas, on dit alors qu'une application est vivante. Nous proposons une méthode efficace pour vérifier cette propriété pour le modèle CSDFG.

L'évaluation du débit maximal

Si chaque tâche d'une application dataflow a une durée d'exécution maximale connue (souvent nommée le *Worst Case Execution Time*), il est possible d'évaluer la vitesse de fonctionnement maximale de cette application. Nous proposons une solution d'évaluation approchée de cette valeur.

Le dimensionnement mémoire

L'espace disponible pour faire circuler des données dans les canaux de communication d'un dataflow est en théorie infini. Ce n'est plus vrai pour de véritables applications. Nous cherchons alors à déterminer la quantité minimale de mémoire nécessaire pour qu'une application puisse fonctionner correctement et atteigne les performance attendues. Nous proposons différents algorithmes dans ce sens.

Plan de la thèse et contributions

Dans le chapitre 2, nous mettons en avant le contexte et les motivations de nos travaux. Nous y présentons l'architecture MPPA et la chaîne de compilation AccessCore, deux produits commercialisés par la société Kalray.

Nous commençons par fournir une classification des différentes architectures embarquées existantes. Nous présentons ensuite les principaux outils disponibles pour programmer ces architectures. Dans un second temps, nous considérons la modélisation dataflow comme une solution efficace de la programmation parallèle, et nous énumérons quelques-uns des modèles dataflows utilisés aujourd'hui. Nous terminons alors sur la présentation de la chaîne de compilation AccessCore, et nous situons les étapes sur lesquelles portent nos contributions.

Dans le chapitre 3, nous spécifions les problématiques concernées par nos travaux. Nous définissons tout d'abord des résultats d'analyse fondamentaux, puis nous énumérons un à un chaque problème étudié. Pour chacun, nous considérons les méthodes existantes pour

les résoudre. Nous remarquons alors que toutes les techniques d'analyse exactes appliquées aux CSDFG se basent sur un ordonnancement *au plus tôt*. Ce type d'ordonnancement exécute chaque tâche d'un dataflow dès qu'elle dispose de suffisamment de données en entrée. La Figure 1.3 en est un exemple.

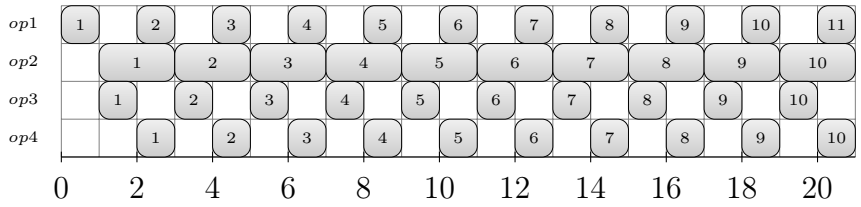


FIGURE 1.3 – Un exemple d'ordonnancement pour le dataflow de la Figure 1.2 page 3

La suite de ce manuscrit s'articule en deux parties. La première partie traite de nos contributions les plus théoriques, qui portent sur l'ordonnancement des modèles dataflows. Dans la seconde partie, ces résultats sont appliqués à l'analyse des modèles CSDFG. Nous détaillons les travaux d'intégration de ces méthodes dans la chaîne de compilation AccessCore.

Première partie

Le premier chapitre de cette partie, le chapitre 4, présente une méthode polynomiale pour vérifier la faisabilité de l'ordonnancement périodique d'un CSDFG. Il s'agit d'une contribution fondamentale pour nos travaux, et pour le domaine de l'ordonnancement des modèles CSDFG. Contrairement aux ordonnancements *au plus tôt*, qui peuvent être de taille exponentielle, la définition d'un ordonnancement périodique est de taille polynomiale. Dans ce chapitre, nous prouvons que le calcul d'un ordonnancement périodique pour un CSDFG est un problème polynomial. Ce résultat est important ; il va nous permettre, dans la suite de nos travaux, d'appliquer les ordonnancements périodiques à l'analyse des dataflows, et de fournir des techniques approchées bien plus rapides que les méthodes exactes connues (puisque la complexité des ordonnancements périodiques est bien plus faible que celle des ordonnancements *au plus tôt*).

La méthode d'ordonnancement précédemment proposée ne nous permet d'obtenir que des solutions d'analyse approchées. Pour améliorer nos résultats, nous étudions dans le chapitre 5 une nouvelle approche : les ordonnancements K-périodiques. Pour tout ordonnancement K-périodique, l'on doit définir un vecteur de périodicité K. Ainsi, plus les valeurs contenues dans ce vecteur sont pertinentes, plus l'ordonnancement K-périodique généré peut être précis. De cette manière, cette famille d'ordonnancement généralise les ordonnancements périodiques (avec un vecteur de périodicité minimal) et *au plus tôt* (avec un vecteur de taille maximale). Nous proposons une technique de calcul d'un

ordonnancement K-périodique pour un vecteur périodicité fixé, et nous étudions l'impact du vecteur de périodicité sur les performances des ordonnancements obtenus.

Il est important de noter que le contexte industriel de nos travaux a fortement influencé la direction de ces recherches. Les résultats théoriques d'ordonnancement obtenus ici furent donc appliqués à l'analyse des applications ΣC. La seconde partie de cette thèse est consacrée à ces applications.

Deuxième partie

Le premier chapitre de cette partie, le chapitre 6, présente les différents problèmes d'analyse traités tout au long de cette thèse. Il définit différents algorithmes, et présente les expérimentations réalisées pour évaluer leurs performances. Dans un premier temps, nous proposons un nouvel outil utilisé pour générer nos jeux de test. Puis, dans la continuité des travaux de Benazouz [11], nous proposons une solution algorithmique pour tester la vivacité d'un CSDFG. Nous proposons ensuite une méthode d'évaluation du débit d'un CSDFG ainsi qu'une méthode de dimensionnement mémoire d'un CSDFG. Ces techniques appliquent alors directement les contributions exposées dans le chapitre 4.

Après le développement de ces différentes solutions d'analyse, nous avons considéré leur intégration dans la chaîne de compilation AccessCore. Le chapitre 7 porte sur cette intégration. En effet, le langage dataflow proposé par la société Kalray est en perpétuelle évolution. Bien qu'il respecte toujours les contraintes d'un modèle statique, il ne correspond plus à la définition d'un modèle CSDFG. Certaines contraintes supplémentaires sur l'exécution des tâches furent notamment ajoutées. Pour intégrer avec succès les différentes méthodes d'analyse proposées dans le chapitre précédent, il était donc nécessaire d'étendre nos méthodes à ce modèle plus expressif.

Pour finir, dans le chapitre 8, nous concluons ce manuscrit et proposons différentes pistes d'amélioration. Nous envisageons également l'application de nos approches à d'autres domaines.

Dans les dernières pages, nous proposons également un index des notations utilisées dans ce mémoire.

Chapitre 2

Contexte

Introduction

Les systèmes embarqués sont omniprésents dans notre quotidien. Aujourd'hui, il n'y a (presque) plus un instant sans que nous ne sollicitions des équipements complexes, qui requièrent des systèmes informatiques spécialisés communément appelés « systèmes embarqués » ou *System on Chip* (SoC).

La conception de ces systèmes passe le plus souvent par une chaîne de production complexe, sollicitant chercheurs, ingénieurs et techniciens afin de produire le logiciel et le matériel de l'équipement final. Historiquement, un système embarqué est issu de l'assemblage de différents composants, chacun fournissant une fonctionnalité de base comme la mémoire, les interfaces réseau, ou les unités de calcul. Une fois ces éléments rassemblés, on peut alors produire l'*Application Specified Integrated Circuit* (ASIC), le circuit final qui réalise les tâches particulières pour lesquelles le système a été pensé.

Avec la démocratisation de la technologie embarquée, les demandes de performance pour ces équipements n'ont eu de cesse de croître. On attend par exemple des systèmes embarqués plus de puissance ou une consommation énergétique réduite. Cependant, le coût de la technologie et du savoir-faire nécessaire pour produire de tels ASIC n'est pas proportionnel à la performance des systèmes obtenus. Ces désagréments peuvent être estompés par une production de masse, comme c'est le cas pour le marché du mobile. Néanmoins dans le cadre des productions de petites et moyennes séries cela n'est pas possible ; des alternatives ont dû être mises en place.

L'utilisation de microcontrôleurs fut l'une des premières solutions envisagées pour diminuer ces coûts de production. Ce sont des équipements standards auxquels l'on peut associer n'importe quelle tâche et pour un coût de développement extrêmement faible. Cependant, il s'agit le plus souvent de processeurs peu performants et qui ne permettent pas de fournir la même puissance qu'un ASIC.

Les *Digital Signal Processor* (DSP) peuvent aussi être utilisés. Il s'agit de processeurs améliorés qui disposent d'un ensemble d'unités de calcul fournissant des fonctions spécialisées comme l'encodage multimédia. Cependant, les DSP ne sont pas très flexibles et ne fournissent qu'un ensemble limité de fonctions.

Les *Field-Programmable Gate Array* (FPGA) sont alors l'une des alternatives les plus plébiscitées. Il s'agit de composants reprogrammables pour lesquels l'on peut appliquer n'importe quelle fonctionnalité ASIC. Les FPGA restent néanmoins une technologie coûteuse et plutôt volumineuse. D'autre part, le temps de conception logicielle nécessaire pour un FPGA reste presque aussi élevé que pour un ASIC traditionnel.

Plus récemment, des améliorations technologiques nous permettent d'envisager une nouvelle solution, celle des architectures multiprocesseurs : les *Multi-Processors System on*

Chip (MPSoC). Ces systèmes disposent d'un nombre de cœurs de calcul élevé et fournissent suffisamment de puissance pour s'affranchir des ASIC ou des FPGA.

Fondée en 2008, la société Kalray a misé sur cette technologie et elle commercialise aujourd'hui le *Multi-Purpose Processor Array* (MPPA). Il s'agit d'un processeur multicœur destiné au segment de marché des petites et moyennes productions.

Dans ce chapitre, nous allons présenter la famille d'architectures multiprocesseurs dont fait partie le MPPA ainsi que les outils de développement qui lui sont associés. L'architecture du MPPA est particulièrement complexe, et pour en exploiter au mieux les performances, différents modèles de programmation furent envisagés.

2.1 Les architectures matérielles en informatique

L'une des étapes les plus critiques dans la conception d'un système embarqué est très certainement le choix de son architecture matérielle. C'est elle qui va définir les performances du système (comme sa réactivité) et ses interfaces vers l'extérieur (qu'il s'agisse de capteurs thermiques, d'une entrée vidéo, d'un port réseau...).

Pour la réalisation d'un ASIC, un concepteur est libre de ses choix. La conception FPGA quant à elle peut souvent diminuer cette latitude de possibilité ; les interfaces seront par exemple imposées. Cependant, c'est lorsque l'on utilise un DSP ou un microcontrôleur qu'on dispose du moins de flexibilité : toutes les fonctions sont fixées à l'avance.

Pour guider son choix, un concepteur a besoin de pouvoir différencier les processeurs existant afin d'identifier celui qui lui permettra d'atteindre les performances qu'il souhaite.

En 1945, le modèle d'architecture processeur dit « de Von Neumann » fut proposé [89]. Celui-ci considère le processeur comme l'interconnexion de trois composants :

- une mémoire,
- une unité de calcul,
- et une unité de contrôle.

L'unité de contrôle va exécuter l'une après l'autre des instructions définies par un programme en mémoire. Ces instructions définissent alors des opérations qui seront réalisées par l'unité de calcul, et qui dans certains cas modifient l'état de la mémoire.

À l'opposé d'une architecture de Harvard (où les données et le programme sont dans des espaces distincts), la principale nouveauté du modèle de Von Neumann [89] est donc d'utiliser une seule et unique mémoire à la fois pour le programme et les données.

La majeure partie des architectures d'aujourd'hui s'inspire de ce modèle. Elles diffèrent cependant, et intègrent parfois plusieurs unités de calcul. Dans la section suivante, nous présentons des classifications nous permettant de les différencier.

2.1.1 Classification de Flynn

En 1972, Flynn [37] propose une taxinomie des architectures basée sur la sémantique de leurs opérations. Deux critères sont alors utilisés : le nombre de données concernées par chaque instruction (une ou plusieurs) et le nombre d'opérations effectuées par chaque instruction (une ou plusieurs).

On compte 4 combinaisons possibles :

- SISD (*Single Instruction Simple Data*),
- SIMD (*Single Instruction Multiple Data*),
- MISD (*Multiple Instruction Simple Data*),
- MIMD (*Multiple Instruction Multiple Data*).

Tandis que le type SISD correspond à une architecture de Von Neumann [89], les architectures modernes s'apparentent souvent aux types SIMD, MISD ou MIMD. Qui plus est, avec l'avènement du parallélisme, on trouve désormais des architectures multicœurs où chaque cœur peut s'avérer être une architecture MIMD. Cette absence de nuance entre les architectures nous amène donc à utiliser un second système de classification plus adapté aux architectures multiprocesseurs, celui de Raina [72].

2.1.2 Classification de Raina

Dans les architectures modernes, une interconnexion généralisée entre la mémoire et les autres composants n'est pas toujours assurée. L'accès aux mémoires par une unité de calcul peut alors prendre différentes formes. Pour identifier ces différences, Raina [72] propose deux nouveaux critères : l'emplacement physique des mémoires (partagées ou distribuées) et le mode d'adressage utilisé pour y accéder (partagé ou disjoint).

- *Single Address space and Shared Memory* (SASM) : Espace d'adressage et mémoires sont partagés, toutes les zones mémoires sont accessibles par toutes les unités de calcul.
- *Single Address space and Distributed Memory* (SADM) : La mémoire est distribuée, mais l'espace d'adressage est global, bien que l'information soit distribuée sur l'ensemble de l'architecture, les accès à ces données peuvent se faire par n'importe quelle unité de calcul.
- *Disjoint Address space and Distributed Memory* (DADM) : Ici, chaque élément de calcul dispose de son propre espace d'adressage, l'échange de données ne peut donc se faire que par des mécanismes de passage de messages.

– *Disjoint Address space and Shared Memory* (DASM) : Bien qu'il ne soit pas toujours référencé, ce cas reste possible. Raina propose notamment l'exemple des *Remote Procedure Call*[1] (RPC) sur une machine à mémoire partagée.

Architectures à système de mémoire partagée (SASM)

Dans ce type d'architecture, les processeurs sont tous reliés à une même zone mémoire. En dehors de tout conflit dû à des requêtes simultanées, l'hypothèse est faite que les temps d'accès mémoire sont les mêmes pour tous les processeurs ; Raina parle d'architecture UMA (*Uniform memory Access*).

Cette contrainte de mémoire partagée induit généralement de fortes limitations dans le nombre de processeurs disponibles. En effet, il est techniquement difficile de relier un nombre élevé de processeurs à un même banc mémoire.

Exemple : le processeur Intel Xeon E5520 Le Xeon est l'un des processeurs généralistes (*Central Process Unit* ou CPU) les plus utilisés dans les postes de travail haut de gamme et les serveurs. Il est un bon exemple de ce que peut être une architecture UMA. Compte tenu de la topologie de ses unités de calcul (voir Figure 2.1), on parlera même ici d'architecture *Symmetric Multiprocessors* (SMP).

Cette version du Xeon se compose de 4 ou 8 processeurs cadencés à plus de 2GHz. Chaque processeur dispose de deux niveaux de mémoire interne (cache L1 et L2), mais ils sont tous reliés à une mémoire partagée de 8 Mo (cache L3).

Notons que cette architecture dispose aussi d'un mécanisme de *cohérence de cache*. La cohérence de cache permet aux cœurs de calcul de modifier des données situées en mémoire partagée directement dans leur mémoire interne tout en garantissant une mise à jour ultérieure des données partagées.

FIGURE 2.1 – L'architecture simplifiée d'un processeur Xeon E5520. La mémoire partagée est en blanc, les cœurs de calcul sont en gris.

1. Il s'agit d'un protocole réseau permettant d'exécuter des fonctions à distance.

Architectures à mémoire distribuée (SADM, DADM)

Contrairement aux architectures précédentes, dans les architectures à mémoire distribuée, les unités de calcul ne partagent pas de segment mémoire. Pour communiquer, elles devront donc utiliser des techniques de passage de message.

Dans le cas de l'architecture DADM, les zones mémoire de chaque unité de calcul sont privées et Raina parle donc d'architecture mémoire *No Remote Memory Access*(NORMA).

Pour les architectures SADM, 4 sous-classes sont proposés par Raina :

- SADM-NUMA *Non-Uniform Memory Access* : La performance des accès mémoire varie en fonction de caractéristiques comme la localisation des données.
- SADM-CC *Cache-Coherent* : Architecture de type NUMA intégrant un mécanisme de cohérence de cache intégré à l'architecture.
- SADM-OS *Operating System* : La prise en charge de la cohérence de cache doit être assurée par le système d'exploitation.
- SADM-COMA *Cache Only Memory Access* : L'accès à des données distantes se fait uniquement au travers du cache (cohérence matérielle).

Exemple : Tile64 La société Tilera propose depuis quelques années un processeur à mémoire distribuée avec cohérence de cache (SADM-CC), le Tile64 [91]. Celui-ci se compose d'une grille de 64 processeurs (8×8) cadencés à une fréquence de 700Mhz et directement reliés par un réseau sur puce (NoC, *Network on Chip*) dont la topologie est visible sur la Figure 2.2 page ci-contre. Chaque processeur est *Very Long Instruction Word* (VLIW) et dispose de trois voies d'opération (3-way), cela signifie qu'un processeur peut effectuer jusqu'à trois opérations concurrentes par instruction. Ici, le cache est équitablement réparti sur l'ensemble de la puce pour un total de 23 Mo.

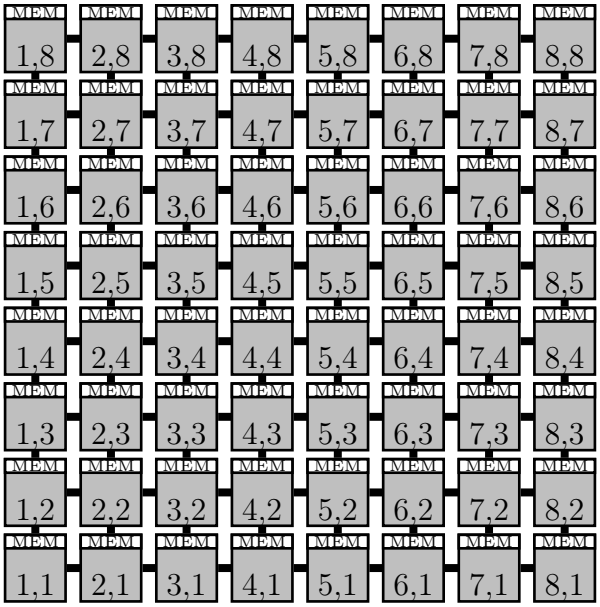


FIGURE 2.2 – Architecture simplifiée d'un processeur Tilera Tile64. Les mémoires sont en blanc, les cœurs de calcul sont en gris.

2.1.3 L'architecture multiprocesseur MPPA

Le MPPA de la société Kalray (Figure 2.3 page suivante) est une architecture multiprocesseur à mémoire distribuée. Contrairement au Tile64, les processeurs du MPPA sont distribués sur un ensemble de 16 « clusters » au travers d'un réseau sur puce.

Chacun de ces clusters est un regroupement de 16 unités de calcul de type 5-Way VLIW. Ces unités de calcul sont cadencées à 400Mhz, nous les appellerons les *Processing Element* (PE). Ils sont accompagnés d'un PE supplémentaire considéré comme le *Ressource Manager* (RM). Quatre clusters supplémentaires sont aussi disponibles afin de gérer les interfaces de l'architecture ; nous les appellerons les *IO Clusters*. Enfin, on trouve sur chaque cluster une mémoire de 2 Mo, pour un total de 40 Mo. Le MPPA combine donc les caractéristiques d'une architecture NORMA avec des clusters individuellement plus proche d'une architecture UMA ; il s'agit donc d'une architecture hybride.

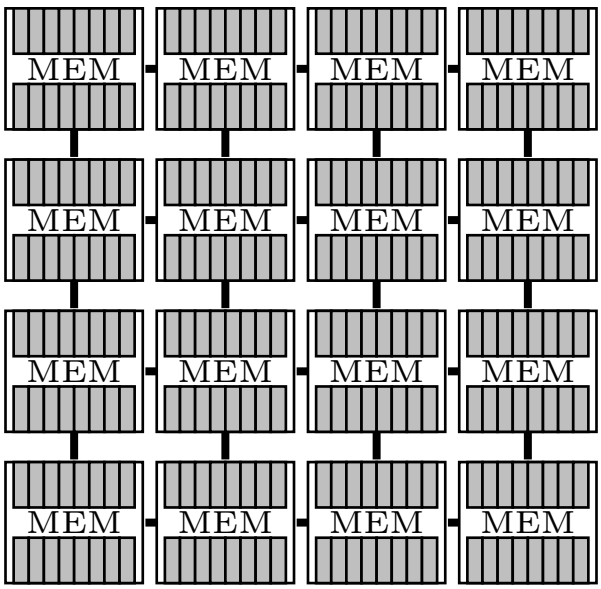


FIGURE 2.3 – Architecture simplifiée d'un processeur MPPA. Les mémoires sont en blanc, les cœurs de calcul sont en gris.

2.2 La programmation parallèle

Les architectures multiprocesseurs disposent généralement de jeux d'instructions permettant d'en exploiter le parallélisme. Pour le processeur Xeon, on parlera par exemple d'*Advanced Programmable Interrupt Controllers* (APIC). Il existe cependant une multitude de mécanismes différents et leur utilisation est extrêmement complexe.

De plus, les langages de programmation les plus populaires (comme les langages C/C++ ou Java) sont séquentiels : ils exécutent une à une les opérations d'un programme en s'apparentant fortement à une architecture de Von Neumann. La parallélisation automatique de tels programmes est une tâche réputée difficile [76].

Il existe bien heureusement des surcouches logicielles qui permettent aux utilisateurs de décrire eux même le parallélisme de leurs programmes. On présentera pour commencer des interfaces logicielles bas niveau comme POSIX ou MPI puis une interface de plus haut niveau, OpenMP, et le langage de programmation StreamIt.

2.2.1 Les normes POSIX et MPI

POSIX Threads

La norme *Portable Operating System Interface* (POSIX) définit une interface ou *Application Programming Interface* (API) supportée par la plupart des systèmes d'exploitation actuels (comme Unix, Windows et Mac OS). Cette norme comprend POSIX Thread [24], une bibliothèque de fonction permettant de définir des processus d'exécution concurrents

(*threads*) accédant à une zone mémoire commune. Les *threads* s'adaptent donc parfaitement aux architectures à mémoire partagée.

Exemple Le programme en Figure 2.4 double les valeurs d'un tableau `tab` à l'aide de deux *threads*. La création des *threads* se fait avec `pthread_create()` ; on utilise ensuite `pthread_join()` pour attendre la fin d'un *thread*.

```
1  void *mythread(void *arg) {
2      int * tab = (int*) arg;
3      for (int * p = tab; p < tab + 100; p++) *p = *p * 2;
4  }
5  ...
6  pthread_create(&th1, NULL, mythread,  (void*)(tab));
7  pthread_create(&th2, NULL, mythread,  (void*)(tab+100));
8
9  pthread_join (th1);
10 pthread_join (th2);
```

FIGURE 2.4 – Une fonction doublant les valeurs d'un tableau `tab` à l'aide de *threads* POSIX.

Une programmation complexe L'exécution concurrente de *threads* peut induire une absence de déterminisme [54]. Afin d'éviter, ou de contrôler ce comportement, l'on doit utiliser des outils de synchronisation, mais leur manipulation est complexe. Lee [54] pose clairement ce problème et expose quelques cas simples comme les interblocages [2]. De plus, comme ce modèle est par nature indéterministe, des comportements non prédictibles peuvent rentre l'observabilité et la correction d'erreur particulièrement difficile.

MPI

Tout comme POSIX Thread, *Message Passing Interface* (MPI) est une norme définissant une API, mais plus adaptée aux architectures à mémoire distribuée [64]. Un programme utilisant MPI est décrit comme un ensemble de processus qui communiquent à l'aide des commandes `MPI_Send()` et `MPI_Recv()`. C'est un mécanisme proche de ce qui doit être fait lorsque l'on communique sur un réseau sur puce (comme pour le processeur Tile64). Le développement avec MPI permet alors de décrire des processus communicants, sans pour autant devoir nécessairement gérer leur synchronisation.

Exemple Un programme MPI s'exécute sur l'ensemble des processeurs disponibles et adapte son comportement selon l'identifiant du processeur courant (récupéré par `MPI_Comm_rank()`). Dans l'exemple en Figure 2.5 page suivante, le processus MPI sur le

2. Situation de blocage souvent causée par une mauvaise utilisation des techniques de synchronisation. Un exemple classique d'interblocage est celui de deux *threads* en attente de fin mutuelle.

processeur 0 va produire des données pour les processus restants. La communication se fera à l'aide des commandes `MPI_Send` et `MPI_Recv`.

```
1       MPI_Comm_size(&numprocs);
2       MPI_Comm_rank(&myid);
3
4       if (myid==0) /* le processus 0 envoie */
5       {
6         for (target = 1; target < numprocs; target++)
7             MPI_Send(tab + 100 * (target-1), target);
8       } else {     /* les autres processus recoivent */
9         int * tab;
10        MPI_Recv(tab);
11        for (int * p = tab; p < tab + 100; p++) *p = *p * 2;
12      }
```

FIGURE 2.5 – Un programme MPI simplifié.

Une portabilité limitée Lorsque l'on décrit le parallélisme d'une application, certains choix peuvent dépendre de l'architecture sur laquelle le programme s'exécutera. Ainsi, pour deux architectures différentes, et afin de maximiser les performances d'un programme, deux implémentations devront être réalisées.

2.2.2 Des interfaces de plus haut niveau

Pour éviter la rigidité de ces standards, plusieurs solutions logicielles (langages ou extensions) furent proposées. Bien qu'il en existe un très grand nombre, nous n'en présenterons que quelques-unes parmi les plus représentatives.

OpenMP

OpenMP [1] est une extension de langage développée depuis 1997 afin de fournir aux développeurs une gestion simplifiée du parallélisme dans leurs applications. Elle est constituée d'un ensemble de directives de compilation (commençant toutes par `#pragma omp`) permettant de caractériser des zones de code automatiquement parallélisables. Bien que OpenMP soit dédié aux architectures à mémoire partagée, il existe un certain nombre de projets visant à en augmenter la portée, tels que OpenMPC, une extension de OpenMP dédiée aux *Graphics Processing Unit* (GPU), ou bien OpenMPI, une implémentation de MPI destinée aux utilisateurs de OpenMP.

Exemple Si l'on dispose d'une boucle sans dépendance portée[3], il est possible d'indiquer à l'aide d'une directive OpenMP que cette boucle est entièrement parallélisable. Dans la

3. Il y a dépendance portée si une itération de boucle dépend du résultat d'une autre itération.

Figure 2.6, la directive `#pragma omp parallel for` indique donc que toutes les itérations de la boucle `for` peuvent être exécutées en parallèle.

```
1  #pragma omp parallel for
2   for (i = 0; i < N; i++)
3          tab[i] = 2 * i;
```

FIGURE 2.6 – Un exemple de boucle classique, facilement parallélisable à l'aide d'OpenMP.

Selon nous, bien que cette solution de parallélisation soit très séduisante (elle est d'ailleurs l'une des plus plébiscitées), il subsiste un effet pervers à son utilisation : le concepteur risque de s'enfermer dans un modèle de programmation séquentiel, et ne jamais pouvoir atteindre le parallélisme maximum de son application. En effet, dans certains cas, une parallélisation avec OpenMP peut exiger la réécriture du programme, et elle n'est pas toujours évidente. Prenons l'exemple de deux boucles qui se suivent ; chacune se parallélisera séparément, mais l'une après l'autre. Il serait possible de les paralléliser ensemble, en les regroupant, mais cela n'est souvent possible que sous certaines conditions.

StreamIt

Contrairement aux API de support comme les *threads* ou OpenMP, StreamIt [86] est un langage dit *dataflow*. Traditionnellement, cette famille de langages s'oppose aux langages impératifs par des restrictions supplémentaires concernant l'absence d'effet de bord ou l'assignation multiple des variables [92]. En pratique, la programmation dataflow revient à décrire un ensemble de tâches autonomes communiquant entre elles exclusivement par l'intermédiaire de canaux de communication (ou *buffers*).

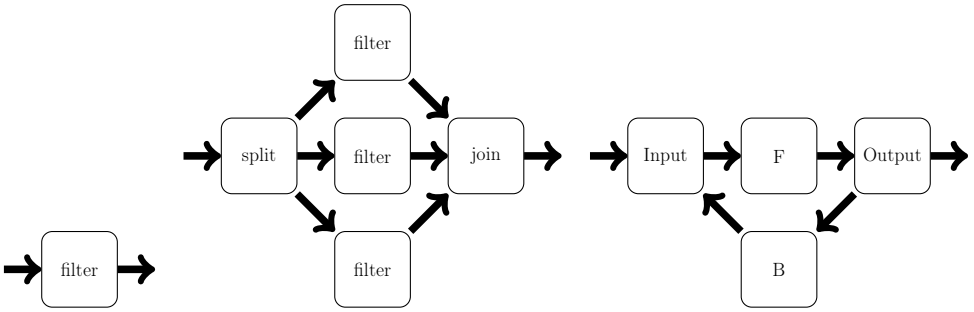
Dans StreamIt, la construction d'une application se fait par assemblage de structures prédéfinies, les `filter`, les `splitjoin` et les `loop`, en les reliant par des canaux de communication (une représentation graphique de ces opérateurs est visible en Figure 2.7 page suivante).

Filter Un `filter` est un agent disposant d'une unique entrée et d'une unique sortie. Cet agent s'exécutera chaque fois qu'il disposera de suffisamment de données dans son buffer d'entrée. La quantité de données nécessaire à son exécution est constante et prédéfinie (il existe une extension de ce langage où ces taux peuvent varier). À chacune de ses exécutions, il consommera des données en entrée et produira des données en sortie, ces quantités seront aussi constantes et prédéfinies.

Splitjoin Un `splitjoin` dispose d'une unique entrée et d'une unique sortie. C'est une composition d'agents. Un premier agent `split` va distribuer les données d'entrée vers

plusieurs `filter` prédéfinis. Les données produites par ces agents sont ensuite rassemblées par un agent `join` et copiées en sortie du `splitjoin`.

Loop Tout comme les `splitjoin`, le `loop` dispose d'une unique entrée, d'une unique sortie, et se compose de quatre agents. Un agent `input` va copier ses données d'entrée vers un agent `F`. Les données produites par `F` sont ensuite envoyées vers un agent `output` et un agent `B`. Pour finir, les données produites par `B` sont renvoyées vers `input` qui les incorporera aux nouvelles données d'entrée de `F`.



(a) `filter` (b) `splitjoin` (c) `loop`

FIGURE 2.7 – Représentation graphique des différents opérateurs StreamIt

Exemple Cette fois-ci notre exemple prend la forme d'un programme dataflow où un `filter` (`Reader` sur les Figures 2.8 pour le listing et 2.9 page suivante pour sa représentation graphique) distribue à l'aide d'un `splitjoin` le contenu du tableau à d'autres `filter` (`Worker`), lesquelles renverront leurs résultats à un dernier `filter`, `Writer`.

```
1  float->float filter Worker() {
2  work push 1 pop 1 {
3  push(2 * pop() );
4  }
5  }
6
7  int->int splitjoin MySJ {
8    split duplicate;
9    add Worker();
10   join roundrobin;
11 }
12
13 int->int pipeline Root {
14   add Reader();
15   add MySJ();
16   and Writer();
17 }
```

FIGURE 2.8 – Un exemple de programme StreamIt

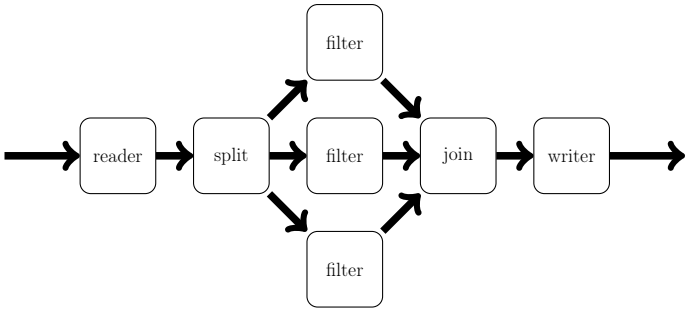


FIGURE 2.9 – Répresentation graphique du programme StreamIt de la Figure 2.8 page précédente.

Les modèles de programmation dataflow comme StreamIt présentent plusieurs atouts pour la programmation des systèmes embarqués. Ils permettent notamment de limiter les erreurs de programmations liées aux interblocage et font abstraction du système sur lequel ils seront exécutés.

2.2.3 Le langage ΣC

En plus de fournir une interface proche du standard POSIX, la société Kalray propose également à ses clients un langage de programmation de plus haut niveau, le ΣC [45]. Le langage ΣC (projet du laboratoire CEA/LIST) est une extension du langage C disposant d'un certain nombre d'opérateurs spéciaux.

Au même titre que StreamIt, ΣC est un langage dataflow. Il est cependant plus généraliste ; les agents disposent d'un nombre quelconque d'entrées/sorties, et les quantités produites et consommées par ses agents sont désormais variables.

Nous verrons plus tard dans la section 2.3.2 page 24 que le langage ΣC respecte un modèle dataflow particulier, le *Cyclo-Static Dataflow Graph* (CSDFG). Parmi ses avantages, notons que ce modèle est déterministe (les données produites par le programme ne dépendent que des données d'entrée) et statique (plusieurs problèmes d'analyse comme l'existence d'interblocages deviennent décidables).

En terme de syntaxe, la description d'une application ΣC se fait à l'aide des mots clefs `agent` et `subgraph`.

```
1  agent worker {
2      interface {
3        in <unsigned char>  input_data;
4        out<unsigned char>  output_data;
5
6        spec{
7          {input_data[1];output_data[0]};
8          {input_data[1];output_data[0]};
9          {input_data[1];output_data[1]}
10       };
11     }
12
13     void readAndDestroy (void) exchange (
14                                  input_data input_data[1],
15                                  output_data output_data[0]){
16     }
17
18     void readAndCopy    (void) exchange (
19                                  input_data input_data[1],
20                                  output_data output_data[1]){
21        output_data[0] = input_data[0];
22     }
23
24         void start (void)
25         {
26                 readAndDestroy();
27                 readAndDestroy();
28                 readAndCopy();
29         }
30 }
```

FIGURE 2.10 – Définition d'un agent de traitement en ΣC qui ne copie qu'une donnée sur trois.

Les agents La Figure 2.10 définit un `agent`. Il se décompose en deux parties. Premièrement l'`interface` qui va décrire les entrées/sorties de l'agent et leurs spécifications (les quantités de données lues et écrites pour chaque exécution). Puis vient le corps principal qui va permettre de décrire les traitements effectués par l'agent à chacune de ses exécutions.

Contrairement au `filter` de StreamIt, un agent ΣC peut disposer d'un nombre d'entrées/sorties quelconque. De plus, en ΣC la quantité de données produite et consommée par un agent dans chacune de ses entrées/sorties peut varier selon une période prédéfinie par la spécification.

Les sous-graphes On utilise ensuite les `subgraph` pour relier librement un ensemble d'agents. Les `subgraph` généralisent donc les structures `loop` et `splitjoin` de StreamIt.

Par exemple, dans la Figure 2.11 page ci-contre, un `subgraph` définit trois instances d'agent, un `reader`, un `writer` et un `worker` (précédemment définie en Figure 2.10). Toutes ces instances sont ensuite reliées l'une à l'autre à l'aide de la fonction `connect`.

```
1  subgraph root {
2    map {
3      agent r = new Reader();
4      agent w = new Writer();
5      agent f = new Worker();
6      connect(r.out,f.input_data);
7      connect(w.in ,f.output_data);
8    }
9  }
```

FIGURE 2.11 – Définition d'un sous-graphe de trois agents en ΣC.

2.3 Modélisation dataflow

On entend par dataflow (ou « flot de données »), un réseau de tâches ne communiquant qu'au travers de canaux de communication. La nature de ces tâches peut varier d'une simple opération (on parlera de dataflows à grain fin) à un processus complet (on parlera de dataflows à gros-grain).

2.3.1 Historique

La modélisation dataflow dans le sens où on l'entend aujourd'hui a fait son apparition vers la fin des années 50. Ce type de modélisation était alors motivé par deux domaines distincts :

- les architectures dataflows (prenons l'exemple de la machine dataflow de Dennis [33]),
- les langages de programmation dataflows et/ou visuels.

Les architectures dataflows Contrairement aux architectures *Von Neumann*, qui traitent séquentiellement les instructions, une architecture dataflow interprète un programme comme un ensemble de tâches liées par des dépendances de données. Plusieurs calculs peuvent alors aisément s'effectuer en parallèle [33, 5]. Le flot d'exécution dans une architecture dataflow n'est alors plus dirigé par le contrôle (l'ordre des opérations), mais par les données.

Jusqu'aux années 80, de nombreux projets d'architectures de ce type firent leur apparition, souvent accompagnés d'un langage ou d'un modèle dataflow spécifique. Cependant, la plupart de ces architectures et modèles ne se focalisaient alors que sur des dépendances de grain fin, ces choix architecturaux furent d'ailleurs critiqués [38].

Rappelons que le MPPA n'est pas une architecture dataflow.

Les langages visuels La programmation « graphique » a souvent été l'objet implicite de recherches portant sur des modèles dataflows [92]. On cherchait alors à résoudre des problématiques telles que la représentation graphique de boucle ou de système récursif. Ce style de programmation est loin d'être parmi les plus répandus, mais aujourd'hui on compte encore quelques exceptions, surtout dans le milieu de l'industrie, comme Simulink de Mathwork [29]. Le langage ΣC n'est pas un langage visuel.

Parmi les langages dataflows d'aujourd'hui, certains s'attachent à résoudre un problème bien plus récent : une description plus simple du parallélisme. C'est notamment le cas des langages StreamIt et ΣC. En effet, là où les *threads* décrivent des dépendances de donnée implicites et un parallélisme explicite, les langages dataflows décrivent des dépendances de donnée explicites, et un parallélisme implicite.

2.3.2 Les modèles dataflows de référence

Afin de nous familiariser avec la modélisation dataflow, nous présentons quelques-uns des modèles les plus fréquemment utilisés pour la conception de systèmes embarqués. Il est impossible de présenter tous les modèles existant tant ils sont nombreux, mais le lecteur curieux est invité à voir la Figure 2.16 page 26 où d'autres modèles significatifs sont énumérés.

Les réseaux de Kahn

Les réseaux de Kahn [50] (ou KPN pour *Kahn Process Network*) ne sont pas parmi les premiers modèles dataflows proposés. Ils restent cependant parmi les modèles les plus généraux ; c'est pour cette raison que nous commençons par les présenter.

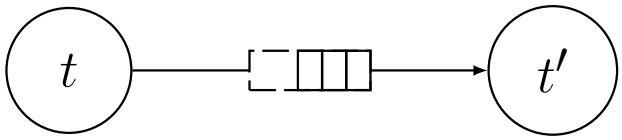


FIGURE 2.12 – Un exemple simple de réseau de Kahn à deux processus.

Un réseau de Kahn se compose d'un ensemble de processus qui communiquent exclusivement au travers de canaux. Un exemple est visible en Figure 2.12.

– Les canaux de communications sont des mémoires FIFO [4] non bornées où seul le processus d'entrée peut écrire et seul celui en sortie peut lire.

4. First In Fist Out

- La lecture est bloquante : un processus lorsqu'il engage une lecture doit attendre la présence de données avant de continuer son exécution et il ne peut pas vérifier la présence des données avant de s'exécuter.
- Les processus ne sont pas réentrants : lorsqu'un processus s'exécute, il doit finir cette exécution avant d'en commencer une nouvelle.

Un modèle déterministe Les caractéristiques d'un réseau de Kahn lui confèrent une propriété essentielle : il est déterministe, c.-à-d. les données de sortie d'un réseau de Kahn ne peuvent dépendre que des données d'entrée. L'ordre dans lequel peuvent s'exécuter les processus, la durée de leur exécution ou des délais de communication n'auront pas d'influence sur la valeur des données de sortie.

Les réseaux de Kahn sont parmi les modèles dataflows les plus fréquemment utilisés pour décrire des applications, mais ils sont trop généraux pour être appliqués à certains domaines de l'informatique embarquée. En effet, lorsqu'une application dataflow doit être exécutée sur un système embarqué, l'espace mémoire disponible est faible, et les réseaux de Kahn ne nous permettent pas de déterminer la quantité de mémoire nécessaire à son exécution [23]. Dans ce cas, soit le concepteur détermine ces tailles mémoires manuellement (en prenant le risque de faire une erreur), soit il s'oriente vers un modèle de programmation statique, un modèle moins expressif, mais pour lequel le dimensionnement et bien d'autres questions deviennent décidables.

Computation Graphs

Les *Computation Graphs* [51] (CG) furent l'un des premiers modèles dataflows statiques proposés. Tout comme les réseaux de Kahn, ils représentent une application par un graphe orienté $\mathcal{G} = (\mathcal{T}, \mathcal{A})$.

- Chaque nœud $t \in \mathcal{T}$ correspond à une tâche (ou agent).
- Chaque arc $a = (t, t')$ est un buffer FIFO non borné reliant la tâche t à la tâche t'.
- Un buffer peut contenir des données que l'on désignera par le terme de jetons. La quantité initiale de jetons dans un buffer a est alors notée $M_0(a)$, on parle aussi de marquage initial.

Les nœuds peuvent s'exécuter ; ils consomment (*resp.* produisent) alors des jetons dans leurs entrées (*resp.* sorties).

- in_a désigne alors la quantité de jetons produite dans un buffer a à chaque exécution de t (aussi appelé le taux de production),
- out_a est la quantité consommée dans le buffer a à chaque exécution de t' (le taux de consommation),

– thr_a est le seuil de données nécessaires dans le buffer a pour l'exécution de la tâche t (le seuil d'exécution). On remarque alors que $thr_a \geq out_a$ doit toujours être vérifié.

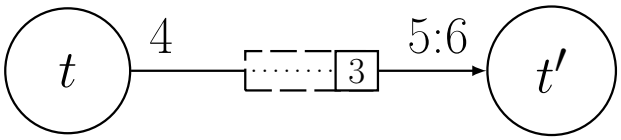


FIGURE 2.13 – Un exemple de Computation Graph

Sur la Figure 2.13, on représente un *Computation Graph* de deux tâches t et t' reliées par le buffer $a = (t, t')$. Le nombre de jetons initiaux du buffer a est $M_0(a) = 3$. À chacune de ses exécutions, la tâche t va produire $in_a = 4$ jetons dans le buffer a. De même, le seuil d'exécution de la tâche t' est $thr_a = 6$, elle devra donc attendre 6 jetons dans le buffer a avant de pouvoir s'exécuter et consommer $out_a = 5$ jetons.

Un modèle statique Comme les taux de production et de consommation de ce modèle sont fixes, il est possible d'en étudier le comportement *a priori* et sans exécuter le programme modélisé : c'est un modèle statique. Plusieurs problèmes fondamentaux deviennent alors décidables. Il est par exemple possible de vérifier si une application est en mesure de s'exécuter sans jamais se bloquer (c.-à-d. si elle est vivante), ou bien de déterminer une quantité d'espace minimale pour ses buffers de sorte de garantir son fonctionnement.

Synchronous Dataflow Graph

Les *Synchronous Dataflow Graphs* [56] (SDFG) sont un cas particulier des *Computation Graphs* où pour les buffers, le seuil d'exécution est égal au taux de lecture,

$$\forall a \in \mathcal{A}, \;\; thr_a = out_a.$$

On notera aussi l'existence d'un cas particulier des *Synchronous Dataflow Graphs*, les *Homogeneous Synchronous Dataflow Graphs* [56] (HSDFG). Il s'agit d'un SDFG où les taux de lecture et d'écriture sont égaux ($out_a = in_a$). Par conventions on notera alors

$$\forall a \in \mathcal{A}, \;\; out_a = in_a = 1.$$

Cyclo-Static Dataflow Graph

Les *Cyclo-Static Dataflow Graphs* [19] (CSDFG) sont une généralisation des SDFG. Dans ce modèle, l'indicateur de production et de consommation est plus précis ; il se décompose en phase. Les quantités out_a et in_a peuvent donc varier dans le temps en respectant un cycle prédéterminé et sont représentées sous forme d'un vecteur. Soit un

buffer $a = (t, t')$, on note alors $out_a(k')$ (*resp.* $in_a(k)$) la quantité de jetons lue (*resp.* écrite) lors de la k'^{e} exécution de la tâche t' (*resp.* k^{e} exécution de la tâche t) avec $k' \in \{1, \cdots, \varphi(t')\}$ (*resp.* $k \in \{1, \cdots, \varphi(t)\}$).

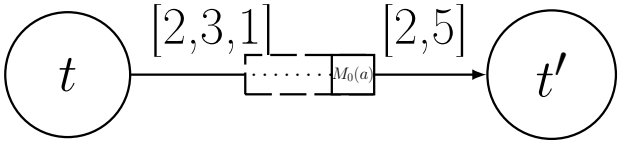


FIGURE 2.14 – Un exemple de CSDFG.

Sur la Figure 2.14, après chacune de ses exécutions, la tâche t produira donc 2 jeton, puis 3 jetons, puis 1 jetons. Cette séquence de $\varphi(t) = 3$ exécutions sera désormais appelée une itération. Une fois cette itération terminée, on reprendra alors au début.

On notera i_a (*resp.* o_a) la quantité totale de données produites (*resp.* consommées) dans le buffer a après une itération de sa tâche productrice (*resp.* consommatrice).

$$i_a = \sum_{k=1}^{\varphi(t)} in_a(k) \quad \text{et} \quad o_a = \sum_{k'=1}^{\varphi(t')} out_a(k')$$

Phased Computation Graph

Les *Phased Computation Graphs* [87] (PCG) sont une extension des CSDFG qui considère des seuils d'exécution comme pour le modèle CG ainsi que des phases d'initialisation. Ces phases s'exécuteront alors une unique fois pendant la première itération de la tâche.

En plus des notations des CSDFG, pour un buffer $a = (t, t')$, le seuil d'exécution de la k'^{e} phase de la tâche t' se note $thr_a(k')$ avec $k' \in \{1, \cdots, \varphi(t')\}$. De plus, on compte $\phi(t)$ phases d'initialisation pour une tâche t, et on indicera ces phases dans un intervalle $[1 - \phi(t), 0]$.

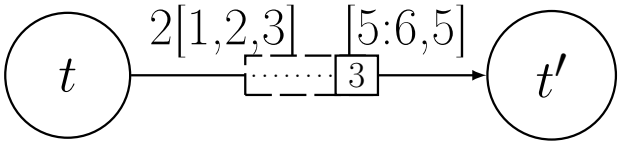


FIGURE 2.15 – Un exemple de PCG. La tâche t dispose d'une phase d'initialisation tandis que la phase t' n'en a pas. Concernant les taux de lecture, la notation des phases n:m indique en premier la quantité de données lues puis le seuil d'execution.

Dans l'exemple de la Figure 2.15, la tâche t dispose par exemple d'une phase d'initialisation ($\phi(t) = 1$) pendant laquelle $in_a(0) = 2$ jetons vont être produits tandis que la tâche t' n'a pas de phase d'initialisation ($\phi(t') = 0$). Le seuil de la tâche t'

Parameterized Dataflow et *Scenario-Aware Dataflow*

Bien que les modèles SDFG ou CSDFG s'apprêtent bien à l'analyse statique, le nombre d'applications supportées par ces modèles reste encore limité. Certaines applications doivent par exemple pouvoir traiter des données différentes à des instants différents ; on parle de mode d'exécution. Dans ce type d'application, comme le passage d'un mode à l'autre n'est plus prédictible, une analyse statique n'est plus possible.

Le modèle *Scenario-Aware DataFlow* [85] permet de vérifier statiquement chacun des modes de l'application, ainsi que les étapes de transition entre deux modes. Une analyse quasi statique est donc possible. Pour le modèle *Parameterized Dataflow*, cette analyse est plus complexe. En effet, ce modèle permet de décrire des situations de non-déterminisme [16].

Dynamic Dataflow Graph

À l'opposé des modèles « statiques », il existe des modèles « dynamiques ». C'est une classe de modèle pour lesquels certains problèmes d'analyse statique comme la détection d'interblocage ne sont plus décidables. Il n'existe pas véritablement de modèle *Dynamic Dataflow Graph*, mais l'on note l'existence de plusieurs modèles associés à cette dénomination. C'est notamment le cas des *Boolean Dataflow Graphs* (BDFG) de Buck et Lee [23], des *Process Networks* (PN) de Parks [70], ou du *U-Interpreter* (U-I) de Arvind et Gostelow [4].

Chacun de ces modèles intègre des fonctionnalités dynamiques qui rendent leurs analyses complexes, voire impossibles. L'U-I intègre des tâches qui ne sont pas soumises à la lecture bloquante tout comme les PN. Les BDFG restent déterministes, mais leur analyse statique n'est plus possible, car leurs taux de production peuvent dépendre des données d'entrées. On compte aussi une extension dynamique du modèle CSDFG, les *Cyclo-Dynamic Dataflow Graphs* [90] (CDDFG) qui, tout comme les CSDFG, indiquent pour chaque entrée/sortie les taux de production et de consommation sous forme de vecteur, mais pour lesquels l'ordre d'exécution des phases n'est plus fixe.

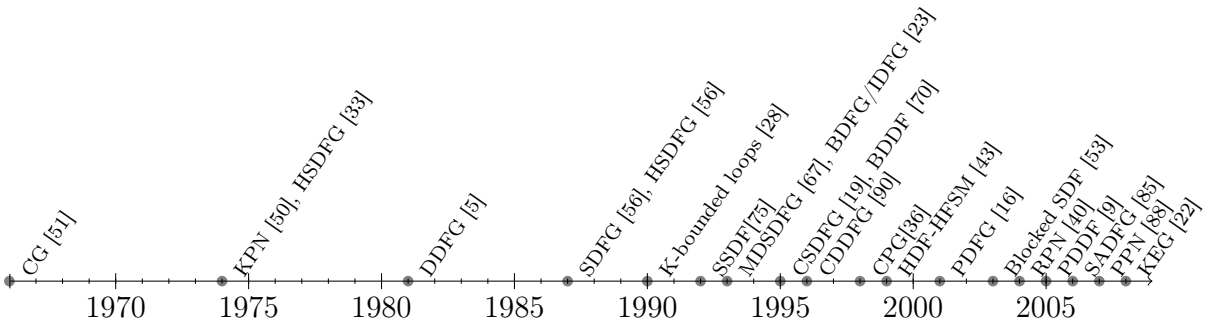


FIGURE 2.16 – Une chronologie des principaux modèles référencés dans la littérature

De même que le choix d'une d'architecture était particulièrement décisif concernant les performances et les interfaces d'un système embarqué, on s'aperçoit que le modèle de programmation utilisé pour concevoir ce système va aussi fortement influer sur ses garanties de fonctionnement et d'efficacité.

Le choix de la modélisation dataflow se justifie d'une part parce qu'elle fournit une expressivité du parallélisme bien plus simple et bien plus sûre qu'un autre modèle, mais d'autre part parce qu'elle permet (dans le cas des dataflows statiques) de vérifier le bon fonctionnement d'une application et d'en étudier les performances avant son exécution.

Dans cette thèse, nous travaillons à l'amélioration d'une chaîne de compilation permettant d'exécuter une application ΣC sur le processeur MPPA. Il s'avère qu'une application ΣC peut être efficacement transformée en un modèle *Cyclo-Static Dataflow Graph* ou *Synchronous Dataflow Graph*. Pour cette raison, nous limitons notre étude à ces modèles particuliers.

Pourtant, simplifier les efforts de modélisation en utilisant les dataflows va repousser une part du travail de conception vers la compilation. Voyons alors plus en détail ces conséquences sur une chaîne de compilation « dataflow ».

2.4 Compilation dataflow à destination d'un multiprocesseur

La compilation (en informatique) est une opération qui consiste à convertir une application décrite à l'aide d'un langage de programmation vers un code compatible avec la machine d'exécution ciblée. Lorsque la cible est une architecture multiprocesseur, des opérations de contrôle relatives à la gestion du parallélisme (comme les *threads* POSIX) peuvent alors être utilisées.

Dans le cadre des langages dataflows, comme la description de tâches concurrentes permet d'induire du parallélisme implicite, il n'est plus nécessaire d'utiliser ces opérations de contrôle.

Du point de vue le plus général qui soit, la compilation dataflow se distingue alors d'une compilation plus standard en gérant elle même cette parallélisation, soulageant ainsi le concepteur. D'un point de vue pratique, un compilateur dataflow doit transformer un programme dataflow en un programme utilisant des API de parallélisation comme les *threads* POSIX ou MPI.

2.4.1 L'exécutif : charnière d'une chaîne de compilation

Une pratique courante en compilation consiste à confiner le support d'exécution parallèle dans un exécutif (ou *runtime*).

L'exécutif est un programme « maître » gérant l'ordre d'exécution des différentes tâches, les emplacements mémoires qu'elles utiliseront pour communiquer et les unités de calcul qui leur seront affectées. Le compilateur dataflow doit alors définir cet exécutif et lui fournir toutes les informations nécessaires pour qu'il fonctionne et exécute un dataflow. La spécification de l'exécutif est déterminante ; elle détermine les tâches à réaliser par le compilateur et les contraintes d'expressivité du langage utilisé.

Notons qu'il existe une taxinomie [55] permettant de caractériser un *runtime* selon l'expressivité des informations qu'il convient de lui fournir. Voici quelques une des classes définies :

- *Fully dynamic* : le *runtime* ne requiert qu'un graphe dataflow de l'application pour fonctionner. Il assurera l'ensemble des opérations, de l'attribution des ressources (processeurs, mémoire...) à l'exécution des tâches.
- *Static assignment* : ce mode est très similaire au *fully dynamic*. Seules les ressources seront fixées à l'avance pour chaque processus (processeurs d'exécution, mémoires,...).
- *Self-timed* : ici un mécanisme de synchronisation est mis en œuvre et permet de garantir un ordre d'exécution partiel des tâches (qu'il sera nécessaire de communiquer au *runtime*).
- *Fully static* : plus aucune variation de fonctionnement n'est envisagée, tout est décidé à l'avance. Le *runtime* nécessite une description exacte de l'ordre d'exécution des tâches et des ressources qu'elles utiliseront.

Une fois le type d'exécutif connu, le rôle du compilateur est double :

- d'une part, il doit valider une application dataflow et vérifier qu'elle respecte ses contraintes de fonctionnement,
- d'autre part il devra générer les informations requises par l'exécutif.

Pour répondre aux attentes de la compilation dataflow, il est donc nécessaire d'analyser le programme concerné et d'en extraire ces informations. En référence à l'analyse statique de programme, nous appellerons cette étude l'analyse statique des dataflows.

2.4.2 Des exemples de chaînes de compilation dataflow

Il existe d'ores et déjà plusieurs outils académiques permettant de compiler des applications dataflows à destination d'une architecture multiprocesseur.

Deadalus et DeadalusRT Les outils Deadalus et DeadalusRT [7] sont deux chaînes de compilation développées conjointement par les universités de Leiden et d'Amsterdam en Hollande. Ces chaînes de compilation sont adaptées aux systèmes embarqués (l'une est spécialisée pour le temps réel dur). L'une des contributions majeures de ces travaux porte sur un outil d'extraction de CSDFG directement à partir d'un programme C (sous certaines contraintes).

Ptolemy Le groupe Ptolemy de l'université de Berkeley est très certainement l'un des précurseurs dans le domaine de l'aide à la conception des systèmes embarqués. Il est entre autres à l'origine de trois outils universitaires, Gabriel [57] et Ptolemy I et II [17] qui couvrent chacun un vaste domaine d'applications allant de la modélisation à la simulation.

D'autre part ces outils disposent d'un langage dataflow : le CAL, repris par d'autres projets de compilation parallèles comme OpenDF [18] ou PREESM [71]. La limite de ces outils porte essentiellement sur la forme et la taille des applications traitées. Ces outils se basent souvent sur des techniques exactes pour lesquelles des applications de taille industrielle ne peuvent être prises en charge.

SDF3 Au sein de l'université technologique d'Eindhoven aux Pays-Bas, le groupe *Electronic Systems* propose SDF3. Il ne s'agit pas d'un compilateur à part entière, mais d'une API fournissant les outils d'analyse nécessaire à la plupart des compilateurs dataflows. Une solution de génération aléatoire d'application dataflow existe aussi. Il permet de tester leurs outils.

StreamIt Le langage StreamIt (vu précédemment) fait partie d'une chaîne de compilation du même nom [86]. La chaîne de compilation StreamIt a été développée par une équipe du groupe CSAIL au Massachusetts Institute of Technology. Elle est à l'origine destinée à la compilation de programme StreamIt vers des processeurs à architectures parallèles comme le RAW [83] ou le Tile64 [91]. Compte tenu de la sémantique de son langage, la principale limitation de StreamIt est son expressivité : une application StreamIt ne peut par exemple disposer que d'une unique d'entrée et d'une unique sortie. Il existe bien des extensions de ce langage, mais aucune n'est intégrée officiellement dans la chaîne de compilation *ad hoc*.

Ces quelques exemples ne nous permettent pas de conclure sur l'ensemble des solutions de compilation dataflow existantes.

Nous souhaiterions cependant faire réagir le lecteur sur un point essentiel. Selon nous, même si chacun de ces outils apporte chaque fois de nouvelles contributions majeures, aucun n'est suffisamment abouti pour fournir une solution de programmation parallèle « clef en main » à destination d'une nouvelle architecture multiprocesseur comme le MPPA.

Afin de faciliter l'accès à son architecture, la société Kalray a donc fait le choix de fournir sa propre solution de compilation dataflow : la chaîne AccessCore.

2.4.3 La chaîne de compilation AccessCore

Le langage ΣC est un support permettant d'exprimer une application sous la forme d'un modèle CSDFG. Le concepteur peut ainsi spécifier le parallélisme de son application sous forme d'un modèle dataflow. Il n'est pas nécessaire à cet instant qu'il décide du séquencement de ses opérations ou des ressources qui leur seront associées. C'est ici qu'intervient la chaîne de compilation AccessCore.

Le passage d'un programme ΣC vers un exécutable compatible avec le MPPA n'est cependant pas évident. Le *runtime* utilisé par cette chaîne de compilation s'apparente à la catégorie des *self-timed runtimes*. Rappelons que pour ce type de *runtime*, il est nécessaire de

- fournir un ordre partiel d'exécution des tâches,
- placer les tâches,
- dimensionner les mémoires.

Pour fournir ces informations, une mécanique complexe a donc été développée par Kalray (en partenariat avec le CEA) et s'articule en cinq étapes :

Instanciation du CSDFG La première étape consiste à générer un CSDFG caractérisant l'application ΣC considérée. C'est ce graphe qui permettra par la suite d'effectuer toutes les analyses nécessaires à la compilation.

Réduction du parallélisme Le MPPA dispose de 256 cœurs de calcul, et la modélisation de certaines applications peut souvent s'exprimer au travers de bien plus de tâches. Afin de simplifier l'analyse du graphe, une opération de réduction est engagée. Celle-ci va diminuer le nombre de tâches en fusionnant certaines d'entre elles. Il s'agit d'un problème NP-Complet [32], mais pour lequel une solution approchée est utilisée [27].

Dimensionnement mémoire Cette étape décide des ressources mémoires minimales nécessaires à l'exécution d'une application. Ces informations sont requises par l'exécutif. Ce problème est NP-Complet [62]. Il est résolu par AccessCore à l'aide d'une heuristique gloutonne s'appuyant sur une exécution symbolique du CSDFG [80].

Partitionnement Chaque cluster dispose d'un espace mémoire et un nombre d'unités de calcul limités. Cette étape doit partitionner le CSDFG en autant de groupes qu'il

existe de clusters disponibles (16 pour le MPPA), et cela en tenant compte du coût de communication entre les clusters. Ce problème est NP-Complet : la solution utilisée par la chaîne de compilation est une métaheuristique à base de recuit simulé [79].

Placement et routage Les partitions calculées précédemment sont ensuite assignées aux clusters, et des voies de communication sont tracées afin de garantir un débit minimal. Ce problème est NP-Complet. La solution utilisée par la chaîne de compilation est une métaheuristique à base de recuit simulé pour le placement [79] et de programmation linéaire en nombre entier pour le routage [78].

Ainsi, en s'aidant d'un modèle de programmation restrictif (le modèle CSDFG), et après toutes ces étapes d'analyse statique (résumées sur la Figure 2.17), la chaîne de compilation AccessCore est en mesure de générer un code parallèle et exécutable sur le processeur MPPA sans qu'à aucun moment l'utilisateur n'ait eu à se préoccuper des détails techniques les plus complexes comme le dimensionnement des mémoires, ou le routage des données.

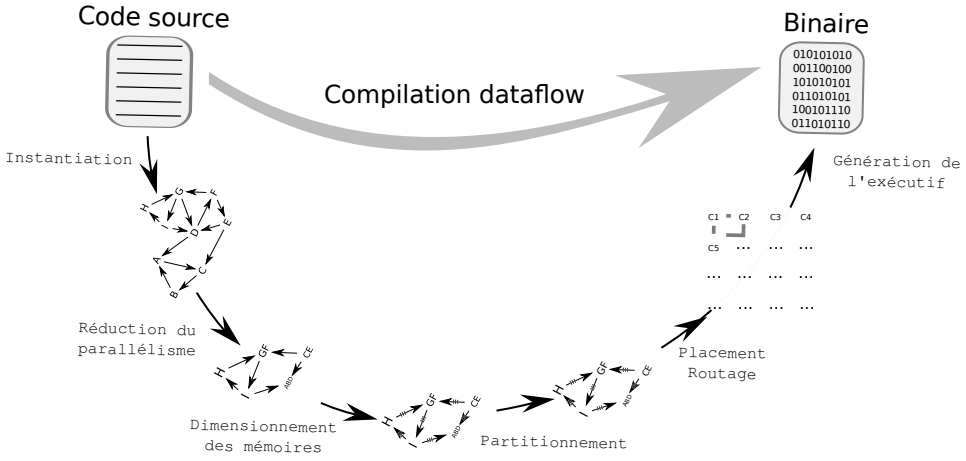


FIGURE 2.17 – Flot de conception mis en œuvre par AccessCore.

Conclusion

Dans ce chapitre, nous exposons le contexte de cette thèse. Nous étudions AccessCore, une chaîne de compilation spécialement développée par Kalray pour le processeur MPPA. AccessCore s'appuie sur le langage dataflow ΣC et ce langage peut être traduit dans un modèle dataflow statique, le *Cyclo-Static Dataflow Graph*.

Au cours de nos travaux, nous avons étudié les différentes étapes de la chaîne de compilation AccessCore, et nous proposons des améliorations pour certaines d'entre elles.

Dans le chapitre suivant, nous présentons plus en détail un état de l'art sur l'analyse des modèles dataflows, et nous détaillons les différents problèmes qui ont été étudiés dans cette thèse.

Chapitre 3

État de l'art et Problématiques

Introduction

La modélisation d'une application sous forme d'un dataflow plutôt qu'un programme séquentiel peut être considérée comme un travail supplémentaire pour un concepteur. Néanmoins, nous connaissons désormais les multiples avantages de ce modèle, comme l'expression d'un parallélisme implicite ou la disparition de dépendances de donnée inutiles.

Nous savons aussi que les modèles dataflows statiques comme les SDFG ou les CSDFG fournissent des informations supplémentaires sur l'exécution d'une application. Il est alors possible d'évaluer avec une grande précision les performances d'un programme, ou de déterminer l'espace mémoire nécessaire à son exécution.

Dans un contexte embarqué, il s'avère que ces questions sont d'un intérêt capital ; c'est d'ailleurs pour cette raison que la modélisation dataflow statique était utilisée pour la programmation des systèmes embarqués bien avant l'émergence du parallélisme.

Nos contributions portent principalement sur trois problèmes d'analyse statique dataflows parmi les plus fondamentaux :

1. **la vivacité**, est-ce qu'une application dataflow est en mesure de s'exécuter sans aucun blocage ?
2. **l'évaluation du débit maximal**, qu'elle est la fréquence de fonctionnement maximale des tâches d'une application dataflow ?
3. **le dimensionnement mémoire**, quelle quantité mémoire est nécessaire pour obtenir les performances attendues d'une application dataflow ?

En tout premier lieu, nous présentons plusieurs résultats fondamentaux de l'analyse statique des SDFG et des CSDFG. Nous détaillons ensuite les trois problèmes traités ici ainsi que les différentes méthodes existantes pour les résoudre.

3.1 Ordonnancement des modèles dataflows statiques

Étant donné un ensemble de tâches, résoudre un problème d'ordonnancement consiste à fixer une date de réalisation pour chacune de ces tâches tout en respectant des contraintes associées.

Il s'avère que les modèles dataflows statiques fournissent suffisamment d'informations pour permettre d'anticiper leur comportement. Il est donc possible de déterminer un ordonnancement pour chacune de leurs tâches. Les contraintes à respecter sont alors les dépendances de données qui peuvent exister entre les exécutions de ces tâches.

3.1.1 Définition d'un ordonnancement

Si l'on considère un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, pour chacune de ses tâches $t \in \mathcal{T}$, on note $\langle t, n\rangle$ la n^{e} exécution de la tâche t, et $\mathcal{S}\langle t, n\rangle$ sa date d'exécution pour un ordonnancement $\mathcal{S}$.

L'ordonnancement $\mathcal{S}$ d'une application est alors valide si l'ensemble des dates d'exécution fixées respecte les contraintes définies par le modèle.

La fréquence de fonctionnement d'une tâche t pour un ordonnancement $\mathcal{S}$ est alors notée $Th_t^{\mathcal{S}}$.

Définition 1 *Dans un ordonnancement $\mathcal{S}$, la fréquence de fonctionnement d'une tâche $t \in \mathcal{T}$ est définie par*

$$Th_t^{\mathcal{S}} = \lim_{n\to\infty} \frac{n}{\mathcal{S}\langle t, n\rangle}.$$

3.1.2 Relation de précédence

La principale contrainte d'exécution dans un modèle dataflow porte sur la quantité de jetons requise à l'exécution d'une tâche. Pour l'ensemble des modèles statiques que nous avons cité précédemment, on parle de seuil d'exécution.

Ainsi, pour tout buffer $a = (t, t') \in \mathcal{A}$, les exécutions de t' peuvent requérir la présence de données produites par t. On dit alors que le buffer a induit une relation de précédence entre les exécutions des tâches t et t'.

Il est communément admis qu'il existe une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si :

- **Condition 1** : $\langle t', n'\rangle$ peut être exécuté après $\langle t, n\rangle$.
- **Condition 2** : $\langle t', n'\rangle$ ne peut s'exécuter avant $\langle t, n\rangle$.

Munier-Kordon [66] définit alors une relation de précédence *stricte* de la manière suivante :

Définition 2 *Considérons le SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Soit $a = (t, t') \in \mathcal{A}$ un buffer et (n, n') un couple d'entiers strictement positifs. Il existe une relation de précédence stricte entre les exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si :*

- ***Condition 1*** *: $\langle t', n'\rangle$ peut être exécuté après $\langle t, n\rangle$.*
- ***Condition 2*** *: $\langle t', n'\rangle$ ne peut s'exécuter avant $\langle t, n\rangle$.*
- ***Condition 3*** *: $\langle t', n' - 1\rangle$ peut s'exécuter avant $\langle t, n\rangle$.*

Dans la suite de ce manuscrit, pour simplifier sa lecture et comme nous ne traitons que de relation de précédence stricte, nous utiliserons toujours le terme « relation de précédence » pour parler de « relation de précédence stricte ».

Ainsi, si l'on prend l'exemple du SDFG de la Figure 3.1 page ci-contre, il existe une relation de précédence entre les exécutions $\langle B, 5\rangle$ et $\langle C, 6\rangle$ induite par le buffer a_{BC}. En effet, après l'exécution de $\langle B, 5\rangle$, le buffer a_{BC} contient $5 \times 8 = 40$ jetons, et après l'exécution $\langle C, 6\rangle$, $6 \times 6 = 36$ jetons ont été consommés, $\langle C, 6\rangle$ peut donc s'exécuter après $\langle B, 5\rangle$ (Condition 1). Après l'exécution $\langle B, 4\rangle$, le buffer a_{BC} contient $4 \times 8 = 32$ jetons, et pour réaliser l'exécution $\langle C, 5\rangle$, $5 \times 6 = 30$ jetons auront été nécessaires ; $\langle C, 5\rangle$ peut donc s'exécuter avant $\langle B, 5\rangle$ (Condition 3). Enfin on remarque qu'il n'y a pas suffisamment de jetons pour permettre l'exécution de $\langle C, 6\rangle$ avant $\langle B, 5\rangle$ (Condition 2).

Pour respecter une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$, un ordonnancement $\mathcal{S}$ doit alors vérifier

$$\mathcal{S}\langle t', n'\rangle \geq \mathcal{S}\langle t, n\rangle + d(t).$$

3.1.3 Ordonnancement *au plus tôt*

L'ordonnancement *au plus tôt* est très certainement la politique d'ordonnancement la plus couramment rencontrée dans les algorithmes d'analyse dataflows. La règle d'exécution *au plus tôt* impose à toutes les tâches d'un graphe de s'exécuter dès qu'elles disposent de suffisamment de données en entrée. Pour cette raison, un ordonnancement *au plus tôt* sans limite de ressource atteint nécessairement la fréquence de fonctionnement maximale de l'application [25].

L'ordonnancement *au plus tôt* d'un dataflow statique comme les SDFG ou les CSDFG se décompose toujours en deux parties.

- La première partie (le régime transitoire) compte un nombre d'exécutions borné (possiblement nulle).
- Elle est suivie par le régime permanent : un motif d'exécution des tâches se répétant à l'infini.

Bien que le nombre d'exécutions nécessaires pour décrire le régime transitoire et le régime permanent soit fini, on ne dispose que de très peu de garanties sur sa valeur exacte. Cependant, il est communément admis que ce nombre d'exécutions est de taille exponentielle.

La Figure 3.1 page suivante montre l'ordonnancement *au plus tôt* d'un SDFG. On retrouve la régime transitoire (dont les exécutions sont marquées par des 0) suivie du régime permanent.

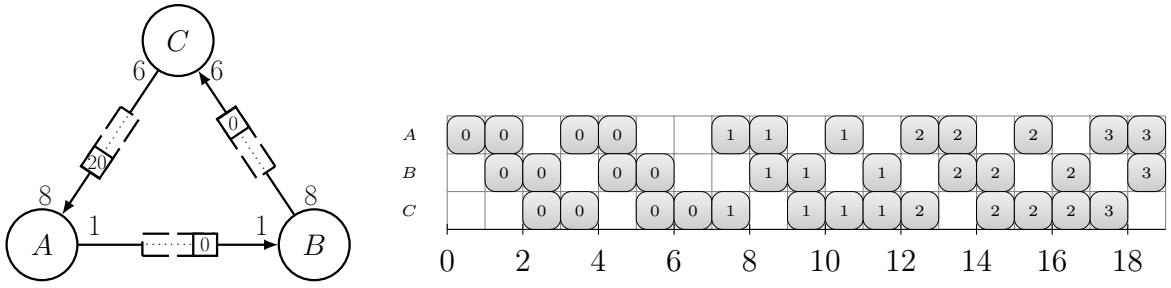


FIGURE 3.1 – Un exemple de SDFG accompagné de son ordonnancement *au plus tôt*.

3.1.4 Ordonnancement périodique

Afin de s'affranchir des ordonnancements *au plus tôt*, et de leur complexité, Benabid et al. [10] s'intéressent aux ordonnancements périodiques. Un ordonnancement périodique caractérise alors l'ensemble des exécutions d'une tâche par une première date d'exécution $\mathcal{S}\langle t, 1\rangle$ accompagnée d'une période de fonctionnement $\mu_t^{\mathcal{S}} \in \mathbb{R}^+ - \{0\}$.

Définition 3 *Benabid et al. [10] Considérons le SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. un ordonnancement $\mathcal{S}$ est* périodique *s'il existe pour chaque tâche $t \in \mathcal{T}$ une période de fonctionnement $\mu_t^{\mathcal{S}} \in \mathbb{R}^+ - \{0\}$ telle que*

$$\forall n > 1, \;\; \mathcal{S}\langle t, n\rangle = \mathcal{S}\langle t, 1\rangle + (n-1)\mu_t^{\mathcal{S}}.$$

Un ordonnancement périodique est soumis à des contraintes restrictives, mais sa définition en est simplifiée. En effet, contrairement à un ordonnancement *au plus tôt*, la quantité d'information à fournir est de taille polynomiale.

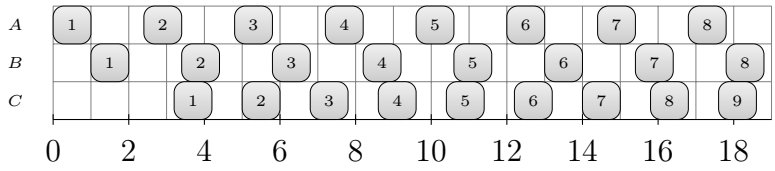


FIGURE 3.2 – Un ordonnancement périodique valide pour le SDFG de la Figure 3.1. Les exécutions marquées d'un 1 correspondent aux dates de départ initiales, les suivantes sont calculées à l'aide de la période $\mu_t^{\mathcal{S}}$.

L'exemple de la Figure 3.2 nous montre l'ordonnancement périodique d'un SDFG. Prenons l'exemple de la tâche A, sa première date de départ est $\mathcal{S}\langle A, 1\rangle = 0$, sa période de fonctionnement $\mu_A^{\mathcal{S}} = \frac{12}{5}$, ainsi chacune de ses prochaines exécutions seront espacées de $\mu_A^{\mathcal{S}}$.

On remarque enfin que la fréquence de fonctionnement d'une tâche dans un ordonnancement périodique est bien l'inverse de sa période.

$$\begin{aligned} Th_t^{\mathcal{S}} &= \lim_{n\to\infty} \frac{n}{\mathcal{S}\langle t, n\rangle} = \lim_{n\to\infty} \frac{n}{\mathcal{S}\langle t, 1\rangle + (n-1)\mu_t^{\mathcal{S}}} \\ Th_t^{\mathcal{S}} &= \frac{1}{\mu_t^{\mathcal{S}}} \end{aligned}$$

3.2 Propriétés et transformations des dataflows

Afin d'étudier les modèles dataflows, il existe différents tests et différentes transformations dont la complexité peut varier. Nous présentons ici les transformations les plus communes (comme la notion de jeton utile ou l'expansion), mais aussi certaines transformations essentielles pour nos travaux comme la normalisation.

3.2.1 Consistance

La consistance d'un SDFG est une condition nécessaire d'existence d'un ordonnancement valide avec des mémoires bornées [56].

Considérons la matrice Γ associée à une SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ et définie de la manière suivante :

$$\Gamma_{a,t} = \begin{cases} in_a & si\ a = (t, t'),\ t' \in \mathcal{T} \\ -out_a & si\ a = (t', t),\ t' \in \mathcal{T} \\ 0 & Sinon. \end{cases}$$

On dit alors que le SDFG $\mathcal{G}$ est consistant si le rang de la matrice Γ vaut $|\mathcal{T}| - 1$.

Prenons l'exemple du SDFG de la Figure 3.1 page précédente, la matrice Γ correspondante est

$$\Gamma = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 8 & -6 \\ -8 & 0 & 6 \end{pmatrix}$$

Il est alors possible de vérifier que le rang de cette matrice est égal à $|\mathcal{T}| - 1 = 2$, et donc que le SDFG est consistant.

La propriété de consistance s'adapte aussi parfaitement aux CSDFG [19]. Pour cela, on doit considérer la matrice Γ associée à un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ de la manière suivante :

$$\Gamma_{a,t} = \begin{cases} i_a & si\ a = (t, t'),\ t' \in \mathcal{T} \\ -o_a & si\ a = (t', t),\ t' \in \mathcal{T} \\ 0 & Sinon. \end{cases}$$

Rappelons que pour un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, soit un arc $a = (t, t') \in \mathcal{A}$; on note $i_a = \sum_{k=1}^{\varphi(t)} in_a(k)$ et $o_a = \sum_{k=1}^{\varphi(t')} out_a(k)$.

3.2.2 Vecteur de répétition

Une autre méthode pour vérifier la consistance d'un SDFG ou d'un CSDFG est de calculer leur vecteur de répétition [56]. Un vecteur de répétition $N^{\mathcal{G}}$ définit le nombre minimal d'exécutions nécessaire pour qu'un graphe $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ revienne à son état de marquage initial. Pour chacune des tâches $t \in \mathcal{T}$, on note alors $N_t^{\mathcal{G}}$ son facteur de répétition. Si ce vecteur de répétition existe, le graphe est consistant.

La condition d'existence d'un vecteur de répétition $N^{\mathcal{G}}$ pour un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ se note

$$\forall a = (t, t') \in \mathcal{A} \quad N_t^{\mathcal{G}} \times in_a = N_{t'}^{\mathcal{G}} \times out_a.$$

Si nous reprenons l'exemple du SDFG de la Figure 3.1 page 37, on doit donc résoudre le système linéaire suivant :

$$\begin{aligned} N_A^{\mathcal{G}} \times 1 &= N_B^{\mathcal{G}} \times 1 \\ N_B^{\mathcal{G}} \times 8 &= N_C^{\mathcal{G}} \times 6 \\ N_C^{\mathcal{G}} \times 6 &= N_A^{\mathcal{G}} \times 8 \end{aligned}$$

Une solution minimale est $N^{\mathcal{G}} = [3, 3, 4]$.

De la même manière que la consistance, cette condition est étendue aux CSDFG [19] de la manière suivante :

$$\forall a = (t, t') \in \mathcal{T} \quad N_t^{\mathcal{G}} \times i_a = N_{t'}^{\mathcal{G}} \times o_a.$$

3.2.3 Notion de jeton utile

Marchetti et Munier-Kordon [63] proposent une simplification du marquage initial d'un SDFG sans impact sur l'ordonnançabilité ou les relations de précédence induites par ses buffers. Cette propriété est essentielle à la validation de nombreux résultats théoriques sur les ordonnancements. Soit un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, pour tout arc $a \in \mathcal{A}$, on note

$$\gcd_a = \gcd(in_a, out_a),$$

où $\gcd(a, b)$ est le plus grand commun diviseur de a et b.

Pour tout entier $\beta \in \mathbb{Z}$ et $\gamma \in \mathbb{N} - \{0\}$, on note aussi

$$\lfloor \beta \rfloor^{\gamma} = \left\lfloor \frac{\beta}{\gamma} \right\rfloor \cdot \gamma.$$

Lemme 1 *Marchetti et Munier-Kordon [63] Soit un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Le marquage initial $M_0(a)$ d'un arc $a = (t, t') \in \mathcal{A}$ peut être remplacé par $\lfloor M_0(a) \rfloor^{gcd_a}$ sans aucune influence sur les dépendances de données induites par l'arc a entre les différentes exécutions des tâches t et t'.*

Ce résultat a été étendu aux CSDFG [82, 11] :

On note pour tout arc $a = (t, t') \in \mathcal{A}$,

$$\text{step}_a = \gcd(in_a(1), \cdots, in_a(\varphi(t)), out_a(1), \cdots, out_a(\varphi(t'))),$$

où $\gcd(a, b, \cdots)$ est le plus grand commun diviseur des entiers fournis.

Lemme 2 *Benazouz [11] Soit un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Le marquage initial $M_0(a)$ d'un arc $a = (t, t') \in \mathcal{A}$ peut être remplacé par $\lfloor M_0(a) \rfloor^{step_a}$ sans aucune influence sur les dépendances de données induites par l'arc a entre les différentes exécutions des tâches t et t'.*

3.2.4 Normalisation

La *normalisation* d'un SDFG est une transformation proposée par Marchetti et Munier-Kordon [63]. Elle n'a aucune influence sur les relations de précédence induites par ses buffers, mais permet de simplifier certaines étapes d'analyse. Elle fut étendue aux CSDFG par Benazouz [11].

Soit un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Pour toute tâche $t \in \mathcal{T}$, notons

- $\mathcal{A}^-(t) = \{a = (t, t') \in \mathcal{A}, t' \in \mathcal{T}\}$ l'ensemble des arcs sortant de t
- et $\mathcal{A}^+(t) = \{a = (t', t) \in \mathcal{A}, t' \in \mathcal{T}\}$ l'ensemble de ses arcs entrant.

Une tâche t est normalisée s'il existe $Z_t \in \mathbb{N} - \{0\}$ tel que

- $\forall a \in \mathcal{A}^-(t)$, $in_a = Z_t$,
- et $\forall a \in \mathcal{A}^+(t)$, $out_a = Z_t$.

Un SDFG est *normalisé* si toutes ses tâches sont normalisées.

La normalisation d'un SDFG consiste alors à construire un SDFG équivalent pour lequel toutes ses tâches sont normalisées. L'idée est de trouver deux vecteurs d'entiers $Z = (Z_1, \cdots, Z_{|T|})$ et $\Delta = (\delta_1, \cdots, \delta_{|A|})$ contenant des valeurs strictement positives telles que

$$\forall t \in \mathcal{T}, \forall a \in \mathcal{A}^-(t), \quad \delta_a \times in_a = Z_t$$

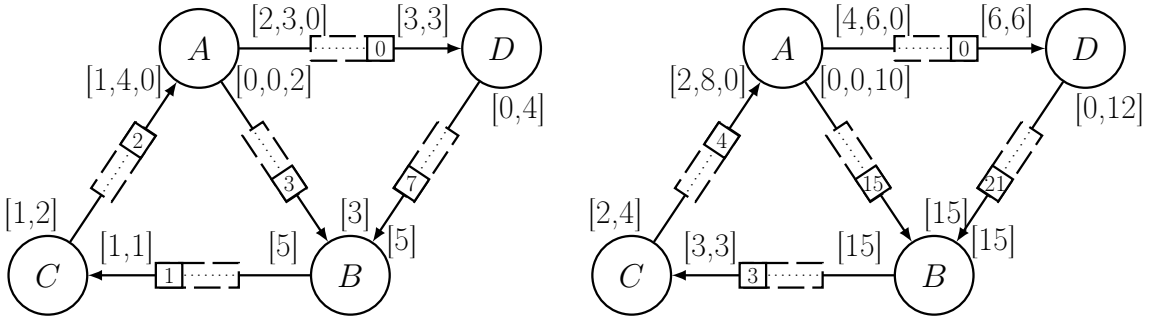
$$\forall t \in \mathcal{T}, \forall a \in \mathcal{A}^+(t), \quad \delta_a \times out_a = Z_t.$$

Marchetti et Munier-Kordon [63] prouvent que tout SDFG consistant peut être normalisé (c.-à-d. le système d'équations ci-dessus admet une solution). Les SDFG normalisés disposent aussi d'une propriété comportementale très importante : le nombre de jetons dans chacun de leurs circuits reste constant.

Pour un CSDFG, la condition de normalisation s'écrit

$$\forall t \in \mathcal{T}, \forall a \in \mathcal{A}^-(t), \delta_a \times i_a = Z_t$$

$$\forall t \in \mathcal{T}, \forall a \in \mathcal{A}^+(t), \delta_a \times o_a = Z_t.$$



(a) CSDF non normalisé

(b) CSDF normalisé

FIGURE 3.3 – (a) Un CSDFG de quatre tâches et cinq buffers. (b) Le résultat de sa normalisation.

Considérons maintenant le CSDFG de la Figure 3.3(a). Le système correspondant est alors :

$$Z_A = \delta_{CA} \times 5 = \delta_{AD} \times 5 = \delta_{AB} \times 2 \qquad Z_C = \delta_{BC} \times 2 = \delta_{CA} \times 3$$
$$Z_B = \delta_{AB} \times 3 = \delta_{DB} \times 5 = \delta_{BC} \times 5 \qquad Z_D = \delta_{AD} \times 6 = \delta_{DB} \times 4$$

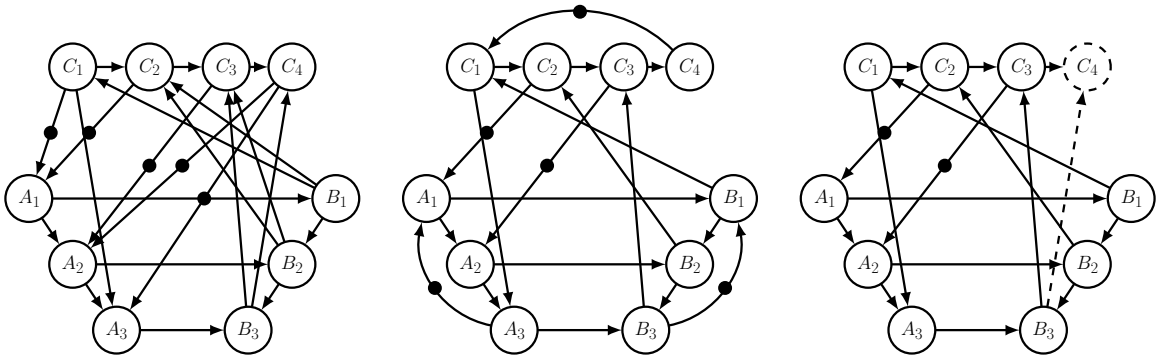
Une solution minimum est $\Delta^\star = (5, 3, 2, 2, 3)$ et $Z^\star = (10, 15, 6, 12)$.

Notons aussi que pour que cette transformation soit valide, le marquage doit aussi être soumis aux facteurs Δ ; le nouveau marquage initial est donc $M_0(a_{AB}) = 15$, $M_0(a_{BC}) = 3$, $M_0(a_{CA}) = 4$, $M_0(a_{AD}) = 0$ et $M_0(a_{DB}) = 21$, Le CSDFG normalisé est alors visible sur la Figure 3.3(b).

La normalisation est un outil d'analyse qui a déjà été utilisé avec succès sur l'étude de différentes conditions suffisantes de vivacité pour les SDFG [63] et pour les CSDFG [11, 15]. Elles simplifient l'écriture de ces conditions et certaines preuves furent grandement écourtées.

3.2.5 Expansion

L'expansion est très certainement la transformation d'application dataflow la plus utilisée par les techniques d'analyse actuelles. Étant donné un SDFG ou un CSDFG, cette transformation consiste à obtenir un *Homegeneous SDFG* (un cas particulier de SDFG où les taux de productions et de consommations sont tous égaux) dont n'importe quel ordonnancement valide est un ordonnancement valide pour le graphe dataflow d'origine. Une grande partie des techniques d'analyse portant sur les HSDFG peuvent alors directement s'appliquer à cette expansion et simplifier l'analyse du graphe.



(a) Sih et Lee [77] (b) Munier-Kordon [66] (c) de Groote et al. [31]

FIGURE 3.4 – Expansions du SDFG de la Figure 3.1 page 37 avec différentes techniques.

Lee et Messerschmitt [56] prouvent que lorsqu'un SDFG est consistant, il existe une séquence d'exécution de ses tâches permettant de revenir à son état initial, et où chaque tâche t est exécutée $N_t^{\mathcal{G}}$ fois. Partant de ce résultat, Sih et Lee [77] définissent l'expansion : une transformation d'un SDFG vers un HSDFG où, pour chaque tâche du SDFG, on compte $N_t^{\mathcal{G}}$ tâches dans le HSDFG équivalent, chacune d'entre elles correspondant à une des premières exécutions de la tâche d'origine. Afin de garantir les relations de précédence entre ces exécutions, des arcs doivent alors être ajoutés au HSDFG.

Dans la première définition de l'expansion [77], le nombre d'arcs nécessaires pour garantir ces relations de précédence dépend de la production des tâches. Pour chaque jeton produit dans buffer par l'exécution d'une tâche, un arc est ajouté. La Figure 3.4(a) montre le résultat de cette expansion pour le SDFG de la Figure 3.1 page 37. Pour cette transformation, on compte 10 tâches et 51 buffers, les arcs multiples n'apparaissent qu'une fois.

Munier-Kordon [66] prouve ensuite que le nombre d'arcs requis par cette transformation peut être diminué ; une nouvelle expansion est proposée telle que ce nombre d'arcs est minimal. Pour tout arc $a = (t, t') \in \mathcal{A}$ du SDFG d'origine, le nombre d'arcs nécessaire pour vérifier les relations de précédence qu'il induit est $\min(N_t^{\mathcal{G}}, N_{t'}^{\mathcal{G}})$. Un exemple de cette transformation est visible sur la Figure 3.4(b). Elle ne compte plus que 19 arcs.

Enfin, plus récemment, de Groote et al. [31] propose une nouvelle expansion. Dans cette méthode, pour tout arc $a = (t, t') \in \mathcal{A}$ du SDFG, l'expansion compte $N_{t'}^{\mathcal{G}}$ arcs différents. Le nombre d'arcs générés est alors plus élevé qu'avec la transformation de Munier-Kordon [66], mais cette transformation dispose d'une propriété structurelle permettant par la suite de réduire son nombre d'arc et de tâches. Un exemple de cette transformation est disponible sur la Figure 3.4(c) page ci-contre. Les tâches supprimables y sont marquées en pointillés.

3.3 Problématiques

S'il existe autant de résultats sur les propriétés structurelles et comportementales des modèles SDFG et CSDFG, c'est avant tout pour permettre de résoudre les problèmes d'analyse rencontrés sur ces modèles. Pour mener à bien la compilation d'une application ΣC modélisée par un CSDFG, nous devons résoudre différents problèmes d'analyse. Nous traitons ici trois de ces problèmes : le test de vivacité de l'application, l'évaluation de son débit et le dimensionnement de ses mémoires.

3.3.1 Test de vivacité

La *vivacité* d'une application dataflow est l'assurance que celle-ci peut fonctionner infiniment souvent. Dans le cas contraire, une application non vivante atteint un état bloquant ; on parle d'interblocage (*deadlock* en anglais).

Il s'agit là d'une des garanties de fonctionnement les plus essentielles de la conception des systèmes embarqués ; or vérifier la vivacité d'un dataflow *statique* est un problème décidable.

L'un des premiers résultats de vivacité porte sur une classe de réseau de Petri équivalente aux HSDFG. Commoner et al. [26] propose la condition nécessaire et suffisante de vivacité suivante :

Théorème 1 *Commoner et al. [26] Un HSDFG est vivant si et seulement si tous ses circuits contiennent au moins un jeton.*

Pour vérifier la vivacité d'un HSDFG, une technique simple consiste alors à supprimer tous les arcs disposant d'au moins un jeton, puis de vérifier l'existence d'un circuit (sans jeton) ; un algorithme polynomial existe, il s'appuie sur un parcours en profondeur du graphe.

Concernant les *Computation Graphs*, Karp et Miller [51] introduisent la notion de *self-terminated loop* et montrent qu'il existe un certificat polynomial vérifiant l'existence d'un interblocage.

Théorème 2 *Karp et Miller [51] Dans un* Computation Graph $\mathcal{G} = (\mathcal{T}, \mathcal{A})$*, un circuit* $c = \{a^0, t^1, a^1, t^2, \cdots, a^n\}$ *avec* $a^0 = a^n$ *est une* self-terminated loop *si et seulement s'il vérifie*

$$\forall k \in \{1, \cdots, n\} \;\; \exists x^k \in \mathbb{N} \;\; out_{a^k} \times x^k \geq M_0(a^k) - thr_{a^k} + 1 + x^k \times in_{a^k}.$$

Un CG où il existe une self-terminated loop *n'est pas vivant.*

Ce résultat prouve l'appartenance à NP du problème d'existence d'un interblocage dans un CG (et s'applique aussi aux SDFG et aux CSDFG). Cependant, ce résultat ne garantit pas l'appartenance du problème de vivacité à NP. Notons bien qu'aujourd'hui, aucun nouveau résultat de complexité n'existe à notre connaissance.

Deux techniques sont utilisées pour vérifier la vivacité d'un SDFG ou d'un CSDFG. La première méthode utilise l'expansion : Sih et Lee [77] prouvent que vérifier la vivacité d'un SDFG est équivalent à vérifier la vivacité de son expansion. Ce résultat s'applique aussi parfaitement aux CSDFG [19, 80]. On peut donc voir que sur la Figure 3.4 page 42 chaque expansion du graphe vérifie bien le théorème 1 page précédente. Cependant, comme la taille des graphes générés est exponentielle, cette méthode l'est aussi.

Une autre façon de vérifier la vivacité d'un SDFG [41, 52] ou d'un CSDFG [3] est d'effectuer l'exécution symbolique du graphe jusqu'à garantir l'absence d'un interblocage. Une exécution symbolique se résume à l'exécution successive de chacune des tâches du graphe. Pour vérifier la vivacité,

– il est nécessaire d'exécuter au moins une fois chacune des tâches du graphe,

– et de retrouver un état de marquage équivalent au marquage initial du graphe.

Par définition, le vecteur de répétition correspond donc exactement au nombre minimal d'exécutions nécessaires pour réaliser cette opération.

Pour le graphe de la Figure 3.1 page 37, on peut vérifier que la séquence d'exécution

$$A, A, B, B, C, A, C, A, B, B, C, C, C$$

ramène ce graphe à son état initial : le graphe est vivant. Encore une fois, cette méthode est exponentielle. On remarque néanmoins qu'en pratique, effectuer une exécution symbolique pour vérifier la vivacité est plus efficace que l'expansion. L'espace mémoire utilisé pour une exécution symbolique est de taille polynomiale, ce qui n'était pas le cas pour l'expansion.

À l'opposé de ces méthodes exactes, Marchetti et Munier-Kordon [63] proposent une nouvelle condition suffisante de vivacité des SDFG. Notons $\gcd_a$ le plus grand diviseur commun de out_a et in_a, le théorème suivant est alors démontré :

Théorème 3 *Marchetti et Munier-Kordon [63] Soit $\mathcal{G}$ un SDFG normalisé. $\mathcal{G}$ est vivant si pour tout circuit C de $\mathcal{G}$,*

$$\sum_{a \in C} M_0(a) > \sum_{a \in C} out_a - gcd_a$$

Remarquons que pour le cas d'un circuit de deux buffers, cette condition est nécessaire et suffisante [63].

Concernant les CSDFG, Benazouz [11] fournit deux nouvelles conditions suffisantes de vivacité qui généralisent ce résultat. Nous rappelons ces conditions dans le chapitre 6.

3.3.2 Évaluation de la fréquence de fonctionnement

Une application ΣC, lorsqu'elle s'exécute sur le MPPA, peut atteindre une certaine fréquence de fonctionnement. Afin d'évaluer la performance d'une application, il est nécessaire de calculer sa fréquence de fonctionnement maximale. Ce problème est alors équivalent à déterminer un ordonnancement valide atteignant cette fréquence de fonctionnement maximale. Pour être le plus juste possible, l'évaluation de ce débit doit tenir compte des ressources de calcul disponibles pour exécuter chacune des tâches, ce problème est NP-Complet [39]. Parce que la quantité de ressources disponibles sur un MPPA est élevée, et parce que nous traitons des instances de grande taille, nous limiterons notre étude à un problème sans limites de ressources. On sait d'ores et déjà qu'un ordonnancement *au plus tôt* et sans limites de ressource atteint toujours la fréquence de fonctionnement maximale de l'application. Pour les HSDFG, Chrétienne [25] montre que cet ordonnancement est de forme K-périodique ; cela signifie qu'il existe pour chaque tâche un motif d'exécution se répétant infiniment souvent. Reiter [74] montre qu'il existe également un ordonnancement périodique qui atteint cette même fréquence de fonctionnement. Il propose alors un programme linéaire pour résoudre ce problème ; ce programme linéaire s'appuie sur le résultat du théorème 4.

Théorème 4 *Reiter [74] Soit un HSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$; un ordonnancement périodique $\mathcal{S}$ du graphe $\mathcal{G}$ est valide, si et seulement si*

$$\forall a = (t, t') \in \mathcal{A} \quad \mathcal{S}\langle t', 1\rangle \geq \mathcal{S}\langle t, 1\rangle + d(t) - \mu_{\mathcal{G}}^{\mathcal{S}} \cdot M_0(a).$$

Reiter montre que ce problème peut être résolu à l'aide d'un algorithme polynomial.

Concernant les SDFG, l'évaluation du débit est un problème qui connait de nombreuses solutions, mais pour lequel aucun résultat de complexité clair n'est démontré. Sih et Lee [77] montrent que le débit maximal d'une expansion est équivalent à celui du graphe d'origine. Munier-Kordon [66] et de Groote et al. [31] améliorent le résultat de l'expansion

et par la même occasion, améliorent cette technique d'évaluation du débit. Cependant toutes ces méthodes sont de complexité exponentielle.

L'exécution symbolique permet aussi d'évaluer le débit maximal d'une application SDFG [42] ou CSDFG [82]. Pour déterminer le débit d'une application, l'on doit alors attendre de rencontrer deux états de fonctionnement identiques. Un état de fonctionnement tient compte à la fois du marquage (comme pour la vivacité), mais aussi de l'exécution des tâches (le temps restant avant la fin de leur exécution). Aujourd'hui, l'on ne sait pas mesurer la distance minimale entre deux états de fonctionnement identiques. De plus, pour rencontrer le premier état de fonctionnement pertinent, l'on doit atteindre la fin du régime transitoire, et il n'existe aucun résultat sur sa durée mise à part le fait que cette période est de durée finie. Pour évaluer la fréquence de fonctionnement, l'exécution symbolique est alors moins pertinente que l'expansion [31].

Une autre façon d'évaluer le débit d'une application est de limiter l'espace de recherche des solutions aux seuls ordonnancements périodiques. Pour les SDFG, ce problème devient polynomial : Benabid et al. [10] prouvent que n'importe quel ordonnancement périodique d'un SDFG peut être caractérisé par un programme linéaire de $\Theta(|\mathcal{A}|)$ équations portant sur les premières dates de départ de chaque tâche. Il est aussi prouvé que le débit maximal d'un ordonnancement périodique reste une borne inférieure du débit maximal de l'application.

Une approche d'évaluation du débit des ordonnancements périodique appliquée aux CSDFG fut ensuite proposée par Bamakhrama et Stefanov [6]. Cependant, cette méthode se limite aux graphes acycliques et ne peut donc pas tenir compte des tailles mémoires.

3.3.3 Dimensionnement mémoire

La plupart des modèles dataflows ne limitent pas le nombre de jetons dans un buffer. Pourtant, les applications qu'ils décrivent doivent nécessairement fonctionner avec des mémoires de taille finie. Qui plus est, dans un contexte embarqué, cette quantité de mémoire se doit d'être le plus faible possible ; rappelons que la mémoire est un des points les plus critiques de la conception des systèmes embarqués.

Selon l'exécutif utilisé, le modèle mémoire peut varier. La mémoire peut, par exemple, être partagée entre les différents buffers d'une application [35]. Dans ce cas, si l'on note $M_\tau(a)$ la quantité de jetons présents dans un buffer a à un instant τ et $MaxSize(\mathcal{G})$ la quantité de mémoire disponible pour un dataflow $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ alors

$$\forall \tau, \quad \sum_{a \in \mathcal{A}} M_\tau(a) \leq MaxSize(\mathcal{G}).$$

Aux contraintes, avec le *runtime* de la chaîne de compilation AccessCore, les buffers doivent être dimensionnés séparément, ils ne partagent pas leurs espaces mémoire. Si l'on

note $B(a)$ la quantité maximale de données autorisée dans un buffer correspondant à un arc a alors un dataflow $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ doit vérifier

$$\forall \tau \ \ \forall a \ \ M_\tau(a) \leq B(a).$$

Afin de modéliser cette borne mémoire, il existe une transformation de graphe qui, pour chaque arc $a = (t, t') \in \mathcal{A}$, intègre a' un nouvel arc retour (*feedback* en anglais) tel que $a' = (t', t)$. Les taux de cet arc retour doivent alors être symétriques avec ceux de l'arc d'origine ($out_{a'} = in_a$ et $in_{a'} = out_a$) et le marquage de cet arc correspond à l'espace disponible dans le buffer c.-à-d. $M_0(a') + M_0(a) = B(a)$. Un exemple de cette transformation est visible sur la Figure 3.5.

Pour un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, le *feedback* buffer a' d'un buffer $a = (t, t') \in \mathcal{A}$ est défini de la manière suivante :

$$\begin{aligned} \forall k \in \{1, \cdots, \varphi(t)\}, \quad out_{a'}(k) &= in_a(k) \\ \forall k' \in \{1, \cdots, \varphi(t')\}, \quad in_{a'}(k') &= out_a(k') \\ M_0(a') &= B(a) - M_0(a) \end{aligned}$$

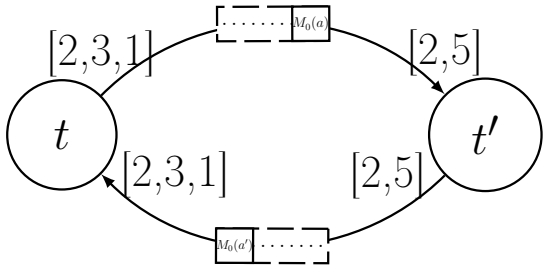


FIGURE 3.5 – Modélisation d'un buffer bornée pour un modèle CSDFG.

On défini alors l'ensemble des arcs retours $Fb(\mathcal{A})$ par

$$\forall a = (t, t') \in \mathcal{A} \ \ \exists a' = (t', t) \in Fb(\mathcal{A}).$$

Au sein de la chaîne de compilation AccessCore, le dimensionnement mémoire doit être effectué dans deux contextes différents.

Dimensionnement mémoire minimal garantissant la vivacité

Premièrement, il est nécessaire de trouver un dimensionnement mémoire minimal qui garantisse la vivacité d'un CSDFG.

Ce problème est connu pour être NP-Complet, même pour le cas simple des HSDFG ; il s'agit alors d'un problème équivalent au marquage minimal d'un graphe d'évènement garantissant la vivacité [84].

Pour les SDFG, Ade et al. [2] proposent une heuristique de dimensionnement polynomiale basée sur l'analyse de motifs spécifiques. Ils montrent ensuite une borne inférieure de la taille d'un buffer :

$$B(a)_{inf} = in_a + out_a - \gcd_a.$$

Dans la chaîne de compilation AccessCore, une méthode approchée est utilisée ; il s'agit d'une heuristique gloutonne inspirée par [80]. Le principe de cette méthode est d'exécuter une à une les tâches d'un dataflow jusqu'à revenir à son état initial. L'ordre dans lequel les tâches sont exécutées va définir les tailles mémoires ; le principe de cette heuristique est de toujours privilégier les exécutions qui minimisent localement la taille mémoire.

Plus tard dans le flot de compilation, lorsque les durées d'exécutions des tâches sont déterminées et qu'une fréquence de fonctionnement minimale est fixée, une deuxième étape survient ; il s'agit du dimensionnement mémoire minimal sous contrainte de débit.

Dimensionnement avec contrainte de débit

Étant donnée une fréquence de fonctionnement fixée pour une ou plusieurs tâches, le dimensionnement minimal avec contrainte de débit est alors un problème d'optimisation qui consiste à trouver des valeurs entières $M_0(a')$ pour chaque arc $a' \in Fb(\mathcal{A})$ telles que

1. le débit requis est atteint ;
2. la taille mémoire est minimale.

Ning et Gao [68] proposent une première solution de dimensionnement minimal des HSDFG à débit maximal supposée optimale ; elle fut ensuite corrigée par Moreira et al. [65].

Pour résoudre ce problème de dimensionnement à débit fixe pour les SDFG, Govindarajan et Gao [46] utilisent l'expansion accompagnée d'une technique similaire à celle proposée par Ning et Gao [68]. Benazouz et al. [14] proposent aussi une méthode de dimensionnement minimal sous contrainte de débit fixe. Cette méthode se limite aux SDFG pour lesquels un ordonnancement périodique existe, la solution proposée est une borne supérieure de la solution optimale.

Une première méthode appliquée aux CSDFG fut proposée par Stuijk et al. [82]. Celle-ci s'appuie alors sur une recherche exhaustive des solutions de dimensionnement, en utilisant pour fonction d'objectif une méthode d'évaluation du débit par exécution symbolique. La combinaison de ces deux méthodes combinatoires rend cette technique inefficace même sur

des instances de petite taille ; notons qu'il s'agit néanmoins d'une des uniques méthodes exactes de la littérature pour les CSDFG.

Une autre façon de résoudre ce problème consiste à limiter encore une fois les solutions à des sous-classes d'ordonnancements comme les ordonnancements périodiques. Wiggers et al. [94] remarquent par exemple qu'une sous-classe d'ordonnancement périodique peut être étudiée en fixant *a priori* l'écart entre les dates de départ des premières phases de chaque tâche.

Ils proposent deux politiques pour fixer ces écarts, une première que nous nommerons *Average* (toutes les dates de départ sont équitablement réparties dans une période d'exécution de la tâche) et une méthode dite *Phase Schedule*, basée sur un ordonnancement local des phases de sorte d'éviter une exécution précipitée des tâches. Benazouz et al. [13] observent ensuite que ce problème peut être entièrement résolu par un programme linéaire. Encore une fois, différentes règles pour fixer les dates de départ sont proposées (*Burst*, où toutes les exécutions de phase d'une tâche se font sans interruption, et *Average*, identique à la définition précédente). Plus récemment Benazouz et Munier-Kordon [12] prouvent que l'ordonnancement des phases peut aussi se modéliser par un programme linéaire. Ils proposent alors une méthode en deux étapes : le Min-Max. L'algorithme Min-Max montre de meilleures performances que toutes les méthodes précédentes ; néanmoins la complexité du séquencement des phases du Min-Max est quadratique là où celle de Wiggers et al. [94] était linéaire. Des exemples de ces différentes politiques sont visibles sur la Figure 3.6.

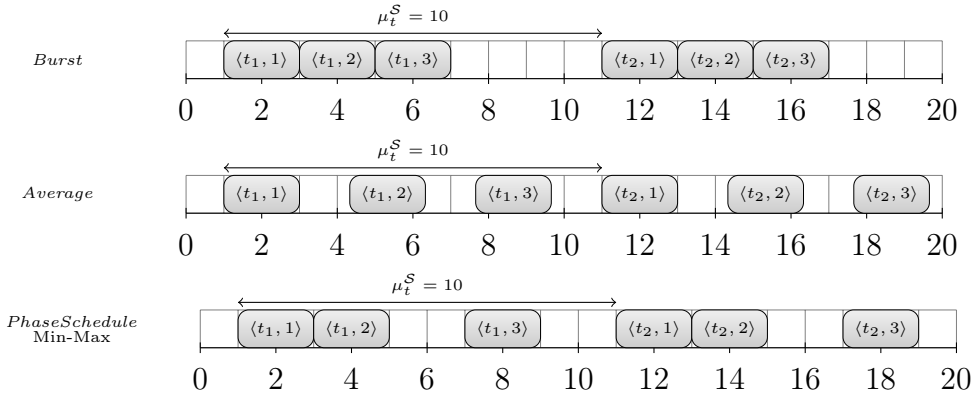


FIGURE 3.6 – Différentes politiques proposées pour fixer les écarts entre les premières dates d'exécution des phases d'un CSDFG.

3.4 Nos contributions

Nos contributions sont le prolongement de travaux théoriques en recherche opérationnelle effectués par l'équipe ALSOC du LIP6. Le cadre industriel dans lequel s'est déroulée cette thèse nous a également amené à étudier l'application de ces contributions à

la compilation dataflow. Plusieurs de nos contributions sont aujourd'hui en production, d'autres en cours d'intégration.

3.4.1 Ordonnancement et analyse statique

Dans le chapitre 4, nous généralisons aux CSDFG l'ordonnancement périodique défini par Benabid et al. [10] pour les SDFG. Cette première contribution est un résultat fondamental pour nos travaux. Dans le chapitre 6, nous proposons différents algorithmes d'analyse statique (portant sur l'évaluation du débit et sur le dimensionnement mémoire) directement déduits de cette contribution. Ces travaux sont publiés sous l'intitulé « Periodic Schedules for Cyclo-static Dataflow » [21].

Dans le chapitre 5, nous allons plus loin dans la caractérisation des ordonnancements en introduisant la notion d'ordonnancement K-périodique. Cette classe d'ordonnancement généralise les ordonnancements périodiques et les ordonnancements *au plus tôt* des SDFG et des CSDFG. Il s'agit d'un outil particulièrement décisif pour l'amélioration des résultats d'analyse que nous produisons. Cette contribution est publiée sous l'intitulé « K-Periodic Schedules for Evaluating the Maximum Throughput of a Synchronous Dataflow Graph » [20].

Nous étudions alors trois problèmes d'analyse dataflow.

Vivacité

Afin d'améliorer les performances des méthodes d'évaluation de la vivacité, nous avons participé à l'élaboration de deux conditions suffisantes de vivacité pour les CSDFG : les conditions SC1 et SC2. Nous prouvons maintenant leur équivalence, et nous fournissons deux algorithmes polynomiaux pour évaluer ces conditions. Finalement, un algorithme d'évaluation de la vivacité est proposé. Ses performances sont supérieures à la technique la plus reconnue jusqu'ici : l'exécution symbolique. Ces travaux sont publiés sous l'intitulé « Liveness Evaluation of a Cyclo-Static Dataflow Graph » [15].

Évaluation du débit

En nous appuyant sur nos contributions théoriques (la caractérisation d'un ordonnancement périodique des CSDFG), nous proposons un algorithme exact d'évaluation du débit maximal d'un ordonnancement périodique. Cet algorithme fournit implicitement une borne inférieure du débit maximal pour un ordonnancement quelconque. Il permet d'obtenir une première évaluation du débit en un temps extrêmement court. L'utilisation de notre méthode permet alors d'obtenir de premières estimations de performance lorsque l'exécution symbolique ne fournit aucune solution, faute d'un temps de calcul trop élevé.

Cette contribution fait partie des travaux présentés dans l'article « Periodic Schedules for Cyclo-static Dataflow » [21].

Dimensionnement mémoire

Pour résoudre le problème de dimensionnement minimal garantissant la vivacité, nous utilisons les conditions suffisantes de vivacité SC1 et SC2 (cités précédemment). Nous fournissons une technique de dimensionnement sous forme d'un programme linéaire. Nous comparons les performances de cette méthode avec celle de l'algorithme de dimensionnement actuellement en place dans la chaîne de compilation AccessCore.

On constate expérimentalement que notre méthode est alors bien plus rapide, et fournit de meilleures solutions pour de nombreux cas. Notons néanmoins que certaines applications ne peuvent être dimensionnées par cette méthode si elles n'admettent pas d'ordonnancement périodique. Ces résultats sont publiés dans l'article « Liveness evaluation of a cyclo-static dataflow graph » [15].

Pour résoudre le problème de dimensionnement avec contrainte de débit, nous utilisons à nouveau les résultats d'ordonnancement du chapitre 4. Nous proposons deux algorithmes :

- L'un est un programme linéaire en nombres entiers, PSizing-ILP,
- l'autre est son modèle relaxé, PSizing-LP.

Par définition PSizing-ILP donne les solutions optimales périodiques : il est donc nécessairement meilleur que toutes les autres méthodes périodiques existantes. Nous remarquons que PSizing-LP fourni des résultats très proches de PSizing-ILP, et semble bien souvent plus performant que les principales méthodes périodiques connues [93, 13, 12]. Ces algorithmes font partie des contributions présentées dans l'article « Periodic Schedules for Cyclo-static Dataflow » [21].

3.4.2 Intégration

Afin d'intégrer ces différentes méthodes à la chaîne de compilation AccessCore, nous avons étudié d'autres modèles dataflows. Dans le chapitre 7, nous présentons le résultat de cette étude. Nous généralisons les travaux de Benabid et al. [10] aux *Computation Graphs* ainsi qu'à deux autres modèles plus expressifs que les *Cyclo-Static Dataflow Graphs*.

3.4.3 Générateur d'instances aléatoires

Finalement, pour évaluer l'ensemble de nos contributions, nous avons développé un générateur d'instances aléatoires. Nous ne connaissons qu'un unique générateur de CSDFG [81], et il n'était pas assez rapide pour générer les graphes nécessaires à nos expérimentations

(nous souhaitons qu'il soit complexe et de grande taille). Le générateur que nous fournissons comble ce manque. Son temps de calcul et bien moins élevé, et les graphes générés disposent de propriétés plus intéressantes. Les jeux d'essai qui ont été générés sont aujourd'hui en ligne [48], et nous planifions la mise en ligne du générateur dans les mois à venir.

Conclusion

Une fois abstraction faite du langage de programmation ΣC au profit des modèles SDFG et CSDFG dans le chapitre précédent, nous nous sommes maintenant concentrés sur l'analyse de ces modèles. Nous avons tout d'abord présenté différentes techniques essentielles à la compréhension de l'état de l'art. Puis nous nous sommes concentrés sur les trois problèmes étudiés dans cette thèse.

On remarque, malgré les 20 à 25 années d'existences des modèles SDFG et CSDFG, que la majeure partie des techniques utilisées pour leur analyse ne peuvent s'appliquer qu'à des instances de petite taille.

Le leitmotiv de nos travaux a donc été de fournir des solutions garantissant un passage à l'échelle. Nous souhaitons fournir une solution dans un temps raisonnable, même si celle-ci n'est pas optimale. Dans cette optique, le chapitre suivant présente une caractérisation de l'ordonnancement périodique d'un CSDFG. Cette caractérisation est polynomiale, et elle sera utilisée par la suite pour définir différents algorithmes d'analyse.

Première partie

Contributions à l'ordonnancement des dataflows

Chapitre 4

Ordonnancement périodique

Introduction

Pour mener à bien la compilation d'une application ΣC, le logiciel AccessCore effectue différentes analyses dataflows. Ces analyses portent notamment sur trois points essentiels : la vivacité, l'évaluation du débit, et le dimensionnement mémoire.

Les techniques utilisées pour résoudre ces problèmes s'appuient généralement sur un résultat majeur des ordonnancements cycliques : l'ordonnancement *au plus tôt* d'un dataflow atteint toujours le débit maximum de l'application.

Partant de ce résultat, deux familles d'algorithmes virent le jour :

– celles à base d'exécution symbolique *au plus tôt* [41, 3],

– et celles à base d'expansion [77, 66, 31].

Néanmoins, pour les SDFG et les CSDFG, nous savons que la taille minimale d'un ordonnancement *au plus tôt* est exponentielle en la taille de l'instance, et qu'il en va de même pour leur expansion. Face à l'augmentation de la taille des instances à traiter, ces techniques seront alors inefficaces.

Pour cette raison, des techniques approchées portant sur les ordonnancements *périodiques* furent envisagées. Ramamoorthy et Ho [73] prouve qu'il existe toujours un ordonnancement périodique de débit maximal pour les HSDFG ; il propose également un algorithme polynomial pour calculer ce débit. Bien que cette propriété ne soit plus vraie pour les SDFG [10], l'approche périodique fut également appliquée aux SDFG pour résoudre les problèmes de vivacité, d'évaluation du débit [10] et de dimensionnement mémoire [13]. Les algorithmes obtenus sont des méthodes approchées, mais leurs résultats demeurent particulièrement intéressants. Par exemple, pour les SDFG acycliques, l'évaluation du débit est exacte [10]. De la même manière, Wiggers et al. [93] utilisent les caractéristiques des ordonnancements périodiques pour dimensionner des CSDFG. Ils ne considèrent alors qu'une unique date de départ par tâche, et cela, quel que soit leur nombre de phases. Bamakhrama et al. [8] expérimenteront également l'ordonnancement périodique pour évaluer le débit d'un CSDFG. Néanmoins, leur méthode ne s'applique alors qu'aux graphes acycliques.

Dans ce chapitre, nous présentons une définition de l'ordonnancement périodique d'un CSDFG. Nous caractérisons alors sous forme de contraintes linéaires les relations de précédences induites par un buffer pour un ordonnancement périodique. Notre but est de généraliser aux CSDFG les travaux d'ordonnancement de Benabid et al. [10] afin de vérifier la validité d'un ordonnancement périodique. Ces résultats s'appliqueront ensuite à l'évaluation du débit maximal ou du dimensionnement mémoire minimal dans le chapitre 6.

4.1 Ordonnancement périodique d'un CSDFG

Un ordonnancement doit définir les dates d'exécution d'un ensemble de tâches. Comme les CSDFG modélisent des applications qui s'exécutent infiniment souvent, l'ordonnancement d'un CSDFG doit alors définir une infinité de dates d'exécutions ; on parle généralement d'ordonnancements *cycliques*.

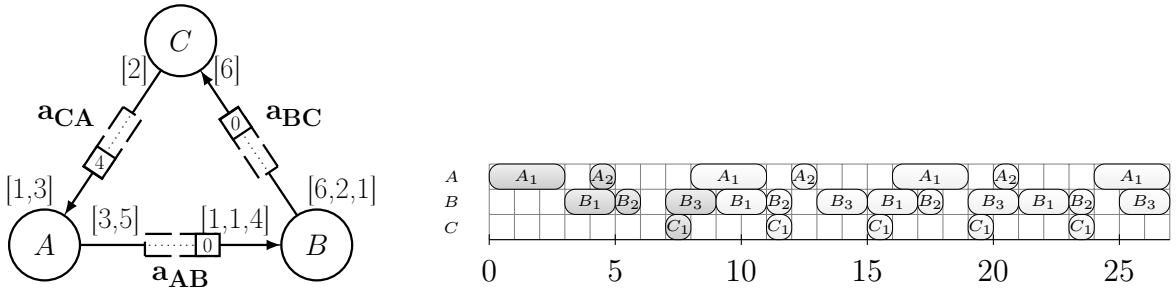
Pour un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ et pour toute tâche $t \in \mathcal{T}$, on note t_k la k^{e} phase de la tâche t avec $k \in \{1, \cdots, \varphi(t)\}$, et $\langle t_k, n\rangle$ la n^{e} exécution de la phase t_k avec $n \geq 1$.

Si l'on considère un ordonnancement $\mathcal{S}$, alors on note $\mathcal{S}\langle t_k, n\rangle$ la date de départ de l'exécution $\langle t_k, n\rangle$ pour cet ordonnancement. Un ordonnancement cyclique est dit *périodique* s'il respecte la définition suivante.

Définition 4 *Soit $\mathcal{S}$ l'ordonnancement* périodique *d'un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Pour toute tâche $t \in \mathcal{T}$, si l'on note $\mu_t^{\mathcal{S}}$ la période de fonctionnement de cette tâche pour l'ordonnancement $\mathcal{S}$, l'ensemble des dates d'exécution définies par $\mathcal{S}$ doivent vérifier*

$$\forall n \geq 1, \forall k \in \{1, \cdots, \varphi(t)\} \quad \mathcal{S}\langle t_k, n\rangle = \mathcal{S}\langle t_k, 1\rangle + (n-1)\mu_t^{\mathcal{S}}.$$

D'après cette définition, l'ordonnancement périodique d'un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ est donc défini par les périodes $\mu_t^{\mathcal{S}}$ de chaque tâche $t \in \mathcal{T}$ ainsi que leurs $\varphi(t)$ premières dates d'exécution.



(a) Un CSDFG consistant et vivant

(b) Un ordonnancement périodique pour ce CSDFG

FIGURE 4.1 – (a) Un CSDFG composé de trois tâches. (b) Un ordonnancement périodique valide pour ce CSDFG. Les périodes de fonctionnement de ses tâches sont $\mu_A^{\mathcal{S}} = 8$, $\mu_B^{\mathcal{S}} = 6$ et $\mu_A^{\mathcal{S}} = 4$. Les exécutions en surbrillances correspondent aux dates de départ initiales des tâches, les suivantes sont induites par leur période.

La Figure 4.1(b) montre un exemple d'ordonnancement périodique pour le CSDFG associé. La première date d'exécution de chaque phase d'une tâche est marquée en surbrillance. Ses exécutions suivantes sont ensuite calculées en fonction de sa période $\mu_t^{\mathcal{S}}$. Pour la tâche A, qui compte $\varphi(A) = 2$ phases, on note bien que deux exécutions sont fixées. Comme $\mu_A^{\mathcal{S}} = 8$, la prochaine exécution de A, $\langle A_1, 2\rangle$ à lieu à l'instant $\mathcal{S}\langle A_1, 2\rangle = \langle A_1, 1\rangle + \mu_A^{\mathcal{S}} = 8$.

D'après la définition 1 page 35, la fréquence de fonctionnement d'une tâche t pour un ordonnancement $\mathcal{S}$ est notée

$$Th_t^{\mathcal{S}} = \lim_{n\to\infty} \frac{n}{\mathcal{S}\langle t, n\rangle}.$$

Pour l'ordonnancement périodique d'un CSDFG, on obtient alors par substitution :

$$Th_t^{\mathcal{S}} = \lim_{n\to\infty} \frac{n}{\mathcal{S}\langle t_k, 1\rangle + (n-1)\mu_t^{\mathcal{S}}} = \frac{1}{\mu_t^{\mathcal{S}}}$$

Lorsqu'un CSDFG modélise une application réelle, il doit pouvoir s'exécuter avec une quantité de mémoire finie. En moyenne, la quantité de données produite dans un buffer doit donc être égale à la quantité de données lue. Le théorème 5 fut démontré sur cette idée.

Théorème 5 *Munier-Kordon [66], Stuijk et al. [82] Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, un SDFG ou un CSDFG consistant à mémoire borné, et $\mathcal{S}$ un ordonnancement valide. On note $N^{\mathcal{G}}$ le vecteur de répétition du graphe $\mathcal{G}$. Alors pour chaque couple de tâches $(t, t') \in \mathcal{T}^2$,*

$$\frac{Th_t^{\mathcal{S}}}{N_t^{\mathcal{G}}} = \frac{Th_{t'}^{\mathcal{S}}}{N_{t'}^{\mathcal{G}}}.$$

On déduit alors la définition d'une période normalisée.

Définition 5 *Lorsqu'un CSDFG consistant est à mémoire bornée, il existe un équilibre entre la fréquence de fonctionnement de ses tâches. Cet équilibre implique l'existence d'une période normalisée $\Omega_{\mathcal{G}}^{\mathcal{S}}$ définie par*

$$\Omega_{\mathcal{G}}^{\mathcal{S}} = \mu_t^{\mathcal{S}} \cdot N_t^{\mathcal{G}} \quad \forall t \in \mathcal{T}.$$

4.2 Relation de précédence

L'exécution d'une tâche est conditionnée par la présence de données dans chacun de ses buffers d'entrée. Le lien qui existe entre les exécutions des tâches productrices et consommatrices est appelé une *relation de précédence*.

4.2.1 Définition d'une relation de précédence

Munier-Kordon [66] montre que l'existence d'une relation de précédence entre deux exécutions d'un SDFG vérifie trois conditions simples (elles ont été rappelées dans le chapitre 3). Benazouz [11] étend cette définition aux CSDFG.

Pour tout couple $(k, n) \in \{1, \cdots, \varphi(t)\} \times \mathbb{N} - \{0\}$, on note $Pr\langle t_k, n\rangle$ l'exécution de la tâche t qui précède l'exécution $\langle t_k, n\rangle$. Plus formellement,

$$Pr\langle t_k, n\rangle = \begin{cases} \langle t_{k-1}, n\rangle & si\ k > 1 \\ \langle t_{\varphi(t)}, n-1\rangle & si\ k = 1 \end{cases}$$

L'exécution $\langle t_{\varphi(t)}, 0\rangle$ est fictive, et n'est introduite que pour simplifier la définition de Pr. Notons également les deux quantités $I_a\langle t_k, n\rangle$ et $O_a\langle t'_{k'}, n'\rangle$, le cumul de jetons produits et consommés par les tâches t et t' :

$$\begin{aligned} I_a\langle t_k, n\rangle &= I_a Pr\langle t_k, n\rangle + in_a(k) \\ O_a\langle t'_{k'}, n'\rangle &= O_a Pr\langle t'_{k'}, n'\rangle + out_a(k'). \end{aligned}$$

Définition 6 *Benazouz [11] Un buffer $a = (t, t')$ induit une relation de précédence stricte entre les exécutions $\langle t_k, n\rangle$ et $\langle t_{k'}, n'\rangle$ si et seulement si :*

1. *$\langle t'_{k'}, n'\rangle$ peut être exécuté après $\langle t_k, n\rangle$*
2. *$\langle t'_{k'}, n'\rangle$ ne peut pas être exécuté avant $\langle t_k, n\rangle$*
3. *$Pr\langle t'_{k'}, n'\rangle$ peut s'exécuter avant la fin de $\langle t_k, n\rangle$.*

Pour illustrer cette propriété, nous considérerons l'arc a_{AB} reliant les tâches A et B de la Figure 4.1(a) page 57. Vérifions qu'il existe bien une relation de précédence entre les exécutions $\langle A_2, 2\rangle$ et $\langle B_2, 3\rangle$:

– La quantité de données présentes dans le buffer a_{AB} après l'exécution $\langle A_2, 2\rangle$ est

$$M_0(a_{AB}) + I_{a_{AB}}\langle A_2, 2\rangle = 0 + 16 = 16.$$

Comme elle est supérieure à la quantité de données consommées après l'exécution $\langle B_2, 3\rangle$,

$$\begin{aligned} M_0(a_{AB}) + I_{a_{AB}}\langle A_2, 2\rangle &\geq O_{a_{AB}}\langle B_2, 3\rangle \\ 0 + 16 &\geq 12 \end{aligned}$$

alors $\langle B_2, 3\rangle$ peut être exécutée après $\langle A_2, 2\rangle$ (Condition 1).

– De la même manière,

$$\begin{aligned} M_0(a_{AB}) + I_{a_{AB}}\langle A_2, 2\rangle &< O_{a_{AB}}\langle A_1, 2\rangle \\ 4 + 0 &< 5, \end{aligned}$$

donc $\langle B_2, 3\rangle$ ne peut être exécuté avant $\langle A_2, 2\rangle$ (Condition 2).

– et pour finir

$$\begin{aligned} M_0(a_{AB}) + I_{a_{AB}}\langle A_2, 2\rangle &\geq O_{a_{AB}} Pr\langle B_2, 3\rangle \\ 4 + 0 &\geq 4, \end{aligned}$$

alors $Pr\langle B_2, 3\rangle$ peut s'exécuter avant la fin de $\langle A_2, 2\rangle$ (Condition 3).

Les trois conditions sont vérifiées ; il existe donc bien une relation de précédence entre les exécutions $\langle A_2, 2\rangle$ et $\langle B_2, 3\rangle$.

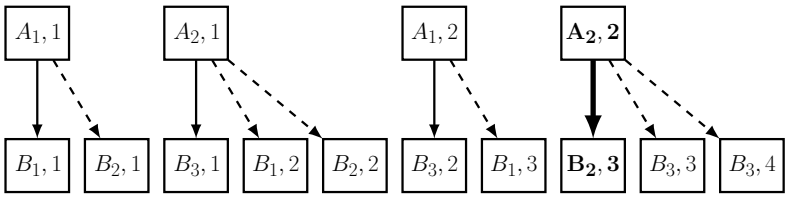


FIGURE 4.2 – Une représentation graphique des premières relations de précédence intervenant dans le CSDFG de la Figure 4.1(a) page 57. Les arcs pleins sont les relations de précédence stricte.

Sur la Figure 4.2, nous avons représenté les premières relations de précédences induites par le buffer a_{AB} du CSDFG de la Figure 4.1(a) page 57. Chaque nœud est une exécution et les arcs représentent les relations de précédence entre ces exécutions. Les arcs pleins sont les relations de précédence stricte que nous traitons. On note l'existence d'un arc entre les exécutions $\langle A_2, 2\rangle$ et $\langle B_2, 3\rangle$.

4.2.2 Formulation de l'existence d'une relation de précédence

Benazouz [11] propose de formaliser l'existence d'une relation de précédence entre deux exécutions d'un CSDFG par une inéquation ; il s'agit du lemme 3 :

Lemme 3 *Benazouz [11] Soit l'arc $a = (t, t') \in \mathcal{A}$, ainsi que deux exécutions $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ avec $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ et $(n, n') \in (\mathbb{N} - \{0\})^2$. Il existe une relation de précédence entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ si et seulement si*

$$in_a(k) > M_0(a) + I_a\langle t_k, n\rangle - O_a\langle t'_{k'}, n'\rangle \geq \max\{0, in_a(k) - out_a(k')\}.$$

Pour l'exemple vu précédemment,

$$\begin{aligned} max\{0, in_a(k) - out_a(k')\} &= 0 \\ M_0(a_{AB}) + I_{a_{AB}}\langle A_1, 1\rangle - O_{a_{AB}}\langle B_1, 2\rangle &= 0. \end{aligned}$$

L'inégalité $2 > 0 \geq 0$ est vérifiée, ce qui s'accorde avec notre résultat précédent.

La définition d'un ordonnancement valide est un ordonnancement vérifiant l'ensemble des relations de précédence induites par les buffers. Dans le cas des ordonnancements cycliques, ce nombre de relations de précédence est infini. Pour vérifier qu'un ordonnancement est valide, nous devons donc caractériser cet ensemble infini au travers d'un nombre de contraintes fini.

4.3 Périodicité des relations de précédence

Plutôt que de définir une condition vérifiant l'existence d'une relation de précédence entre deux exécutions $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ (lemme 3 page ci-contre), nous allons maintenant définir une contrainte vérifiant l'existence de relations de précédence entre deux phases de deux tâches. Par la suite, nous vérifions qu'un ordonnancement périodique respecte bien chacune de ces relations de précédence.

4.3.1 Une condition d'existence nécessaire

Considérons le buffer $a = (t, t') \in \mathcal{A}$. Le lemme qui suit est une condition nécessaire d'existence d'une relation de précédence entre deux exécutions quelconques de deux phases t_k et $t'_{k'}$ fixées.

Pour tout couple $(\beta, \gamma) \in \mathbb{Z} \times \mathbb{N} - \{0\}$, considérons $\lfloor\beta\rfloor^\gamma$ et $\lceil\beta\rceil^\gamma$ définies par :

$$\lfloor\beta\rfloor^\gamma = \left\lfloor \frac{\beta}{\gamma} \right\rfloor \times \gamma \quad \text{et} \quad \lceil\beta\rceil^\gamma = \left\lceil \frac{\beta}{\gamma} \right\rceil \times \gamma.$$

On note également pour tout arc $a = (t, t') \in \mathcal{A}$, $\gcd_a = \gcd(i_a, o_a)$ le plus grand diviseur commun entre i_a et o_a.

Rappelons que i_a (*resp.* o_a) est la somme des productions (*resp.* des consommations) dans le buffer a après une itération :

$$i_a = \sum_{k=1}^{\varphi(t)} in_a(k) \quad o_a = \sum_{k=1}^{\varphi(t')} out_a(k).$$

Lemme 4 *Considérons l'arc $a = (t, t') \in \mathcal{A}$, le couple de valeur $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ et deux exécutions $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ avec $(n, n') \in (\mathbb{N} - \{0\})^2$. On note aussi $\alpha = (n-1) \times i_a - (n'-1) \times o_a$.*

Si a induit une relation de précédence entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ alors

$$\alpha_a^{\min}(k, k') \leq \alpha \leq \alpha_a^{\max}(k, k')$$

avec

$$\begin{aligned} \alpha_a^{\min}(k,k') &= \lceil \max\{0, in_a(k) - out_a(k')\} + O_a\langle t'_{k'}, 1\rangle - I_a\langle t_k, 1\rangle - M_0(a) \rceil^{gcd_a} \\ et \quad \alpha_a^{\max}(k,k') &= \lfloor O_a\langle t'_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - M_0(a) - 1 \rfloor^{gcd_a} . \end{aligned}$$

PREUVE : D'après le lemme 3 page 60, l'existence d'une relation de précédence entre deux exécutions $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ est équivalente à

$$in_a(k) > M_0(a) + I_a\langle t_k, n\rangle - O_a\langle t'_{k'}, n'\rangle \geq \max\{0, in_a(k) - out_a(k')\}.$$

Comme

$$I_a\langle t_k, n\rangle = (n-1)i_a + I_a\langle t_k, 1\rangle,$$

$$O_a\langle t'_{k'}, n'\rangle = (n'-1)o_a + O_a\langle t'_{k'}, 1\rangle$$

Nous obtenons

$$in_a(k) > M_0(a) + \alpha + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle \geq \max\{0, in_a(k) - out_a(k')\}.$$

1. L'inégalité de gauche devient alors

   $$O_a\langle t'_{k'}, 1\rangle - (I_a\langle t_k, 1\rangle - in_a(k)) - M_0(a) - 1 \geq \alpha.$$

   Et comme

   $$I_a\langle t_k, 1\rangle = I_a Pr\langle t_k, 1\rangle + in_a(k),$$

   on obtient

   $$O_a\langle t'_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - M_0(a) - 1 \geq \alpha.$$

   Comme α est divisible par gcd_a, l'inégalité $\alpha_a^{\max}(k,k') \geq \alpha$ est vérifiée.

2. De la même manière, l'inégalité de droite devient

   $$\alpha \geq \max\{0, in_a(k) - out_a(k')\} + O_a\langle t'_{k'}, 1\rangle - I_a\langle t_k, 1\rangle - M_0(a),$$

   alors $\alpha \geq \alpha_a^{\min}(k,k')$, ce qui conclut cette preuve.

∎

Une conséquence simple du lemme 4 page précédente est que si $\alpha_a^{\min}(k,k') > \alpha_a^{\max}(k,k')$, alors il n'existe pas de couple d'entiers (n,n') tels que a puisse induire une contrainte d'exécution entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$.

Par exemple, considérons l'arc $a_{AB} = (A,B)$ de la Figure 4.1(a) page 57. Comme $\alpha_{a_{AB}}^{\min}(1,3) = 4$ et $\alpha_{a_{AB}}^{\max}(1,3) = 0$, alors $\alpha_{a_{AB}}^{\min}(1,3) > \alpha_{a_{AB}}^{\max}(1,3)$ et il n'existe aucune relation

de précédence induite par a_{AB} entre les exécutions $\langle A_1, n\rangle$ et $\langle B_3, n'\rangle$ qu'elle que soit les valeurs de n et n'.

4.3.2 Une condition d'existence suffisante

Nous allons maintenant présenter une condition suffisante à l'existence d'une relation de précédence entre deux phases. Le lemme 5 est la réciproque du lemme 4 page 61.

Lemme 5 *Soit un arc $a = (t, t') \in \mathcal{A}$ et un couple $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$. Si*

$$\alpha_a^{\min}(k, k') \leq \alpha_a^{\max}(k, k'),$$

alors il existe une infinité de couples $(n, n') \in (\mathbb{N} - \{0\})^2$ tels que l'arc a induit une relation de précédence entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$.

PREUVE : D'après Bezout, il existe $(x, y) \in (\mathbb{Z} - \{0\})^2$ tels que $x \times i_a - y \times o_a = \gcd_a$.

Pour tout entier z, les valeurs

$$\begin{aligned} n(z) &= 1 + x \times \frac{\alpha_a^{\max}(k, k')}{\gcd_a} + z \times o_a \\ \text{et} \quad n'(z) &= 1 + y \times \frac{\alpha_a^{\max}(k, k')}{\gcd_a} + z \times i_a \end{aligned}$$

vérifient alors

$$\begin{aligned} (n(z)-1)i_a - (n'(z)-1)o_a &= (x \times \frac{\alpha_a^{\max}(k, k')}{\gcd_a} + z \times o_a)i_a - (y \times \frac{\alpha_a^{\max}(k, k')}{\gcd_a} + z \times i_a)o_a \\ (n(z)-1)i_a - (n'(z)-1)o_a &= \frac{x \times i_a - y \times o_a}{\gcd_a}\alpha_a^{\max}(k, k') \\ &= \alpha_a^{\max}(k, k'). \end{aligned}$$

Lorsque z est suffisamment grand, $n(z)$ et $n'(z)$ sont positifs.

Soit $A(z) = M_0(a) + I_a\langle t_k, n(z)\rangle - O_a\langle t'_{k'}, n'(z)\rangle$. Prouvons maintenant que

$$\max\{0, in_a(k) - out_a(k')\} \leq A(z) < in_a(k).$$

D'après les définitions de I_a et O_a,

$$\begin{aligned} I_a\langle t_k, n(z)\rangle &= I_a\langle t_k, 1\rangle + (n(z)-1)i_a \\ O_a\langle t'_{k'}, n'(z)\rangle &= O_a\langle t'_{k'}, 1\rangle + (n'(z)-1)o_a, \end{aligned}$$

et $A(z)$ peut-être réécrit,

$$\begin{aligned} A(z) &= M_0(a) + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle + (n(z)-1)i_a - (n'(z)-1)o_a \\ &= M_0(a) + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle + \alpha_a^{\max}(k, k'). \end{aligned}$$

– D'après la définition de $\alpha_a^{\max}(k, k')$,

$$\alpha_a^{\max}(k, k') < -M_0(a) - I_a Pr\langle t_k, 1\rangle + O_a\langle t'_{k'}, 1\rangle.$$

Alors,

$$A(z) < M_0(a) + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle - M_0(a) - I_a Pr\langle t_k, 1\rangle + O_a\langle t'_{k'}, 1\rangle.$$

Comme $I_a Pr\langle t_k, 1\rangle + in_a(k) = I_a\langle t_k, 1\rangle$, l'inégalité $A(z) < in_a(k)$ est vérifiée.

– Maintenant, d'après l'hypothèse, $\alpha_a^{\min}(k, k') \leq \alpha_a^{\max}(k, k')$, alors

$$\alpha_a^{\max}(k, k') \geq \max\{0, in_a(k) - out_a(k')\} - M_0(a) - I_a\langle t_k, 1\rangle + O_a\langle t'_{k'}, 1\rangle$$

et donc $A(z) \geq \max\{0, in_a(k) - out_a(k')\}$.

Nous prouvons que $\max\{0, in_a(k) - out_a(k')\} \leq A(z) < in_a(k)$. L'inégalité du lemme 3 page 60 est vérifiée : il existe une relation de précédence entre $\langle t_k, n(z)\rangle$ et $\langle t'_{k'}, n'(z)\rangle$, ce qui complète cette preuve. ∎

Prenons à nouveau l'exemple de l'arc $a_{AB} = (A, B)$ sur la Figure 4.1(a) page 57. On calcule $\alpha_{a_{AB}}^{\min}(1,1) = 0$ et $\alpha_{a_{AB}}^{\max}(1,1) = 0$. Comme $\alpha_{a_{AB}}^{\min}(1,1) \leq \alpha_{a_{AB}}^{\max}(1,1)$, il existe donc une relation de précédence, induite par a_{AB} entre les exécutions $\langle A_1, n\rangle$ et $\langle B_1, n'\rangle$ lorsque les valeurs $(n, n') \in (\mathbb{N} - \{0\})^2$ sont telles que

$$(n-1)i_{a_{AB}} - (n'-1)o_{a_{AB}} = 8(n-1) - 12(n'-1) = 0.$$

Lorsqu'ils sont combinés, les lemmes 4 page 61 et 5 page précédente nous permettent de décrire une condition nécessaire et suffisante de l'existence de relations de précédence induites par un buffer $a = (t, t') \in \mathcal{A}$ entre deux phases t_k et $t'_{k'}$. Vérifions maintenant qu'un ordonnancement respecte bien ces relations de précédence.

4.4 Une condition nécessaire et suffisante de validité

Nous allons maintenant utiliser les conditions d'existence précédentes pour définir le théorème 6 page ci-contre. Ce théorème définit un jeu de contraintes vérifiant l'ensemble

des relations de précédence induites par un buffer pour un ordonnancement périodique $\mathcal{S}$ dont la période normalisée est $\Omega_{\mathcal{G}}^{\mathcal{S}}$.

4.4.1 Caractérisation des contraintes périodiques

Le théorème 6 définit des *contraintes périodiques*, une condition nécessaire et suffisante pour qu'un ordonnancement périodique respecte les relations de précédence induites par un buffer.

Théorème 6 *Soit $\mathcal{G}$ un CSDFG consistant et à mémoire bornée. Pour tout ordonnancement périodique $\mathcal{S}$ de période normalisée $\Omega_{\mathcal{G}}^{\mathcal{S}}$, les relations de précédence induites par un arc $a = (t, t')$ sont vérifiées si et seulement si, pour tout couple $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ avec $\alpha_a^{\min}(k, k') \leq \alpha_a^{\max}(k, k')$,*

$$\mathcal{S}\langle {t'}_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \cdot \frac{\alpha_a^{\max}(k, k')}{N_t^{\mathcal{G}} \cdot i_a}.$$

Preuve : Supposons que l'arc a induise une relation de précédence entre deux exécutions $\langle t_k, n\rangle$ et $\langle {t'}_{k'}, n'\rangle$, alors l'ordonnancement $\mathcal{S}$ doit vérifier

$$\mathcal{S}\langle t_k, n\rangle + d(t_k) \leq \mathcal{S}\langle {t'}_{k'}, n'\rangle.$$

Comme $\mathcal{S}$ est périodique, il vérifie

$$\mathcal{S}\langle t_k, 1\rangle + (n-1)\mu_t^{\mathcal{S}} + d(t_k) \leq \mathcal{S}\langle {t'}_{k'}, 1\rangle + (n'-1)\mu_{t'}^{\mathcal{S}}.$$

Avec $\alpha = (n-1)i_a - (n'-1)o_a$, on obtient $(n-1) = \dfrac{(n'-1)o_a + \alpha}{i_a}$ et alors

$$\mathcal{S}\langle {t'}_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) - (n'-1)\mu_{t'}^{\mathcal{S}} + \frac{(n'-1)o_a + \alpha}{i_a}\mu_t^{\mathcal{S}},$$

Ce qui revient à

$$\mathcal{S}\langle {t'}_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + (n'-1)(\frac{o_a\mu_t^{\mathcal{S}}}{i_a} - \mu_{t'}^{\mathcal{S}}) + \frac{\alpha}{i_a}\mu_t^{\mathcal{S}}.$$

Maintenant, d'après le théorème 5 page 58, dans un CSDFG à mémoire bornée la relation entre les périodes de fonctionnement des tâches t et t' s'exprime par $\dfrac{N_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} = \dfrac{\mu_t^{\mathcal{S}}}{\mu_{t'}^{\mathcal{S}}}$. D'après la définition du vecteur de répétition (définition 3.2.2 page 39), $\dfrac{N_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} = \dfrac{i_a}{o_a}$ et donc $\dfrac{o_a\mu_t^{\mathcal{S}}}{i_a} = \mu_{t'}^{\mathcal{S}}$.

$$\mathcal{S}\langle {t'}_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \mu_{\mathcal{G}}^{\mathcal{S}}\frac{\alpha}{i_a}.$$

De plus, d'après la définition 5 page 58, $\mu_t^{\mathcal{S}} = \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_t^{\mathcal{G}}}$, l'inégalité devient

$$\mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{i_a \cdot N_t^{\mathcal{G}}}\alpha.$$

Enfin, le terme de droite augmente selon la valeur de α et d'après le lemme 5 page 63 la valeur $\alpha = \alpha_a^{\max}(k, k')$ est atteinte. La relation de précédence est assurée si

$$\mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \cdot \frac{\alpha_a^{\max}(k, k')}{N_t^{\mathcal{G}} \cdot i_a}. \tag{4.1}$$

Réciproquement, faisons l'hypothèse que l'ordonnancement périodique $\mathcal{S}$ vérifie les inégalités exprimées par le lemme. Soit $a = (t, t') \in \mathcal{A}$ et le couple d'entiers (n, n') et $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ tels qu'il existe une relation de précédence entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$. Vérifions que cette relation de précédence est bien vérifiée par l'équation

$$\mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \frac{\alpha_a^{\max}(k, k')}{N_t^{\mathcal{G}} \cdot i_a}. \tag{4.2}$$

D'après le lemme 4 page 61, $\alpha = (n-1)\times i_a - (n'-1)\times o_a$ est bornée par $\alpha_a^{\max}(k, k')$. Donc lorsque la contrainte (4.2) est vérifiée, la contrainte (4.1) l'est aussi. D'après l'argument inverse, on prouve alors que $\mathcal{S}\langle t_k, n\rangle + d(t_k) \leq \mathcal{S}\langle t'_{k'}, n'\rangle$ pour toute valeur $\alpha_a^{\min}(k, k') \leq \alpha(p, p') \leq \alpha_a^{\max}(k, k')$. ∎

Le théorème 6 page précédente définit une condition nécessaire et suffisante de validité d'un ordonnancement périodique. Comme cette condition peut être vérifiée dans un temps polynomial en fonction de la taille du problème, la recherche d'un ordonnancement périodique de CSDFG appartient à NP.

Afin d'entrevoir certaines possibilités d'utilisation de ce théorème, dans la section suivante nous l'appliquons à la vérification d'un ordonnancement périodique.

4.4.2 Validité d'un ordonnancement périodique

Notons maintenant $\mathcal{G}$ le graphe de la Figure 4.1(a) page 57 et $\mathcal{S}$ l'ordonnancement visible sur la Figure 4.1(b) page 57. Utilisons le théorème 6 page précédente pour vérifier que l'ordonnancement $\mathcal{S}$ respecte bien les relations de précédence induites par les buffers du graphe $\mathcal{G}$.

Pour chaque buffer, on doit vérifier chacune des contraintes définies par le théorème.

Prenons l'exemple du buffer a_{AB} reliant les tâches A et B. Pour tout couple $(k, k') \in \{1, \cdots, \varphi(A)\} \times \{1, \cdots, \varphi(B)\}$ tel que $\alpha_{a_{AB}}^{\min}(k, k') \leq \alpha_{a_{AB}}^{\max}(k, k')$ on doit vérifier que l'in-

équation suivante est vérifiée :

$$\mathcal{S}\langle B_{k'},1\rangle - \mathcal{S}\langle A_k,1\rangle \quad \geq d(A_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \cdot \frac{\alpha_{a_{AB}}^{\max}(k,k')}{i_{a_{AB}} \cdot N_A^{\mathcal{G}}}.$$

Pour le premier couple $(k,k') = (1,1)$, on calcule $\alpha_{a_{AB}}^{\min}(1,1) = 0$ et $\alpha_{a_{AB}}^{\max}(1,1) = 0$. Comme $\alpha_{a_{AB}}^{\min}(1,1) \leq \alpha_{a_{AB}}^{\max}(1,1)$ on doit donc vérifier

$$\mathcal{S}\langle B_1,1\rangle - \mathcal{S}\langle A_1,1\rangle \quad \geq d(A_1) + \Omega_{\mathcal{G}}^{\mathcal{S}} \cdot \frac{\alpha_{a_{AB}}^{\max}(1,1)}{i_{a_{AB}} \cdot N_A^{\mathcal{G}}}. \tag{4.3}$$

On rappelle que la période normalisée de l'ordonnancement $\mathcal{S}$ est $\Omega_{\mathcal{G}}^{\mathcal{S}} = 24$ et que le facteur de répétition de la tâche A est $N_A^{\mathcal{G}} = 3$. Par substitution,

$$3 - 0 \quad \geq 3 + 24 \times \frac{0}{8\times 3}.$$

Cette première contrainte est vérifiée.

La Figure 4.3 représente les premières relations de précédence induites par le buffer a_{AB}. Toutes les relations de précédence pouvant exister entre les exécutions des phases A_1 et B_1 sont alors vérifiées par la contrainte (4.3).

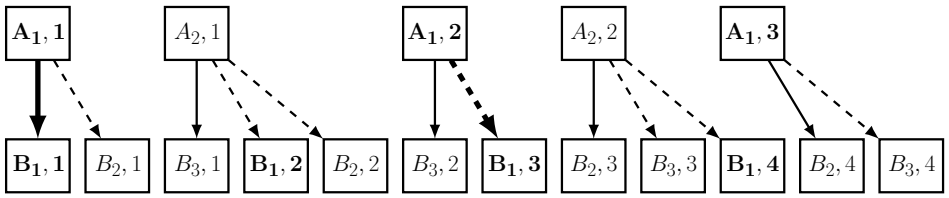


FIGURE 4.3 – Une représentation graphique des premières relations de précédence intervenant dans le CSDFG de la Figure 4.1(a) page 57. Les relations en surbrillance sont celle entre les phases A_1 et B_1.

Prenons maintenant le second couple $(k,k') = (2,1)$, on calcule $\alpha_{a_{AB}}^{\min}(2,1) = 12$ et $\alpha_{a_{AB}}^{\max}(2,1) = 6$. Cette fois-ci $\alpha_{a_{AB}}^{\min}(2,1) \geq \alpha_{a_{AB}}^{\max}(2,1)$, il n'est donc pas nécessaire de vérifier cette inégalité, car il n'existe aucune relation de précédence (stricte) entre ces phases.

De cette manière, nous vérifions que l'ensemble des contraintes est vérifié pour ce buffer a_{AB} et qu'il en est de même pour les deux autres buffers. L'ordonnancement $\mathcal{S}$ respecte alors toutes les relations de précédences induites par les buffers du graphe $\mathcal{G}$; cet ordonnancement est un ordonnancement périodique valide pour le CSDFG $\mathcal{G}$.

Conclusion

Dans ce chapitre, nous fournissons la définition d'un ordonnancement périodique pour un CSDFG puis nous caractérisons un ensemble de contraintes nécessaires et suffisantes pour que l'ordonnancement périodique d'un CSDFG soit un ordonnancement valide.

Ce résultat est fondamental pour la suite de notre étude : dans le chapitre 6, nous appliquerons ces contraintes à l'évaluation du débit et au dimensionnement mémoire d'un CSDFG.

Néanmoins, rappelons que Marchetti et Munier-Kordon [63] montrent l'existence de SDFG vivants, mais pour lesquels aucun ordonnancement périodique n'est valide. Notre méthode généralise ces travaux, et souffre des mêmes limitations. Nous souhaitons alors améliorer ce résultat, sans pour autant atteindre la complexité d'un ordonnancement *au plus tôt*.

Dans le chapitre suivant, nous nous intéressons à un ensemble d'ordonnancements bien plus général, les ordonnancements K-périodiques.

Chapitre 5

Ordonnancement K-périodique

Introduction

Les techniques d'analyse dataflows que nous connaissons se classent aujourd'hui dans deux familles bien distinctes. Tout d'abord, on trouve des techniques basées sur l'ordonnançabilité *au plus tôt*. Elles sont souvent exactes, mais la complexité de leur temps de calcul est exponentielle. Il existe ensuite d'autres méthodes – généralement approchées – qui limitent leur espace de solutions aux ordonnancements *périodiques*. Dans le chapitre 4, nous caractérisons par exemple l'ordonnancement périodique d'un CSDFG au travers de contraintes linéaires. Nous les appliquons ensuite à la résolution de différentes problématiques d'analyse, mais ces contributions seront alors limitées aux seuls ordonnancements périodiques.

Le langage de programmation ΣC nous amène à rencontrer des CSDFG de taille très importante. Ces instances ne sont pas supportées par des techniques *au plus tôt*, dont la complexité est exponentielle. D'autre part, pour de nombreuses applications embarquées, une solution périodique n'est pas suffisante. Marchetti et Munier-Kordon [63] montrent qu'il existe des cas de CSDFG vivants, mais pour lesquelles il n'existe pas de solution périodique (le SDFG de la Figure 5.1 en est l'un d'entre eux). Au cours de notre étude, nous avons aussi rencontré des cas industriels pour lesquels les techniques périodiques ne conviennent plus.

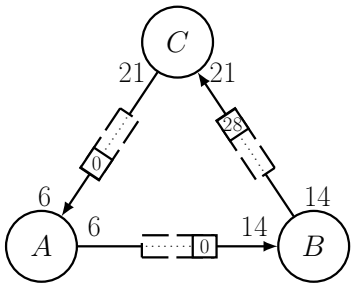


FIGURE 5.1 – Ce SDFG est vivant, il admet notamment la séquence d'exécution $C, A, A, A, B, C, A, A, A, A, B, B$, mais il n'admet pas d'ordonnancement périodique.

Pour ces instances, on ne dispose d'aucune méthode d'analyse efficace. Nous proposons donc un compromis entre les ordonnancements *au plus tôt*, et les ordonnancements périodiques : les ordonnancements K-périodiques [25].

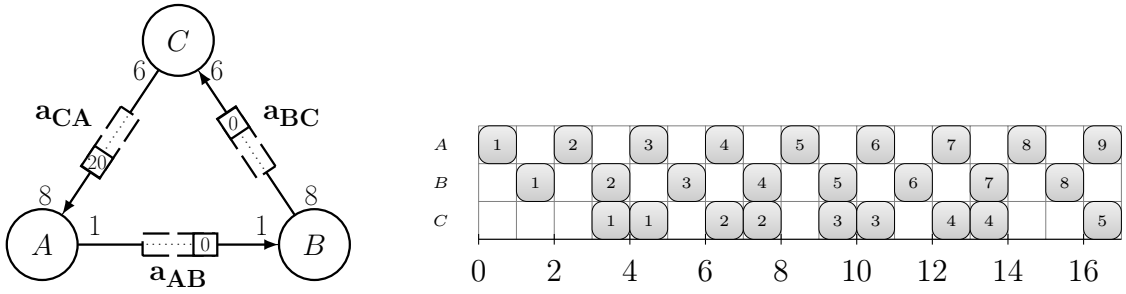
Un ordonnancement K-périodique est un ordonnancement cyclique tel que l'ordonnancement des exécutions d'une tâche t se construit en répétant périodiquement l'ordonnancement de ses K_t premières exécutions. K_t est le *facteur de périodicité* de cette tâche. Un ordonnancement périodique est donc un cas particulier des ordonnancements K-périodiques où $K_t = 1$ pour toutes ses tâches.

Dans ce chapitre, nous définissons un ensemble de contraintes linéaires vérifiant la validité d'un ordonnancement K-périodique et nous étudions ensuite l'influence que peuvent avoir les facteurs de périodicité sur les performances d'un ordonnancement.

Dans la première section de ce chapitre, nous présentons l'ordonnancement K-périodique d'un SDFG. Nous prouvons ensuite une condition nécessaire et suffisante d'existence d'une relation de précédence entre deux exécutions périodiques ; cette condition nous permet alors de fournir une méthode d'ordonnancement K-périodique d'un SDFG minimisant sa période. Afin d'envisager l'intégration de cette approche dans des outils d'analyse dataflows, nous étudions l'influence que peut avoir la K-périodicité sur les performances d'un ordonnancement. Pour finir, nous envisageons l'application des ordonnancements K-périodiques aux CSDFG.

5.1 Ordonnancement K-périodique d'un SDFG

D'après la définition 3 page 37, l'ordonnancement périodique d'un SDFG se caractérise par la première date d'exécution et la période de chacune des tâches du graphe. Pour un ordonnancement K-périodique, le nombre de dates d'exécution à définir pour une tâche peut alors varier en fonction de son facteur de périodicité.



(a) Exemple de SDFG

(b) Ordonnancement K-périodique

FIGURE 5.2 – L'ordonnancement K-périodique (b) est un ordonnancement valide pour le SDFG (a) et dont le vecteur de périodicité est $K^{\mathcal{G}} = [1, 1, 2]$. Les périodes des différentes tâches sont $\mu_A^{\mathcal{S}} = \mu_B^{\mathcal{S}} = 2$, $\mu_C^{\mathcal{S}} = 3$.

Définition 7 *Considérons le SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Un ordonnancement $\mathcal{S}$ est K-périodique s'il existe un vecteur de périodicité $K^{\mathcal{G}}$ tel que l'ordonnancement $\mathcal{S}$ vérifie pour toute tâche $t \in \mathcal{T}$ et pour toute valeur $n = k + p \times K_t^{\mathcal{G}}$ avec $p \in \mathbb{N}$ et $k \in \{1, \cdots, K_t^{\mathcal{G}}\}$*

$$\mathcal{S}\langle t, n\rangle = \mathcal{S}\langle t, k\rangle + \mu_t^{\mathcal{S}} \cdot p.$$

On note $\mu_t^{\mathcal{S}}$ la période de la tâche t et $K_t^{\mathcal{G}}$ son facteur de périodicité.

L'ordonnancement périodique est un cas particulier des ordonnancements K-périodiques où $\forall t \in \mathcal{T}$, $K_t^{\mathcal{G}} = 1$. Nous le désignons aussi comme l'ordonnancement 1-périodique.

La Figure 5.2 page précédente nous montre un exemple d'ordonnancement K-périodique. Les ordonnancements des tâches A et B sont périodiques, mais la tâche C respecte un motif d'exécution 2-périodique.

5.1.1 Fréquence de fonctionnement

La fréquence de fonctionnement d'une tâche pour un ordonnancement K-périodique peut alors s'exprimer en fonction de sa période et de son facteur de périodicité. D'après la définition de la fréquence de fonctionnement d'une tâche

$$Th_t^{\mathcal{S}} = \lim_{n\to\infty} \frac{n}{\mathcal{S}\langle t, n\rangle}.$$

Par substitution on trouve

$$Th_t^{\mathcal{S}} = \lim_{p\to\infty} \frac{k + p \cdot K_t^{\mathcal{G}}}{\mathcal{S}\langle t, k\rangle + \mu_t^{\mathcal{S}} \cdot p}.$$

Le débit d'une tâche $t \in \mathcal{T}$ est alors

$$Th_t^{\mathcal{S}} = \frac{K_t^{\mathcal{G}}}{\mu_t^{\mathcal{S}}}.$$

D'après le théorème 5 page 58 , dans un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, pour chaque couple de tâches $(t, t') \in \mathcal{T}^2$,

$$\frac{Th_t^{\mathcal{S}}}{N_t^{\mathcal{G}}} = \frac{Th_{t'}^{\mathcal{S}}}{N_{t'}^{\mathcal{G}}}.$$

On déduit alors la définition d'une période normalisée pour un ordonnancement K-périodique $\mathcal{S}$.

Définition 8 *Lorsqu'un SDFG consistant est à mémoire bornée, il existe pour tout ordonnancement valide un équilibre entre la fréquence de fonctionnement de ses tâches. Cet équilibre implique l'existence d'une période normalisée*

$$\Omega_{\mathcal{G}}^{\mathcal{S}} = \frac{N_t^{\mathcal{G}}}{Th_t^{\mathcal{S}}} = \frac{N_t^{\mathcal{G}} \cdot \mu_t^{\mathcal{S}}}{K_t^{\mathcal{G}}}.$$

Pour l'exemple de la Figure 5.2 page précédente, on calcule son vecteur de répétition minimal

$$N^{\mathcal{G}} = [3, 3, 4].$$

Alors la période normalisée du graphe se calcule en choisissant l'une de ses tâches (la tâche A par exemple) :

$$\Omega_{\mathcal{G}}^{\mathcal{S}} = \frac{N_A^{\mathcal{G}} \cdot \mu_A^{\mathcal{S}}}{K_A^{\mathcal{G}}} = \frac{3 \cdot 2}{1} = 6$$

5.1.2 Définition d'une relation de précédence

Dans un SDFG, une relation de précédence est une contrainte induite par un buffer entre deux exécutions. Munier-Kordon [66] rappelle que l'ordonnancement d'un SDFG est *valide* si et seulement s'il respecte l'ensemble des relations de précédence induites par ce SDFG. Nous souhaitons alors caractériser l'existence des relations de précédence d'un SDFG afin de vérifier par la suite qu'un ordonnancement K-périodique respecte ces relations de précédence.

Existence d'une relation de précédence entre deux exécutions

Munier-Kordon [66] définit l'existence d'une relation de précédence entre deux exécutions au travers d'un ensemble de trois conditions :

Définition 9 *Munier-Kordon [66] Considérons le SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Soit $a = (t, t') \in \mathcal{A}$ un buffer et (n, n') un couple d'entiers strictement positifs. Il existe une relation de précédence stricte entre les exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si :*

- ***Condition 1*** *: $\langle t', n'\rangle$ peut être exécutée après $\langle t, n\rangle$.*
- ***Condition 2*** *: $\langle t', n'\rangle$ ne peut s'exécuter avant $\langle t, n\rangle$.*
- ***Condition 3*** *: $\langle t', n' - 1\rangle$ peut s'exécuter avant $\langle t, n\rangle$.*

Rappelons aussi que pour respecter une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$, un ordonnancement $\mathcal{S}$ doit alors vérifier

$$\mathcal{S}\langle t', n'\rangle \geq \mathcal{S}\langle t, n\rangle + d(t). \tag{5.1}$$

Si l'on prend l'exemple du buffer a_{BC} de la Figure 5.2 page 71, et de deux exécutions $\langle B, 2\rangle$ et $\langle C, 2\rangle$,

- $\langle C, 2\rangle$ peut s'exécuter après $\langle B, 2\rangle$,
- $\langle C, 2\rangle$ ne peut s'exécuter avant $\langle B, 2\rangle$,
- et $\langle C, 1\rangle$ peut s'exécuter avant $\langle B, 2\rangle$.

Les trois conditions sont vérifiées, il existe une relation de précédence entre ces deux exécutions.

Formalisation de l'existence d'une relation de précédence

Une condition nécessaire et suffisante d'existence d'une relation de précédence entre deux exécutions a ensuite été formalisée par Munier-Kordon [66] :

Lemme 6 *Munier-Kordon [66] Considérons un SDFG noté $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Un buffer $a = (t, t') \in \mathcal{A}$ induit alors une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si :*

$$in_a > M_0(a) + in_a \times n - out_a \times n' \geq \max\{in_a - out_a, 0\}.$$

Si l'on considère à nouveau les deux exécutions $\langle B, 2\rangle$ et $\langle C, 2\rangle$ pour le buffer a_{BC}, d'après le lemme 6, comme l'inégalité

$$8 > 0 + 8 \times 2 - 6 \times 2 \geq \max\{8 - 6, 0\}$$

est valide ; il existe bien une relation de précédence entre ces deux exécutions.

La condition (5.1) page précédente vérifie qu'une exécution précède une autre exécution. Pour qu'un ordonnancement soit valide, cette condition doit donc toujours être respectée lorsqu'il existe une relation de précédence entre ces deux exécutions (elle peut aussi se vérifier dans les autres cas, mais ce n'est pas nécessaire). Nous proposons maintenant une contrainte entre deux exécutions d'un ordonnancement K-périodique. C'est une condition nécessaire et suffisante pour qu'un ordonnancement respecte la relation de précédence entre ces exécutions (lorsqu'elle existe).

5.2 Caractérisation d'une contrainte de précédence

Nous prouvons pour commencer une condition nécessaire d'existence d'une relation de précédence. Nous définissons ensuite une contrainte garantissant qu'un ordonnancement K-périodique respecte une relation de précédence entre deux exécutions uniquement si elle existe.

5.2.1 Condition d'existence d'une relation de précédence

Nous définissons une condition nécessaire à l'existence d'une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ dont les indices seront de la forme $n = k + K_t^{\mathcal{G}} \cdot p$ et $n' = k' + K_{t'}^{\mathcal{G}} \cdot p'$ avec $k \in \{1, \ldots, K_t^{\mathcal{G}}\}$ et $k' \in \{1, \ldots, K_{t'}^{\mathcal{G}}\}$, deux entiers fixés.

Afin de simplifier la lecture de ce chapitre, nous utiliserons les notations suivantes :

$$\begin{aligned}
\alpha_a(k,k') &= \frac{in_a \cdot k - out_a \cdot k'}{\gcd_a} \\
\pi_a(p,p') &= \frac{in_a \cdot p \cdot K_t^{\mathcal{G}} - out_a \cdot p' \cdot K_{t'}^{\mathcal{G}}}{\gcd_a^k} \\
\pi_a^{max}(k,k') &= \left\lfloor \frac{in_a - \gcd_a - M_0(a) - \alpha_a(k,k') \cdot \gcd_a}{\gcd_a^k} \right\rfloor \\
\pi_a^{min}(k,k') &= \left\lceil \frac{\max\{in_a - out_a, 0\} - M_0(a) - \alpha_a(k,k') \cdot \gcd_a}{\gcd_a^k} \right\rceil
\end{aligned}$$

On note également $\gcd_a$ (*resp.* $\gcd_a^k$) le plus grand diviseur commun entre in_a et out_a (*resp.* $K_t^{\mathcal{G}} \cdot in_a$ et $K_{t'}^{\mathcal{G}} \cdot out_a$).

Lemme 7 *Dans un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, si un buffer $a = (t, t') \in \mathcal{A}$ induit une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ avec $n = k + p \times K_t^{\mathcal{G}}$, $n' = k' + p' \times K_{t'}^{\mathcal{G}}$, $k \in \{1, \ldots, K_t^{\mathcal{G}}\}$ et $k' \in \{1, \ldots, K_{t'}^{\mathcal{G}}\}$ alors l'inéquation suivante est vérifiée*

$$\pi_a^{min}(k,k') \leq \pi_a(p,p') \leq \pi_a^{max}(k,k').$$

Preuve : Nous savons d'ores et déjà que

$$\begin{aligned}
in_a \cdot n - out_a \cdot n' &= in_a \cdot (p \cdot K_t^{\mathcal{G}} + k) - out_a \cdot (p' \cdot K_{t'}^{\mathcal{G}} + k') \\
&= in_a(p \cdot K_t^{\mathcal{G}}) - out_a(p' \cdot K_{t'}^{\mathcal{G}}) + in_a \cdot k - out_a \cdot k'.
\end{aligned}$$

D'après la définition de $\alpha_a(k,k')$ et $\pi_a(p,p')$, l'inégalité suivante est déduite :

$$in_a \cdot n - out_a \cdot n' = \pi_a(p,p') \cdot \gcd_a^k + \alpha_a(k,k') \cdot \gcd_a.$$

Prouvons que

$$\pi_a^{min}(k,k') \leq \pi_a(p,p') \leq \pi_a^{max}(k,k').$$

D'après le lemme 6 page précédente comme a induit une relation de précédence entre $\langle t, n\rangle$ et $\langle t', n'\rangle$ alors

$$in_a > M_0(a) + in_a \cdot n - out_a \cdot n' \geq \max\{in_a - out_a, 0\}.$$

Donc,

$$in_a > M_0(a) + \pi_a(p,p') \cdot \gcd_a^k + \alpha_a(k,k') \cdot \gcd_a \geq \max\{in_a - out_a, 0\}.$$

Comme $\pi_a(p,p')$ est une valeur entière, la partie droite de l'inégalité devient

$$\left\lceil \frac{\max\{in_a - out_a, 0\} - M_0(a) - \alpha_a(k,k') \cdot \gcd_a}{\gcd_a^k} \right\rceil \quad \leq \quad \pi_a(p,p'),$$

et donc $\pi_a(p,p') \geq \pi_a^{min}(k,k')$.

Comme tous les termes de l'inégalité de gauche sont divisibles par $\gcd_a$ on obtient,

$$in_a - \gcd_a \geq M_0(a) + \pi_a(p,p') \cdot \gcd_a^k + \alpha_a(k,k') \cdot \gcd_a,$$

alors

$$\left\lfloor \frac{in_a - \gcd_a - M_0(a) - \alpha_a(k,k') \cdot \gcd_a}{\gcd_a^k} \right\rfloor \quad \geq \quad \pi_a(p,p')$$

et $\pi_a(p,p') \leq \pi_a^{max}(k,k')$; ce qui prouve le lemme. ∎

Considérons maintenant les exécutions $\langle t, k + p \times K_t^{\mathcal{G}} \rangle$ et $\langle t', k' + p' \times K_{t'}^{\mathcal{G}} \rangle$. Le lemme 7 page précédente nous permet alors d'affirmer que si

$$\pi_a^{min}(k,k') > \pi_a^{max}(k,k')$$

alors il n'existe aucune relation de précédence entre ces exécutions, quelles que soient les valeurs de p et p'.

5.2.2 Caractérisation d'une contrainte de précédence

Le théorème qui va suivre est une condition nécessaire et suffisante pour qu'un ordonnancement K-périodique de périodicité $K^{\mathcal{G}}$ respecte les relations de précédence entre deux exécutions de la forme $\langle t, k + K_t^{\mathcal{G}} \cdot p \rangle$ et $\langle t', k' + K_{t'}^{\mathcal{G}} \cdot p' \rangle$.

Théorème 7 *Un ordonnancement K-périodique $\mathcal{S}$ de périodicité $K^{\mathcal{G}}$ respecte les relations de précédence induites par un buffer $a = (t,t') \in \mathcal{A}$ entre deux exécutions $\langle t, k + K_t^{\mathcal{G}} \cdot p \rangle$ et $\langle t', k' + K_{t'}^{\mathcal{G}} \cdot p' \rangle$ si et seulement si*

$$\mathcal{S}\langle t', k' \rangle - \mathcal{S}\langle t, k \rangle \quad \geq \quad d(t) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{K_t^{\mathcal{G}}} \frac{\pi_a(p,p') \cdot gcd_a^k}{in_a}. \tag{5.2}$$

PREUVE : Considérons un buffer $b = (t,t') \in \mathcal{A}$ qui induit une relation de précédence entre les exécutions $\langle t, n \rangle$ et $\langle t', n' \rangle$, alors

$$\mathcal{S}\langle t', n' \rangle \geq \mathcal{S}\langle t, n \rangle + d(t).$$

D'après la définition 7 page 71, $\mathcal{S}\langle t,n\rangle = \mathcal{S}\langle t,k\rangle + \mu_t^{\mathcal{S}} \cdot p$ et $\mathcal{S}\langle t',n'\rangle = \mathcal{S}\langle t',k'\rangle + \mu_{t'}^{\mathcal{S}} \cdot p'$. La contrainte de précédence devient

$$\mathcal{S}\langle t',k'\rangle - \mathcal{S}\langle t,k\rangle \;\geq\; d(t) + \mu_t^{\mathcal{S}} \cdot p - \mu_{t'}^{\mathcal{S}} \cdot p'. \tag{5.3}$$

D'après la définition de $\pi_a(p,p')$,

$$\begin{aligned} in_a \cdot K_t^{\mathcal{G}} \cdot p - out_a \cdot K_{t'}^{\mathcal{G}} \cdot p' &= \pi_a(p,p') \cdot \gcd_a^k \\ p' &= \frac{1}{out_a \cdot K_{t'}^{\mathcal{G}}}(in_a \cdot K_t^{\mathcal{G}} \cdot p - \pi_a(p,p') \cdot \gcd_a^k). \end{aligned}$$

Alors, en substituant p' dans l'équation (5.3),

$$\begin{aligned} \mathcal{S}\langle t',k'\rangle - \mathcal{S}\langle t,k\rangle &\geq d(t) + \mu_t^{\mathcal{S}} \cdot p - \frac{\mu_{t'}^{\mathcal{S}}}{out_a \cdot K_{t'}^{\mathcal{G}}}(in_a \cdot K_t^{\mathcal{G}} \cdot p - \pi_a(p,p') \cdot \gcd_a^k) \\ \mathcal{S}\langle t',k'\rangle - \mathcal{S}\langle t,k\rangle &\geq d(t) + p(\mu_t^{\mathcal{S}} - \mu_{t'}^{\mathcal{S}} \cdot \frac{in_a \cdot K_t^{\mathcal{G}}}{out_a \cdot K_{t'}^{\mathcal{G}}}) + \frac{\mu_{t'}^{\mathcal{S}}}{out_a \cdot K_{t'}^{\mathcal{G}}}\pi_a(p,p') \cdot \gcd_a^k. \end{aligned}$$

D'après la définition du vecteur de répétition (définition 3.2.2 page 39), on note

$$\forall a = (t,t') \in \mathcal{A} \quad \frac{N_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} = \frac{in_a}{out_a}.$$

Sous l'hypothèse d'un buffer à mémoire bornée et d'après la définition de la période normalisée d'un ordonnancement K-périodique (définition 8 page 72),

$$\mu_t^{\mathcal{S}} = \frac{\Omega_{\mathcal{G}}^{\mathcal{S}} \cdot K_t^{\mathcal{G}}}{N_t^{\mathcal{G}}}.$$

On trouve alors

$$\mu_t^{\mathcal{S}} - \mu_{t'}^{\mathcal{S}} \frac{in_a \cdot K_t^{\mathcal{G}}}{out_a \cdot K_{t'}^{\mathcal{G}}} = 0.$$

On obtient donc l'inéquation

$$\mathcal{S}\langle t',k'\rangle - \mathcal{S}\langle t,k\rangle \;\geq\; d(t) + \frac{\mu_t^{\mathcal{S}}}{K_t^{\mathcal{G}}} \frac{\pi_a(p,p') \cdot \gcd_a^k}{in_a}. \tag{5.4}$$

Enfin, d'après la définition de la période normalisée

$$\mathcal{S}\langle t',k'\rangle - \mathcal{S}\langle t,k\rangle \;\geq\; d(t) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_t^{\mathcal{G}}} \frac{\pi_a(p,p') \cdot \gcd_a^k}{in_a}.$$

Réciproquement, supposons que l'inégalité (5.4) soit vérifiée pour un ordonnancement K-périodique $\mathcal{S}$, alors on peut prouver par l'argument inverse que $\mathcal{S}$ vérifie aussi l'inégalité (5.3) tout comme les relations de précédence entre les exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$. ∎

Pour l'exemple du SDFG de la Figure 5.2 page 71, nous savons qu'il existe une relation de précédence entre les exécutions $\langle B, 5\rangle$ et $\langle C, 6\rangle$. Dans l'ordonnancement proposé, la condition

$$\mathcal{S}\langle C, 6\rangle \geq \mathcal{S}\langle B, 5\rangle + d(B)$$

est donc respectée.

D'après le théorème 7 page 76, une autre contrainte nous assurant qu'un ordonnancement K-périodique vérifie cette relation de précédence se note

$$\begin{aligned} \mathcal{S}\langle C, 2\rangle - \mathcal{S}\langle B, 1\rangle &\geq d(B) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_B^{\mathcal{G}}} \frac{\pi_a(4,2) \cdot \gcd_{a_{BC}}^k}{in_{a_{BC}}} \\ 4 - 1 &\geq 1 + 2 \end{aligned}$$

Cette contrainte est aussi vérifiée.

5.3 Validité d'un ordonnancement K-périodique

Nous allons maintenant décrire un jeu de contrainte vérifiant la validité d'un l'ordonnancement K-périodique pour un SDFG.

5.3.1 Condition suffisante d'existence d'une relation de précédence

La contrainte de précédence définie précédemment ne s'applique qu'à un couple d'exécutions particulières. Afin de généraliser cette contrainte à une infinité d'exécutions, nous allons définir le lemme 8, une condition suffisante d'existence d'une relation de précédence. C'est la réciproque du lemme 7 page 75.

Lemme 8 *Soit un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ et le buffers $a = (t, t') \in \mathcal{A}$. Considérons les deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ avec $n = k + K_t^{\mathcal{G}} \cdot p$, $n' = k' + K_{t'}^{\mathcal{G}} \cdot p'$, $k \in \{1, \ldots, K_t^{\mathcal{G}}\}$, $k' \in \{1, \ldots, K_{t'}^{\mathcal{G}}\}$, $p \in \mathbb{N}$ et $p' \in \mathbb{N}$. Si $\pi_a(p, p')$ vérifie*

$$\pi_a^{min}(k, k') \leq \pi_a(p, p') \leq \pi_a^{max}(k, k')$$

alors il existe une relation de précédence entre les exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$.

PREUVE : D'après Bezout, il existe $(x, y) \in \mathbb{Z}^{\star 2}$ tels que

$$x \cdot K_t^{\mathcal{G}} \cdot in_a + y \cdot K_{t'}^{\mathcal{G}} \cdot out_a = \gcd_a^k.$$

Alors,

$$\pi_a(p, p') \cdot x \cdot K_t^{\mathcal{G}} \cdot in_a + \pi_a(p, p') \cdot y \cdot K_{t'}^{\mathcal{G}} \cdot out_a = \pi_a(p, p') \cdot \gcd_a^k. \tag{5.5}$$

Notons les valeurs entières z, p et p' telles que

$$\begin{aligned} p &= \pi_a(p, p') \cdot x + z \cdot K_{t'}^{\mathcal{G}} \cdot out_a \\ \text{et} \quad p' &= -\pi_a(p, p') \cdot y + z \cdot K_t^{\mathcal{G}} \cdot in_a. \end{aligned}$$

Considérons aussi n et n' tels que $n = k + K_t^{\mathcal{G}} \cdot p$ et $n' = k' + K_{t'}^{\mathcal{G}} \cdot p'$. Pour z suffisamment grand ($z \geq z_0$), $p \geq 1$ et $p' \geq 1$ et alors n et n' sont tous deux supérieurs à 1. Alors,

$$\begin{aligned} in_a \cdot n - out_a \cdot n' &= in_a(k + K_t^{\mathcal{G}} \cdot p) - out_a(k' + K_{t'}^{\mathcal{G}} \cdot p') \\ &= k \cdot in_a + K_t^{\mathcal{G}} \cdot in_a \cdot p - k' \cdot out_a - K_{t'}^{\mathcal{G}} \cdot out_a \cdot p' \\ &= (k \cdot in_a - k' \cdot out_a) + (K_t^{\mathcal{G}} \cdot in_a \cdot \pi_a(p, p') \cdot x + K_{t'}^{\mathcal{G}} \cdot out_a \cdot \pi_a(p, p') \cdot y). \end{aligned}$$

D'après l'inégalité (5.5), on obtient

$$in_a \cdot n - out_a \cdot n' = k \cdot in_a - k' \cdot out_a + \pi_a(p, p') \cdot \gcd_a^k$$

et donc, d'après la définition de $\alpha_a(k, k')$,

$$in_a \cdot n - out_a \cdot n' = \alpha_a(k, k') \cdot \gcd_a + \pi_a(p, p') \cdot \gcd_a^k.$$

D'après l'hypothèse de départ $\pi_a^{min}(k, k') \leq \pi_a(p, p') \leq \pi_a^{max}(k, k')$, alors

$$in_a \cdot n - out_a \cdot n' \leq \pi_a^{max}(k, k') \cdot \gcd_a^k + \alpha_a(k, k') \cdot \gcd_a$$

et

$$\begin{aligned} \pi_a^{max}(k, k') &\leq \frac{in_a - \gcd_a - M_0(a) - \alpha_a(k, k') \cdot \gcd_a}{\gcd_a^k} \\ in_a \cdot n - out_a \cdot n' &\leq in_a - \gcd_a - M_0(a) \\ &< in_a - M_0(a). \end{aligned}$$

De la même manière,

$$\begin{aligned} in_a \cdot n - out_a \cdot n' &\geq \pi_a^{min}(k,k') \cdot \gcd_a^k + \alpha_a(k,k') \cdot \gcd_a \\ &\geq \max\{in_a - out_a, 0\} - M_0(a). \end{aligned}$$

Ainsi, d'après le lemme 6 page 74 il existe bien une relation de précédence entre $\langle t, n\rangle$ et $\langle t', n'\rangle$; ce qui prouve le lemme. ∎

Ce lemme nous montre que si $\pi_a^{min}(k,k') \leq \pi_a^{max}(k,k')$, alors il existe toujours une valeur $\pi_a(p,p') \in \{\pi_a^{min}(k,k'), \ldots, \pi_a^{max}(k,k')\}$ ainsi qu'une relation de précédence entre deux exécutions $\langle t, k + K_t^{\mathcal{G}} \cdot p\rangle$ et $\langle t', k' + K_{t'}^{\mathcal{G}} \cdot p'\rangle$. Nous sommes en mesure de définir l'ensemble $\mathcal{U}(a)$ des couples (k,k') pour lesquels il existe une relation de précédence entre deux exécutions $\langle t, k + K_t^{\mathcal{G}} \cdot p\rangle$ et $\langle t', k' + K_{t'}^{\mathcal{G}} \cdot p'\rangle$:

$$\mathcal{U}(a) = \{(k,k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\},\ \ \pi_a^{\min}(k,k') \leq \pi_a^{\max}(k,k')\}.$$

5.3.2 Une condition nécessaire et suffisante de validité

Le théorème 7 page 76 était une condition nécessaire et suffisante pour qu'un ordonnancement K-périodique respecte les relations de précédence entre deux exécutions particulières $\langle t, k + K_t^{\mathcal{G}} \cdot p\rangle$ et $\langle t', k' + K_{t'}^{\mathcal{G}} \cdot p'\rangle$. Le théorème suivant, s'appuyant sur les lemmes 7 page 75 et 8 page 78, va généraliser ce théorème à toutes les valeurs p, p'.

Théorème 8 *Un ordonnancement K-périodique $\mathcal{S}$ pour un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ est un ordonnancement valide si et seulement s'il vérifie pour tout arc $a = (t,t') \in \mathcal{A}$:*

$$\forall (k,k') \in \mathcal{U}(a)\ \ \mathcal{S}\langle t', k'\rangle - \mathcal{S}\langle t, k\rangle \geq d(t) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_t^{\mathcal{G}}} \frac{\pi_a^{max}(k,k') \cdot gcd_a^k}{in_a}$$

Preuve : Supposons qu'un ordonnancement $\mathcal{S}$ vérifie ce jeu d'équation. Sachant que $\mathcal{U}(a)$ est l'ensemble des couples (k,k') tel que $\pi_a^{\min}(k,k') \leq \pi_a^{\max}(k,k')$, alors d'après le lemme 8 page 78, l'ensemble des relations de précédence induite par les buffers du graphe sont vérifiées, l'ordonnancement $\mathcal{S}$ est donc un ordonnancement K-périodique valide.

Inversement, si l'ordonnancement $\mathcal{S}$ est un ordonnancement K-périodique valide alors l'ensemble des relations de précédence est respecté. Ainsi, d'après le théorème 7 page 76, pour tout couple d'exécutions $\langle t, k + K_t^{\mathcal{G}} \cdot p\rangle$ et $\langle t', k' + K_{t'}^{\mathcal{G}} \cdot p'\rangle$, l'ordonnancement $\mathcal{S}$ vérifie

$$\mathcal{S}\langle t', k'\rangle - \mathcal{S}\langle t, k\rangle \ \ \geq \ \ d(t) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{K_t^{\mathcal{G}}} \frac{\pi_a(p,p') \cdot \gcd_a^k}{in_a}.$$

Et comme d'après le lemme 7 page 75, la valeur $\pi_a(p,p') = \pi_a^{max}(k,k')$ est toujours atteinte par un couple, alors l'ordonnancement $\mathcal{S}$ respecte ces contraintes. Ce qui prouve le théorème. ∎

Comme la condition de validité d'un ordonnancement K-périodique pour un vecteur de périodicité fixé se calcule dans un temps polynomial, on peut conclure que ce problème d'ordonnancement K-périodique appartient à NP.

Nous allons maintenant étudier l'évaluation du débit maximal d'une application à l'aide des ordonnancements K-périodiques.

5.4 Évaluation du débit maximal

Les contraintes définies par le théorème 8 page ci-contre peuvent être appliquées à l'évaluation du débit maximal d'un ordonnancement K-périodique dont le vecteur de périodicité est fixé. Pour cela nous utilisons la programmation linéaire.

Les contraintes qui furent définies dans la section précédente sont des contraintes linéaires. Lorsque les dates d'exécutions des tâches et la période de fonctionnement sont inconnues, il est toujours possible de calculer un ordonnancement K-périodique vérifiant ces contraintes :

Minimiser $\Omega_{\mathcal{G}}^{\mathcal{S}}$ avec

$$\begin{cases} \forall a \in \mathcal{A}, \forall (k,k') \in \mathcal{U}(a), \\ \quad \mathcal{S}\langle t', k' \rangle - \mathcal{S}\langle t, k \rangle \geq d(t) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_t^{\mathcal{G}}} \frac{\pi_a^{max}(k,k') \cdot \gcd_a^k}{in_a} \\ \forall t \in \mathcal{T}, \forall k \in \{1, \ldots, K_t^{\mathcal{G}}\}, \mathcal{S}\langle t, k \rangle \in \mathbb{Q}^{+\star} \\ \Omega_{\mathcal{G}}^{\mathcal{S}} \in \mathbb{Q}^{+\star} \end{cases}$$

On compte alors $1 + \sum_{t \in \mathcal{T}} K_t^{\mathcal{G}}$ variables pour ce programme linéaire, et le nombre de contraintes est bornée par

$$C_K = \sum_{\forall a=(t,t') \in \mathcal{A}} (K_t^{\mathcal{G}} \times K_{t'}^{\mathcal{G}}). \tag{5.6}$$

Le système associé au SDFG de la Figure 5.2 page 71 pour un vecteur de périodicité $K = [1, 1, 2]$ est donné par :

Minimiser $\Omega_{\mathcal{G}}^{\mathcal{S}}$ with

$$\begin{cases} a_{AB}: & \mathcal{S}\langle B,1\rangle - \mathcal{S}\langle A,1\rangle \geq 1 \\ a_{BC}: & \mathcal{S}\langle C,1\rangle - \mathcal{S}\langle B,1\rangle \geq 1 + 2\Omega_{\mathcal{G}}^{\mathcal{S}} \\ & \mathcal{S}\langle C,2\rangle - \mathcal{S}\langle B,1\rangle \geq 1 + 4\Omega_{\mathcal{G}}^{\mathcal{S}} \\ a_{CA}: & \mathcal{S}\langle A,1\rangle - \mathcal{S}\langle C,1\rangle \geq 1 - 8\Omega_{\mathcal{G}}^{\mathcal{S}} \\ & \forall t \in \mathcal{T}, \forall k \in \{1,\ldots,K_t^{\mathcal{G}}\}, \mathcal{S}\langle t,k\rangle \in \mathbb{Q}^{+\star} \\ & \Omega_{\mathcal{G}}^{\mathcal{S}} \in \mathbb{Q}^{+\star} \end{cases}$$

On remarque que $\pi_{a_{CA}}^{max}(2,1) < \pi_{a_{CA}}^{min}(2,1)$ et donc qu'il n'est pas nécessaire de considérer l'inégalité entre $\mathcal{S}\langle A,1\rangle$ et $\mathcal{S}\langle C,2\rangle$.

Une solution optimale vérifie $\Omega_{\mathcal{G}}^{\mathcal{S}} = 0.5$.

La Figure 5.2 page 71 représente un ordonnancement K-périodique optimal pour ce problème avec les dates de première exécution égales à :

$$\mathcal{S}\langle A,1\rangle = 0$$
$$\mathcal{S}\langle B,1\rangle = 1$$
$$\mathcal{S}\langle C,1\rangle = 3$$
$$\mathcal{S}\langle C,2\rangle = 4.$$

La complexité de cette méthode et les résultats de performance obtenus dépendent alors des facteurs de périodicité choisis. Intéressons-nous alors à l'effet du vecteur de périodicité sur la performance et le temps de calcul d'un ordonnancement K-périodique.

5.5 Influence du vecteur de périodicité

Lorsqu'un vecteur de périodicité est fixé, nous sommes en mesure de calculer un ordonnancement K-périodique valide de débit maximal pour un SDFG. La question fondamentale qui se pose maintenant porte sur le choix de ce vecteur de périodicité. Dans cette section, nous présentons différentes propriétés des vecteurs de périodicité facilitant ce choix. Nous fournissons notamment un ensemble de vecteurs jugés pertinents.

5.5.1 Étude des solutions K-périodiques par l'exemple

Afin de mieux percevoir l'impact du vecteur de périodicité sur les performances d'un ordonnancement et sur la complexité de son calcul, nous allons parcourir un ensemble d'ordonnancements valides pour le SDFG de la Figure 5.3.

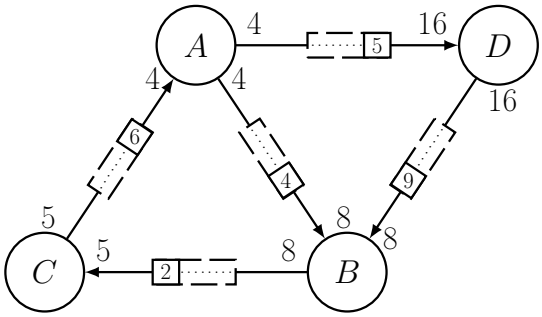


FIGURE 5.3 – Un SDFG pour lequel les durées d'exécution de ses tâches sont $d(t_1) = d(t_4) = 1$ et $d(t_2) = d(t_3) = 5$.

La Figure 5.4 est une exploration des solutions d'ordonnancement pour le SDFG de la Figure 5.3. Chaque nœud correspond à un ordonnancement K-périodique, dont le vecteur de périodicité est fixé. Deux nœuds reliés sont alors deux vecteurs proches c.-à-d. qui ne comptent pas plus d'un facteur de périodicité différent et pour lesquels cet écart est minimal. La taille d'un nœud correspond à la performance du débit maximal obtenu avec ce vecteur de périodicité, les nœuds les plus gros sont les ordonnancements dont le débit est le plus élevé. De même la couleur du nœud correspond à la durée du calcul de l'ordonnancement, les nœuds les plus sombres sont les ordonnancements les plus rapides à calculer.

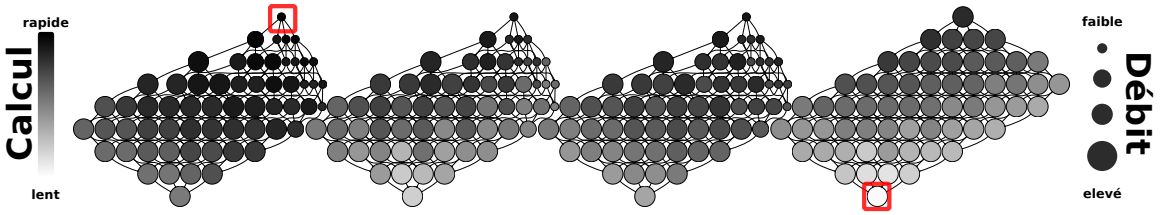


FIGURE 5.4 – Exploration d'un ensemble d'ordonnancements K-périodique. Les nœuds correspondent à des ordonnancements différents, les arcs relient des ordonnancements aux vecteurs similaires. Plus sombre est le nœud, plus le calcul de l'ordonnancement est rapide ; plus grand est le nœud, meilleurs est son débit.

Considérons maintenant deux nœuds particuliers (encadrés sur la figure).

- D'une part, le nœud le plus haut à gauche. Il s'agit d'un ordonnancement 1-périodique. Son débit maximal est le plus faible, mais son temps de calcul est aussi le plus court.
- D'autre part, le nœud le plus bas à droite : l'ordonnancement $N^{\mathcal{G}}$-périodique. Il s'agit d'un ordonnancement de débit optimal, mais pour lequel le temps de calcul

est le plus long. Pour cet ordonnancement tous les facteurs de périodicité sont égaux au facteur de répétition. Pour ce cas on note $K^{\mathcal{G}} = N^{\mathcal{G}} = [16, 20, 5, 10]$.

On s'aperçoit qu'il existe aussi d'excellents compromis entre ces deux ordonnancements. Deux d'entre eux sont proposés dans le Tableau 5.1. Il s'agit des vecteurs $[2, 1, 1, 1]$ et $[4, 1, 1, 1]$. La complexité de leur calcul est quasi-identique à celle du vecteur 1-périodique, mais leurs performances sont deux et quatre fois plus élevées.

Vecteur K	Taille (C_K)	Periode	Optimalité
$[1, 1, 1, 1]$	7	7	20%
$[2, 1, 1, 1]$	10	3.5	41%
$[\mathbf{4}, \mathbf{1}, \mathbf{1}, \mathbf{1}]$	**16**	**1.75**	**82**%
$[16, 20, 5, 10]$	87	1.45	100%

TABLE 5.1 – Différents exemples d'ordonnancements K-périodiques pour le SDFG de la Figure 5.3 page précédente. On précise leur vecteur de périodicité (K) ainsi que le nombre de contraintes requises pour les calculer avec notre programme linéaire (C_K), leur période minimale et la performance de cette période par rapport à l'optimal.

On remarque que la meilleure performance est atteinte par l'ordonnancement $N^{\mathcal{G}}$-périodique. Dans la section suivante, nous montrons qu'un ordonnancement $N^{\mathcal{G}}$-périodique de débit maximal atteint toujours le débit optimal d'une l'application.

5.5.2 Un ordonnancement K-périodique de débit maximal

Plusieurs auteurs [66, 77, 31] prouvent que les contraintes induites par un SDFG sur ses différentes exécutions peuvent être exprimées sous forme d'un HSDFG où chaque tâche est dupliquée $N_t^{\mathcal{G}}$ fois : c'est l'expansion. Il a aussi été prouvé qu'un ordonnancement 1-périodique de débit maximal pour ce HSDFG fournit la solution du débit maximal d'un ordonnancement *au plus tôt* [66]. Le théorème suivant est déduit :

Théorème 9 *Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ un SDFG avec un vecteur de répétition $N^{\mathcal{G}}$. Considérons un ordonnancement K-périodique dont le vecteur de périodicité est $K^{\mathcal{G}} = N^{\mathcal{G}}$. Son débit maximal atteignable est alors le débit maximal atteignable de $\mathcal{G}$.*

Prenons l'exemple du SDFG de la Figure 5.2 page 71 avec $K^{\mathcal{G}} = [3, 3, 4]$ son vecteur de périodicité.

Un ordonnancement K-périodique optimal est visible sur la Figure 5.5 page ci-contre. Son débit est alors égal à celui d'un ordonnancement *au plus tôt*.

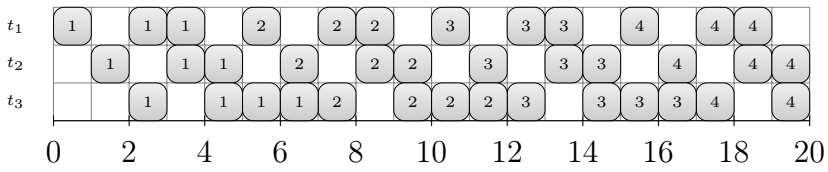


FIGURE 5.5 – Un ordonnancement K-périodique au vecteur de périodicité $K^{\mathcal{G}} = [3, 3, 4]$. La période minimale des tâches est $\mu^{\mathcal{S}}_{t_1} = \mu^{\mathcal{S}}_{t_2} = \mu^{\mathcal{S}}_{t_3} = 5$.

La complexité de calcul de cet ordonnancement $N^{\mathcal{G}}$-périodique est du même ordre que celle des méthodes *au plus tôt* que nous souhaitons éviter. Entre les ordonnancements 1-périodiques et $N^{\mathcal{G}}$-périodiques, on peut cependant considérer un ensemble d'ordonnancements K-périodiques nous permettant d'atteindre un débit pertinent dans un temps de calcul réduit. Néanmoins, parcourir l'espace de ces solutions d'ordonnancements K-périodiques n'est pas aussi simple qu'il puisse paraître : cet espace de solution n'est pas linéaire, deux vecteurs de périodicité proches peuvent par exemple donner des performances très différentes.

5.5.3 Une non-linéarité des solutions K-périodiques

Pour un graphe $\mathcal{G}$, le débit maximal atteignable à l'aide d'un ordonnancement K-périodique n'augmente pas nécessairement lorsque les valeurs du vecteur périodicité augmentent.

En effet, prenons l'exemple de la Figure 5.3 page 83 et des deux vecteurs de périodicité $[2, 1, 1, 1]$ et $[4, 1, 1, 1]$ rappelés dans le Tableau 5.1 page ci-contre. Ces deux vecteurs permettent de définir des ordonnancements dont le débit est au moins supérieur à deux fois celui d'un ordonnancement 1-périodique. Pourtant, il existe le vecteur de périodicité $[3, 1, 1, 1]$, pour lequel aucun ordonnancement K-périodique ne donne de meilleur résultat qu'un ordonnancement 1-périodique.

Il s'agit ici d'un flagrant exemple de non-linéarité entre les vecteurs de périodicité et le débit maximal qui leur est associé. Pour progresser dans l'espace des solutions K-périodiques, nous définissons des relations de dominance entre ces différents vecteurs.

5.5.4 Un ensemble de solutions dominantes

Pour tout graphe $\mathcal{G}$, on note $S^{opt}_{\mathcal{G}}(K^{\mathcal{G}})$ l'ensemble des ordonnancements K-périodiques de ce graphe dont le débit est maximal pour un vecteur de périodicité $K^{\mathcal{G}}$.

Le théorème suivant va alors définir un ordre de dominance entre différents vecteurs de périodicité pour un SDFG.

Théorème 10 *Soit un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ et deux vecteurs périodicité $K^{\mathcal{G}}$ et $K'^{\mathcal{G}}$. Si $\forall t \in \mathcal{T}$, $K'^{\mathcal{G}}_t = x \times K^{\mathcal{G}}_t$ avec $x \in \mathbb{N}^\star$, alors*

$$\forall \mathcal{S} \in S^{opt}_{\mathcal{G}}(K^{\mathcal{G}}),\ \forall \mathcal{S}' \in S^{opt}_{\mathcal{G}}(K'^{\mathcal{G}}),\ \Omega^{\mathcal{S}'}_{\mathcal{G}} \leq \Omega^{\mathcal{S}}_{\mathcal{G}}.$$

PREUVE : Considérons un ordonnancement $\mathcal{S} \in S^{opt}_{\mathcal{G}}(K^{\mathcal{G}})$. Pour tout vecteur $K'^{\mathcal{G}}$, un ordonnancement K'-périodique noté $\mathcal{S}'$ pour être construit à partir de l'ordonnancement $\mathcal{S}$ de la manière suivante :

1. $\forall t \in \mathcal{T}$, $\forall k \in \{1, \ldots, K'^{\mathcal{G}}_t\}$, $\mathcal{S}'\langle t, k\rangle = \mathcal{S}\langle t, k\rangle$;
2. La période d'une tâche $t \in \mathcal{T}$ est alors égale à $\mu^{\mathcal{S}'}_t = x \times \mu^{\mathcal{S}}_t$ et donc $\Omega^{\mathcal{S}'}_{\mathcal{G}} = \Omega^{\mathcal{S}}_{\mathcal{G}}$.

Observons que $\forall t \in \mathcal{T}$, $\forall p \in \mathbb{N}^\star$, $\mathcal{S}'\langle t, p\rangle = \mathcal{S}\langle t, p\rangle$.

Comme $\mathcal{S}'$ est un ordonnancement $K'^{\mathcal{G}}$-périodique valide, tout ordonnancement $\mathcal{S}' \in S^{opt}(K'^{\mathcal{G}})$ vérifie alors $\Omega^{\mathcal{S}'}_{\mathcal{G}} \leq \Omega^{\mathcal{S}'}_{\mathcal{G}}$, ce qui montre le théorème. ∎

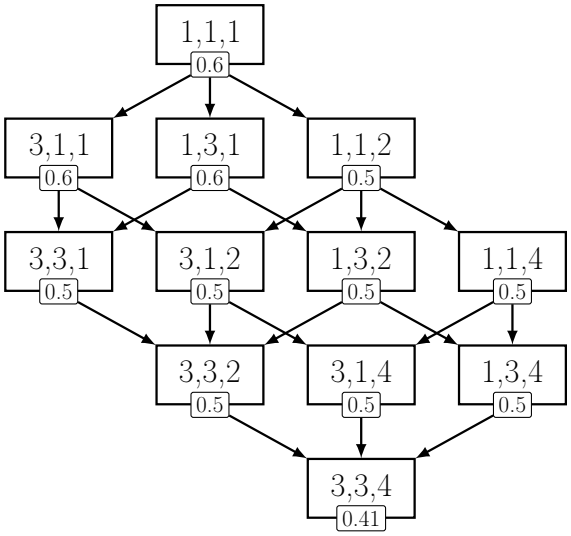


FIGURE 5.6 – Un Graphe $\mathcal{H}$ associé au SDFG de la Figure 5.2 page 71. Les nœuds sont étiquetés par le vecteur de périodicité correspondant et la période de fonctionnement minimal atteignable.

Les théorèmes 9 page 84 et 10 page précédente permettent alors de définir un ensemble de vecteurs de périodicité liés par une relation d'ordre.

En effet soit un graphe acyclique $\mathcal{H} = (V, E)$ défini par :

1. Un ensemble de nœuds $V = \{K^{\mathcal{G}} / \forall t \in \mathcal{T}, \exists x \in \mathbb{N}^\star, K^{\mathcal{G}}_t \times x = N^{\mathcal{G}}_t\}$, un sous-ensemble de tous les vecteurs de périodicité possibles et diviseur du vecteur de répétition.
2. Un ensemble d'arcs E tels que pour tout arc $e = (K^{\mathcal{G}}, K'^{\mathcal{G}}) \in E$ alors $\forall t \in \mathcal{T}$, $\frac{K'^{\mathcal{G}}_t}{K^{\mathcal{G}}_t} \in \mathbb{N}^\star$.

On remarque alors que le vecteur $\mathbb{1} = [1, \ldots, 1]$ est la racine de ce graphe et que le vecteur $N^{\mathcal{G}} = [N^{\mathcal{G}}_{t_1}, \ldots, N^{\mathcal{G}}_{t_n}]$ est un puits. Le débit de chaque nœud du graphe augmente le

long des chemins existants de $\mathbb{1}$ vers $N^{\mathcal{G}}$. D'autre part, comme le vecteur $N^{\mathcal{G}}$ est le puits de $\mathcal{H}$, l'ensemble est dominant.

La Figure 5.6 page précédente montre le graphe $\mathcal{H}$ associé au SDFG de la Figure 5.2 page 71. Les périodes optimales associées à chaque nœud y figurent.

5.6 Ordonnancement K-périodique d'un CSDFG

Il existe de fortes ressemblances entre les résultats de ce chapitre et ceux du chapitre précédent sur l'ordonnancement périodique des CSDFG ; ceci n'est pas une coïncidence.

On remarque par exemple l'existence d'une transformation du problème d'ordonnancement K-périodique d'un SDFG vers un problème ordonnancement périodique de CSDFG. Pour chaque tâche t d'un SDFG avec un facteur de périodicité $K_t^{\mathcal{G}}$ correspond alors une tâche t', du CSDFG avec $\varphi(t') = K_t^{\mathcal{G}}$ phases et produisant toujours les taux d'origine.

D'autre part, l'approche K-périodique peut être très facilement appliquée aux CSDFG. Il suffit alors pour toute tâche t d'un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ avec un facteur de périodicité $K_t^{\mathcal{G}}$ de répéter son itération $K_t^{\mathcal{G}}$ fois. Son nombre de phases devient $K_t^{\mathcal{G}} \times \varphi(t)$.

Dans son manuscrit de thèse, Benazouz [11] propose une transformation similaire qu'il appelle l'*harmonisation*. Celle-ci consiste, pour chaque tâche, à répéter l'ensemble de ses phases autant de fois qu'il est nécessaire afin de ramener son vecteur de répétition à 1, une solution minimale est alors $\forall t \in \mathcal{T}$, $K_t^{\mathcal{G}} = N_t^{\mathcal{G}}$. Des outils de dimensionnement basés sur les ordonnancements périodiques pourraient alors se comporter comme s'ils traitaient le cas $N^{\mathcal{G}}$-périodique. Bien évidemment, cette transformation est exponentielle, mais les améliorations des résultats de dimensionnement sont alors incontestables [11].

Conclusion

Dans ce chapitre, nous proposons une méthode d'ordonnancement K-périodique des SDFG pour un vecteur de périodicité fixé. La question du choix des facteurs de périodicité est ensuite posée ; nous fournissons le graphe $\mathcal{H}$, un premier ensemble de solutions pertinentes. Nous remarquons aussi la forte similitude entre les démarches analytiques mises en œuvre pour prouver ces travaux et ceux du chapitre 4 portant sur l'ordonnancement périodique des CSDFG. L'application de la K-périodicité sur les CSDFG est alors possible.

Comme la complexité de calcul d'un ordonnancement *au plus tôt* est trop élevée pour s'appliquer aux nouvelles instances de ΣC, nous pensons que l'ordonnancement K-périodique est une solution pertinente pour pallier les limites des ordonnancements périodiques. Cependant, nos travaux sur ce sujet restent encore incomplets. Afin de sérieusement envisager l'application des ordonnancements K-périodiques à l'analyse statique

d'applications ΣC, il va être nécessaire d'établir des techniques d'exploration efficaces au travers des vecteurs de périodicité.

Pour cette raison, aucune technique utilisant la K-périodicité n'a encore été intégrée à la chaîne de compilation AccessCore. Dans les chapitres qui suivent, toutes les techniques d'analyse considérées seront uniquement basées sur les ordonnancements périodiques.

Deuxième partie

Application à la compilation dataflow

Chapitre 6

Analyse statique des CSDFG

Introduction

Lors de la compilation d'un programme dataflow, différentes étapes d'analyses peuvent survenir. Dans la chaîne de compilation AccessCore, après la modélisation CSDFG d'une application ΣC, on rencontre notamment des étapes de dimensionnement mémoire ou d'évaluation des performances. Ces étapes, énumérées dans le chapitre 3, sont aujourd'hui résolues par des algorithmes approchés et dont la complexité s'avère être exponentielle.

Pourtant, l'expérience nous montre que la taille et la complexité des CSDFG rencontrés avec le langage ΣC augmentent de plus en plus ; c'est en partie liée à la modularité du langage. Aujourd'hui, on rencontre des compilations ΣC d'une durée moyenne de deux à cinq minutes, mais certaines applications – pour l'instant, jugées mal-formées – peuvent entrainer des compilations de plusieurs heures. Cette situation n'est pas acceptable pour un produit du commerce.

Pour y remédier, en nous aidant des résultats théoriques présentés dans les chapitres 4 et 5, nous proposons de nouvelles techniques algorithmiques pour résoudre les problèmes de vivacité, d'évaluation du débit ou de dimensionnement mémoire des CSDFG. Nous évaluons ensuite ces contributions en nous appuyant sur des jeux de test industriels et académiques.

Dans ce chapitre, nous commençons par présenter nos jeux de test, ainsi que le générateur d'instances aléatoires que nous avons développé. Nous traitons ensuite les étapes de compilation pour lesquelles nous avons contribué : la vivacité, l'évaluation du débit, et le dimensionnement mémoire. Pour chacune de ces étapes, nous présentons nos algorithmes et les études expérimentales qui les accompagnent.

6.1 Nos jeux de test

Les problèmes d'analyse qui sont étudiés dans ce chapitre ont tous une caractéristique commune : il n'existe aucun algorithme polynomial pour les résoudre.

À défaut d'algorithmes exacts et polynomiaux, nous avons proposé différentes méthodes approchées, mais leur l'écart de performance avec l'optimal est inconnu. Afin d'en évaluer les performances, nous ne pouvions alors compter que sur des études expérimentales. Pour cette raison, des jeux de tests pertinents furent nécessaires.

Dans un premier temps, nous présentons les applications CSDFG déjà considérées dans la littérature ainsi que l'outil de génération aléatoire SDF3 [81]. Ce générateur ne nous ayant pas permis de générer des instances suffisamment grandes, nous en proposons un nouveau. Nous détaillons ensuite les jeux de test que nous avons sélectionnés. Ces tests répondent à deux besoins précis : nous souhaitons disposer d'applications représentatives

de la réalité industrielle, et nous devons aussi tester le passage à l'échelle de nos méthodes avec des applications de grande taille.

6.1.1 Les jeux de test existants

Pour étudier expérimentalement des outils d'analyse dataflow, différents jeux de test furent proposés dans la littérature, mais leur nombre reste encore très marginal. Cette lacune s'accentue lorsqu'il s'agit de modèle particulier comme les CSDFG. On trouve cependant quelques exemples isolés ; nous en citons trois. Nous présentons ensuite le générateur de test aléatoire SDF3 [81].

H.263 Encoder

Le *H.263 Encoder* est une application d'encodage vidéo basée sur une norme de compression vidéo officialisée pour la première fois en 1996. Elle fut plus récemment modélisée par Oh et Ha [69] afin d'évaluer des outils d'ordonnancement et de placement. Cette modélisation CSDFG est visible sur la Figure 6.1.

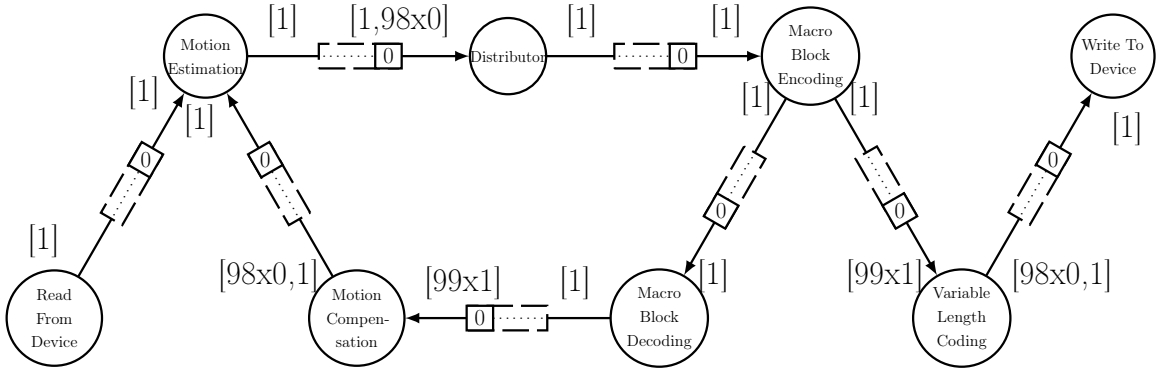


FIGURE 6.1 – L'application *H.263 Encoder* proposée par Oh et Ha [69]. La notation $n \times m$ indique une séquence de n phases avec un taux de m.

Les interfaces d'entrée/sortie sont modélisées par les tâches `Read From Device` et `Write To Device`. Le principal intérêt de ce CSDFG est qu'il contient un circuit. En effet, rappelons qu'il existe des méthodes d'analyse qui ne sont pas en mesure de traiter correctement ce type de structure.

MP3 Playback

Le *MP3 Playback* modélise un équipement de décodage audio proposé par Wiggers et al. [93] pour évaluer leur technique de dimensionnement mémoire. La Figure 6.2 page suivante présente la modélisation CSDFG de cette application.

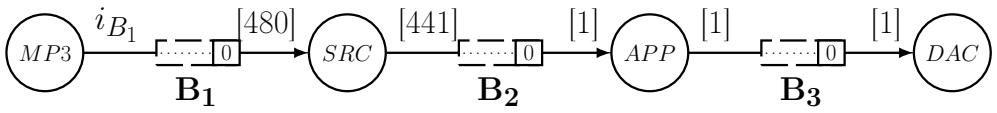


FIGURE 6.2 – L'application *MP3 Playback* proposée par Wiggers et al. [93]. Les taux de production de la tâche MP3 sont $i_{B_1} = [0, 0, 18 \times 32, 0, 18 \times 32]$ et les durées de ses phases $d(MP3) = [670, 2700, 18 \times 40, 2700, 18 \times 40]\mu$s. La durée d'exécution des tâches APP et DAC sont identiques $d(MP3) = d(MP3) = \frac{1}{44100}$s et celle de la tâche SRC, $d(SRC) \in \{2.5, 5, 7.5, 10\}$ms et peut varier en fonction de sa configuration. La notation $n \times m$ indique une séquence de n phases avec un taux de m.

La particularité de cet exemple est la durée de fonctionnement de la tâche SRC qui peut varier selon un choix de configuration. Ce choix pouvait alors être une part du travail d'analyse. Avec cet exemple, il est possible d'entrevoir l'impact de la durée d'une tâche sur le dimensionnement mémoire nécessaire afin d'atteindre un certain débit de fonctionnement.

Reed-Solomon Decoder

Benazouz [11] a présenté récemment le *Reed-Solomon Decoder*, visible sur la Figure 6.3. Il s'agit d'un outil de décodage de *frame* avec correction d'erreur proposé par le centre de recherche STMicroelectronics de Crolles. Des *frames* sont récupérées par la tâche S_1, et le résultat du décodage doit être ensuite être traité par la tâche E_2. Le système dispose alors d'une limite de 1152 cycles pour traiter chaque *frame*.

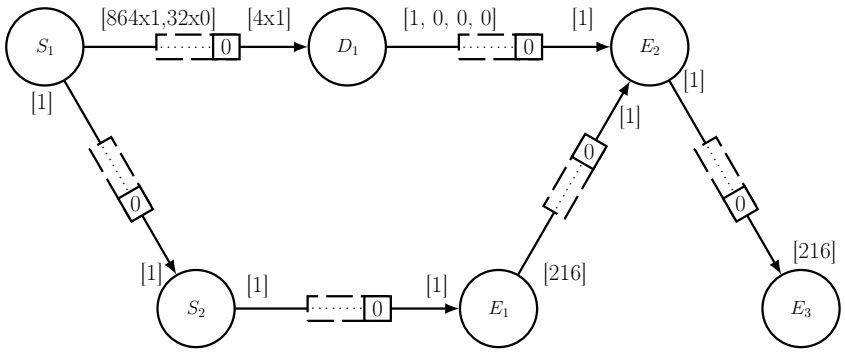


FIGURE 6.3 – L'application *Reed-Solomon Decoders* : la durée des phases de chaque tâche est définie par le nombre de jetons traités ; chaque jeton produit ou consommé par une phase augmente sa durée d'un cycle. Par exemple, la durée de la tâche E_1 est $d(E_1) = 217$. La notation $n \times m$ indique une séquence de n phases avec un taux de m.

Ce CSDFG se démarque par l'important nombre de phases pour la tâche S_1 (il en compte 896). Cette caractéristique peut avoir des conséquences désastreuses pour des algorithmes trop sensibles au nombre de phases d'un CSDFG. Benazouz [11] proposait notamment des algorithmes de complexité quadratique, mais dont la complexité restait linéaire en fonction du nombre de phases.

Le générateur aléatoire SDF3

Compte tenu du faible nombre d'exemple de SDFG ou de CSDFG disponibles, le *Electronic Systems Group* a mis en ligne un générateur de graphe aléatoire, SDF3 [81]. Nous résumons son fonctionnement.

Ce générateur se décompose en trois étapes générant un SDFG. Elles sont suivies d'une quatrième étape pour la génération des CSDFG.

1. Dans une première étape, la structure du graphe est construite. Le nombre de nœuds est défini par N un paramètre de l'algorithme. Le degré de chaque nœud va quant à lui être caractérisé par $X \sim \mathcal{N}(\mu, \sigma^2)$, une loi normale tronquée d'intervalle $[a, b]$ où μ, σ^2,a et b sont des paramètres de l'algorithme. Les taux de production sur chaque arc sont aussi définis aléatoirement par une seconde loi normale tronquée.
2. L'étape suivante doit garantir la consistance du graphe. Pour cela, l'algorithme commence par résoudre le système d'équations utilisé pour calculer le vecteur de répétition. En cas de conflit, il corrige arbitrairement les taux concernés pour garantir l'existence d'une solution.
3. Ensuite, une étape de génération du marquage initial s'applique en ajoutant systématiquement des jetons dans tous les circuits non vivants du graphe.
   À ce stade, le graphe généré est un SDFG vivant et consistant.
4. Pour la génération de CSDFG, une dernière étape va finalement répartir dans différentes phases les taux de production et de consommation des tâches.

À l'aide de cet algorithme, Stuijk et al. [81] génèrent une série de jeux de test SDFG afin d'évaluer leurs outils. Nous avons nous-mêmes utilisé avec succès cet algorithme pour générer des CSDFG de petite taille.

Ce générateur souffre cependant de faiblesses notables :

- Tout d'abord, son système de marquage initial propose généralement des solutions surévaluées, un phénomène rare pour les applications réelles.
- Ensuite, l'algorithme de marquage utilisé est trop complexe pour fournir des solutions aux graphes de grande taille.
- La génération des taux de production et de consommation souffre elle aussi d'une imperfection. Comme un système de réajustement est utilisé pour maintenir la consistance du graphe, les lois de distribution utilisées pour définir ces taux ne sont pas nécessairement respectées.
- Enfin, comme les phases propres aux CSDFG ne sont produites qu'en fin de génération, on rencontre généralement un grand nombre de taux nuls et parfois même des phases illégales sans aucune production ou consommation.

6.1.2 Contribution à la génération aléatoire

Compte tenu des limites du générateur de Stuijk et al. [81], nous avons developpé un nouveau générateur pour des CSDFG vivants, consistants et de grandes tailles.

Principe de fonctionnement

Notre objectif est de générer rapidement de nouveaux CSDFG. Nous avons conservé au maximum la structure du générateur SDF3, et n'avons optimisé que les parties que nous jugions nécessaires.

La première étape de notre méthode conserve la génération des arcs et des nœuds de SDF3, mais les taux de production et de consommation n'y sont plus assignés. Dans une seconde étape, nous nous appuyons sur la propriété de normalisation (vue dans le chapitre 3) pour directement produire un CSDFG consistant. Nous générons enfin le marquage initial en nous appuyant des résultats d'ordonnancement périodique du chapitre 4.

Génération des taux normalisés

Une fois le graphe construit, grâce à la normalisation [63], la génération des taux va être une étape particulièrement rapide.

Des taux normalisés Z_t sont fixés pour chaque tâche t. Ces taux normalisés seront sélectionnés de sorte que le vecteur de répétition du graphe respecte bien les paramètres attendus. Ensuite, afin que les taux respectent la loi de distribution requise par les paramètres de l'algorithme, des facteurs multiplicatifs sont appliqués sur chaque arc.

Génération d'un marquage minimal

La principale faiblesse du générateur SDF3 réside dans sa technique de génération du marquage. D'une part le marquage produit est loin d'être un marquage minimal (ce qui est rare dans des applications réelles), d'autre par la complexité de cette technique dépend du nombre de circuits dans le graphe, un nombre dont la complexité pire-cas est exponentielle.

Notre technique consiste à concevoir un système de contraintes vérifiant l'existence d'un ordonnancement périodique, et dont le marquage est inconnu. Un algorithme similaire est présenté dans la section 6.4 page 108 pour calculer le dimensionnement mémoire minimal d'un CSDFG. Il s'agit d'une méthode approchée, mais permettant d'obtenir une solution réalisable très rapidement, et dont les résultats obtenus sont souvent proches de l'optimal.

D'autre part, en nous aidant des résultats du chapitre 5 sur la K-périodicité et sur le vecteur de périodicité optimal, nous sommes en mesure de calculer le marquage optimal.

Cependant le temps d'exécution de cette méthode est potentiellement trop elevé pour générer les graphes de grande taille que nous visons.

6.1.3 Notre sélection de CSDFG

Des exemples autogénérés

À l'aide de notre générateur, nous avons produit différentes applications CSDFG de grande taille numérotées de 1 à 5. Les CSDFG `autogen1`, `autogen2` et `autogen3` sont des applications de taille moyenne, mais dont le vecteur de répétition est considérablement élevé. Les CSDFG `autogen4` et `autogen5` sont quant à eux des applications de grande taille avec un nombre de phases élevé, mais dont le vecteur de répétition reste faible. Le Tableau 6.1 synthétise ce jeu de test.

Application	Tâches	Buffers	Phases	Ordo. Minimal
autogen1	90	617	2651866	1361124
autogen2	70	473	1500829	8869482600
autogen3	154	671	1973411	11005386480
autogen4	2426	2900	10129700	1599780
autogen5	2767	4894	42851864	3614142

TABLE 6.1 – Résumé des cinq CSDFG autogénérés par notre méthode. Les colonnes Tâches, Buffers et Phases correspondent respectivement au nombre de tâches, de buffers et de phases pour les tâches dans le modèle CSDFG. La dernière colonne correspond au nombre minimal d'exécutions nécessaires pour que l'application puisse revenir à son état initial ($\sum_{t\in\mathcal{T}} N_t^{\mathcal{G}}$).

Des applications ΣC

Afin d'attester des performances du processeur MPPA et de la chaîne de compilation AccessCore, la société Kalray a développé différentes applications ΣC. Le choix de ces applications s'est fait selon les demandes du marché ; elles sont alors représentatives des applications que l'on peut rencontrer dans un contexte industriel.

Nous avons été en mesure d'expérimenter nos méthodes sur toutes ces applications. Cependant, par souci de confidentialité et de clarté, seul cinq d'entre elles furent rendues publiques [48]. Nous présentons chacune d'entre elles, et le Tableau 6.2 page suivante en fait la synthèse.

Application	Tâches	Buffers	Phases	Ordo. Minimal
BlackScholes	41	40	1560	79495
Echo	38	82	10332	784024
JPEG2000	240	703	2389497	1236486636
Pdetect	58	76	8664	1191979200
H264 Encoder	665	3128	55243608	120868020

TABLE 6.2 – Résumé des cinq applications ΣC rendues publiques. Les colonnes Tâches, Buffers et Phases correspondent respectivement au nombre de tâches, de buffers et de phases de tâches dans le modèle CSDFG. La dernière colonne correspond au nombre minimal d'exécutions nécessaires pour que l'application puisse revenir à son état initial.

BlackScholes L'application BlackScholes est un outil financier s'appuyant sur un modèle mathématique du même nom. Le code principal de ce programme résout des équations différentielles ; c'est un exemple classique de la programmation parallèle. Dans notre jeu de test, le BlackScholes se caractérise par sa taille raisonnable.

JPEG2000 L'application JPEG2000 est un encodeur vidéo respectant une norme du même nom. Cette norme est connue pour être plus performance que le JPEG standard, elle rend aussi possible la compression sans perte. Pour ces raisons, elle peut être utilisée dans des domaines variés comme l'imagerie médicale, ou l'audiovisuel. Il s'agit d'un exemple de graphe complexe ; son nombre de tâches est important et son vecteur de répétition est élevé.

Pdetect Pdetect est un outil de détection de piétons s'appuyant sur la méthode de ViolaJones. On retrouve dans cette application les mêmes caractéristiques que le JPEG2000. Cependant le nombre de phases de ses tâches y est plus élevé.

Echo Echo est un filtre audio supprimant l'effet d'écho souvent produit au cours d'une conversation téléphonique. Il s'agit d'un exemple d'application de taille moyenne, et contenant un circuit.

H264 Encoder Cette dernière application est un encodeur multimédia basé sur la norme H264. Il s'agit d'un exemple d'application de grande taille et pour laquelle le nombre de circuits est exponentiel. Il est particulièrement difficile d'analyser ce type d'application avec des techniques traditionnelles.

6.2 Vivacité

La vérification de la vivacité d'un CSDFG est une opération dont la complexité est incertaine. Les meilleurs algorithmes s'appuient aujourd'hui sur une exécution symbolique du graphe pour vérifier cette propriété ; c'est notamment le cas de l'algorithme utilisé dans la chaîne de compilation AccessCore.

Dans cette section, nous proposons une nouvelle méthode pour garantir la vivacité d'un CSDFG. Elle s'appuie sur deux conditions suffisantes de vivacité proposées par Benazouz [11] que nous avons couplées avec une technique d'exécution symbolique.

6.2.1 Évaluation des conditions SC1 et SC2

Dans sa thèse, Benazouz [11] propose deux nouvelles conditions suffisantes de vivacité. La première est énoncée par le théorème 11 ; nous l'avons nommée la condition (SC1). Elle est définie grâce à la normalisation d'un CSDFG.

Théorème 11 *[11] (SC1) Soit $\mathcal{G}$ un CSDFG normalisé. Le graphe $\mathcal{G}$ est vivant si pour chacun de ses circuits, $c = (t^1, a_1, t^2, a_2, \cdots, t^m, a_m, t^1)$, et pour toute valeur $k^i \in \{1, \cdots, \varphi(t^i)\}$ avec $i \in \{1, \cdots, m\}$,*

$$\sum_{i=1}^{m} M_0(a_i) > \sum_{i=1}^{m} \left[O_{a_{i-1}}\langle {t^i}_{k^i}, 1\rangle - I_{a_i} Pr\langle {t^i}_{k^i}, 1\rangle \right] - \sum_{i=1}^{m} step_{a_i} \qquad avec \;\; a_0 = a_m.$$

Benazouz [11] remarque que la complexité de l'algorithme utilisé pour vérifier cette condition sur un CSDFG va fortement dépendre du nombre de ses phases. Il propose alors la seconde condition énoncée par le théorème 12 (SC2).

Théorème 12 *(SC2) Soit $\mathcal{G}$ un CSDFG normalisé. Le graphe $\mathcal{G}$ est vivant si pour chacun de ses circuits, $c = (t^1, a_1, t^2, a_2, \cdots, t^m, a_m, t^1)$,*

$$\sum_{i=1}^{m} M_0(a_i) > -\sum_{i=1}^{m} step_{a_i} + \sum_{i=1}^{m} \max_{k^i \in \{1,\cdots,\varphi(t^i)\}} \left[O_{a_{i-1}}\langle t^i, k^i\rangle - I_{a_i} Pr\langle t^i, k^i\rangle \right].$$

Nous prouvons que ces deux conditions sont équivalentes. Nous présentons ensuite deux techniques polynomiales pour vérifier ces conditions et notre algorithme d'évaluation de la vivacité sera proposé.

Les conditions SC1 et SC2 sont équivalentes

Au cours de nos expérimentations, nous nous sommes aperçus que les conditions SC1 et SC2 étaient équivalentes. Nous l'avons par la suite prouvé avec le théorème 13.

Théorème 13 *Soit $\mathcal{G}$ un CSDFG normalisé ; les conditions SC1 et SC2 appliquées au CSDFG $\mathcal{G}$ sont équivalentes.*

PREUVE : Supposons que le graphe $\mathcal{G}$ vérifie la condition SC2, et notons $c = (t^1, a_1, t^2, a_2, \cdots, t^m, a_m, t^1)$ un circuit de $\mathcal{G}$. Pour toute valeur $k^i \in \{1, \cdots, \varphi(t^i)\}$, $i \in \{1, \cdots, m\}$, nous obtenons que

$$\max_{k^i \in \{1,\cdots,\varphi(t^i)\}} \left[O_{a_{i-1}}\langle t^i, k^i\rangle - I_{a_i} Pr\langle t^i, k^i\rangle\right] \geq O_{a_{i-1}}\langle t^i, k^i\rangle - I_{a_i} Pr\langle t^i, k^i\rangle.$$

Alors,

$$\sum_{i=1}^{m} M_0(a_i) > -\sum_{i=1}^{m} \text{step}_{a_i} + \sum_{i=1}^{m} \left[O_{a_{i-1}}\langle t^i, k^i\rangle - I_{a_i} Pr\langle t^i, k^i\rangle\right].$$

et $\mathcal{G}$ vérifie aussi SC1.

Inversement, si SC1 est vérifiée, pour tout circuit $c = (t^1, a_1, t^2, a_2, \cdots, t^m, a_m, t^1)$ et pour toute phase $k_i^\star$ de la tâche t_i maximisant $O_{a_{i-1}}\langle t^i, k_i^\star\rangle - I_{a_i} Pr\langle t^i, k_i^\star\rangle$, l'inégalité est vérifiée. SC2 est donc vérifiée, ce qui conclut la preuve. ∎

Vérification de la condition SC1

Le but de cette section est d'exprimer un algorithme polynomial afin de vérifier la condition de vivacité SC1 sur un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$.

Pour vérifier cette propriété, nous considérons un graphe orienté et pondéré $H_1 = (V_1, E_1)$ défini comme suit :

- V_1 est l'ensemble des phases d'exécution de chacune des tâches appartenant à $\mathcal{T}$, c.-à-d. $V_1 = \{t_k, t \in \mathcal{T}, k \in \{1, \cdots, \varphi(t)\}\}$.
- Pour chaque arc $a = (t, t') \in \mathcal{A}$, on associe $\varphi(t) \times \varphi(t')$ arcs $u = (t_k, t'_{k'}) \in E_1$ pour $k \in \{1, \cdots, \varphi(t)\}$ et $k' \in \{1, \cdots, \varphi(t')\}$ et pondérés par $W_1(u) = O_a\langle t'_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - \text{step}_a - M_0(a)$.

Le coût moyen d'un circuit $c = (t^1_{k^1}, u_1, t^2_{k^2}, \cdots, t^m_{k^m}, u_m, t^1_{k^1})$ est alors défini par

$$W_1(c) = \frac{\sum_{i=1}^{m} W_1(u_i)}{m}.$$

On note $W_1^\star$ le coût moyen maximum d'un circuit de H_1. La condition SC1 est équivalente à $W_1^\star < 0$.

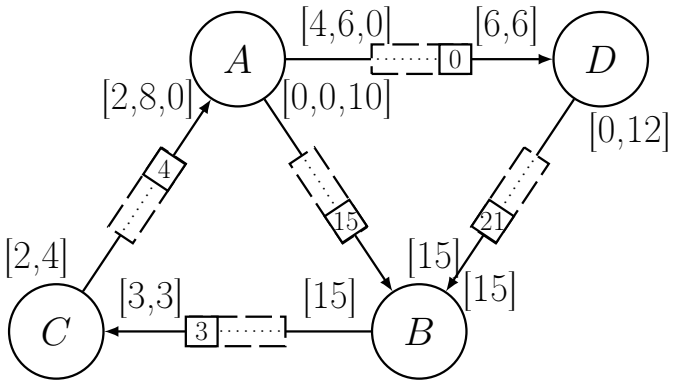


FIGURE 6.4 – Un exemple de CSDFG normalisé.

La Figure 6.5 représente le graphe H_1 associé au CSDFG normalisé de la Figure 6.4. L'arc u entre les phases A_1 et B_1 se calcul par exemple $W_1(u) = 15 - 0 - 5 - 15 = -5$. Le coût moyen maximum du H_1 est $W_1^\star = -\frac{1}{4}$; il est atteint par le circuit passant par les tâches A_2, D_2, B_1, C_2, A_2. Comme $W_1^\star < 0$, la condition SC1 est vérifiée par tous les circuits de H_1 et le graphe $\mathcal{G}$ est vivant.

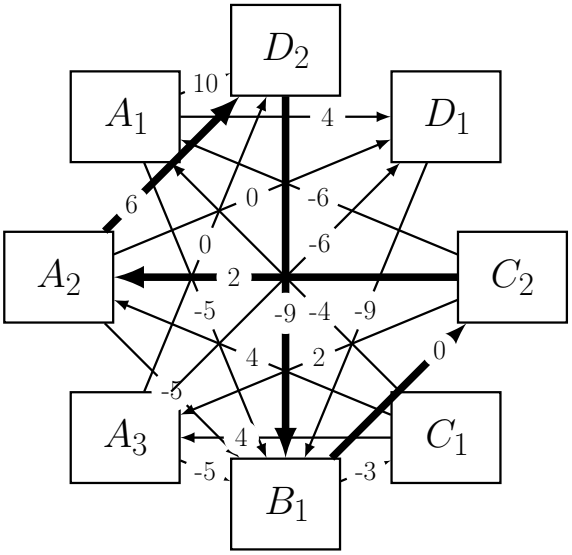


FIGURE 6.5 – Le Graphe H_1 associé au CSDFG normalisé de la Figure 6.4. Le circuit de coût moyen maximum est $c = \{A_2, D_2, B_1, C_2, A_2\}$. Son coût moyen maximum est $-\frac{1}{4}$.

La recherche de $W_1^\star$ est équivalent au problème du *Max Cycle Mean* ou MCM. Plusieurs algorithmes polynomiaux permettent alors de la résoudre [30].

La complexité d'un algorithme utilisé pour résoudre le MCM d'un graphe H_1 est bornée par

$$\Theta(|V_1| \times |E_1|) = \Theta(\sum_{t \in \mathcal{T}} \varphi(t) \times \sum_{(t,t') \in \mathcal{A}} \varphi(t) \times \varphi(t')).$$

Comme Benazouz [11] le fait remarquer, cette complexité dépend fortement du nombre de phases des tâches. C'est pour cette raison que la condition SC2 a été envisagée.

Vérification de la Condition SC2

On note $H_2 = (V_2, E_2)$ le graphe pondéré permettant de vérifier la condition SC2 pour un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, sa construction suit :

- V_2 est l'ensemble des nœuds, chaque nœud correspondant à un arc du graphe $\mathcal{G}$, c.-à-d. $V_2 = \mathcal{A}$.
- Chacun des arcs $e = (a, a') \in E_2$ est associé à une tâche $t \in \mathcal{T}$ tel que $(a, a') \in \mathcal{A}^-(t) \times \mathcal{A}^+(t)$, et est pondéré par

$$W_2(e) = \max_{k \in \{1, \cdots, \varphi(t)\}} [O_a\langle t_k, 1\rangle - I_{a'} Pr\langle t_k, 1\rangle] - \text{step}_{a'} - M_0(a').$$

Tout comme précédemment, satisfaire la condition SC2 revient à vérifier que le circuit de coût moyen maximum du graphe H_2 (noté $W_2^\star$) est strictement négatif. Le graphe H_2 correspondant au CSDFG normalisé $\mathcal{G}$ de la Figure 6.4 page précédente est visible en Figure 6.6. Le poid de l'arc e entre les buffers a_{AB} et a_{BC} se calcul par exemple $W_2(e) = 15 - 3 - 3$.

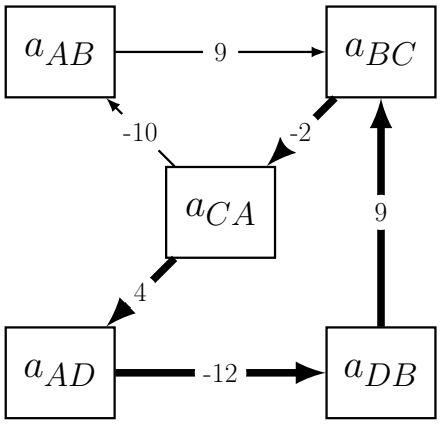


FIGURE 6.6 – Le Graphe H_2 associé au CSDFG normalisé de la Figure 6.4 page précédente. Le circuit de coût moyen maximum passe par les nœuds a_{BC}, a_{CA} a_{AD} et a_{DB} et est surligné.

Ainsi, la complexité pour résoudre le MCM d'un graphe H_2 est

$$\Theta(|\mathcal{A}| \times \sum_{t \in \mathcal{T}} (deg^+(t) \times deg^-(t))).$$

6.2.2 Un algorithme d'évaluation de la vivacité

Les conditions SC1 et SC2 sont équivalentes, mais la complexité de leur évaluation est différente. Nous sommes en mesure d'estimer leur temps de calcul théorique, et de désigner la plus rapide des deux.

D'autre part, ces deux algorithmes ne vérifient qu'une condition suffisante de vivacité. Il est donc possible de rencontrer des CSDFG vivants, mais pour lesquels cette condition n'est pas vérifiée. En cas de résultat négatif, il est nécessaire de confirmer ce résultat à l'aide d'une autre méthode ; nous utiliserons pour cela SE, une exécution symbolique standard développée par Anapalli et al. [3]. Le pire cas de cette exécution symbolique est $\mathcal{O}(|\mathcal{A}| \times \sum_{t \in \mathcal{T}} N_t^{\mathcal{G}})$.

6.2.3 Résultats expérimentaux

Dans cette section, nous comparons les performances des conditions SC1 et SC2 avec une méthode d'exécution symbolique (que nous nommerons SE). Ces résultats nous permettent d'évaluer la performance de l'algorithme SC1/SC2, et d'estimer si cette méthode peut améliorer les techniques actuelles d'évaluation de la vivacité.

Dans le cas d'applications en situation d'interblocage, la complexité de l'algorithme SE est plus faible, mais la complexité des méthodes SC1 et SC2 ne varie pas.

SE atteint son maximum de complexité pour des applications vivantes. Pour cette raison nous ne considérons que des applications vivantes.

Le Tableau 6.3 indique le temps de calcul des trois méthodes (SE, SC1 et SC2) avec nos différents jeux de test (résumés sur les Tableaux 6.1 page 97 et 6.2 page 98). Les résultats encadrés correspondent à la méthode choisie par l'heuristique de SC1/SC2, laquelle consiste à sélectionner l'algorithme dont la complexité théorique est la plus faible. On s'aperçoit que ce choix est généralement correct.

Application	SC1	SC2	SE
BlackScholes	2.10^5 / 14ms	$\boxed{1.10^3 / 0\text{ms}}$	2.10^5 / 1ms
JPEG2000	8.10^6 / 114ms	$\boxed{2.10^6 / 18\text{ms}}$	4.10^7 / 113ms
Echo	$\boxed{6.10^3 / 1\text{ms}}$	1.10^4 / 0ms	6.10^6 / 95ms
Pdetect	1.10^9 / 2500ms	$\boxed{8.10^3 / 4\text{ms}}$	6.10^5 / 5ms
H264	3.10^7 / 504ms	5.10^7 / 936ms	$\boxed{9.10^6 / 114\text{ms}}$
autogen1	$\boxed{1.10^5 / 13\text{ms}}$	2.10^6 / 55ms	3.10^8 / 1544ms
autogen2	$\boxed{7.10^5 / 41\text{ms}}$	1.10^6 / 37ms	3.10^{10} / 4min
autogen3	$\boxed{1.10^6 / 55\text{ms}}$	1.10^6 / 55ms	4.10^{11} / 21min
autogen4	7.10^7 / 217ms	$\boxed{1.10^7 / 71\text{ms}}$	2.10^8 / 132ms
autogen5	5.10^7 / 787ms	$\boxed{4.10^7 / 708\text{ms}}$	3.10^9 / 1442ms

TABLE 6.3 – Ce tableau indique les complexités des méthodes SC1, SC2 et SE [3] calculées selon l'heuristique de SC1/SC2. Chaque complexité est accompagnée du temps de calcul expérimental. Les résultats encadrés correspondent aux méthodes sélectionnées par cette même heuristique.

D'après les résultats obtenus, les conditions SC1 et SC2 semblent complémentaires. SC2 est souvent plus rapide que SC1, mais si le degré des tâches est élevé, comme c'est le cas pour *H264 Encoder*, alors SC1 peut être un meilleur choix.

Avec notre jeu de test industriel, le temps de calcul de la méthode SE reste particulièrement compétitive vis-à-vis des conditions SC1 et SC2. Ce n'est plus le cas pour nos applications académiques, elles sont plus complexes et creusent l'écart ; le passage à l'échelle de SC1 et SC2 semble confirmé.

6.3 Évaluation du débit

Aujourd'hui, la chaîne de compilation AccessCore ne dispose d'aucune méthode statique pour évaluer la fréquence de fonctionnement d'une application ΣC. Cette analyse se fait généralement par simulation ou bien directement en exécutant l'application sur le processeur MPPA.

Dans la littérature, il n'existe que deux méthodes exactes pour évaluer la fréquence de fonctionnement d'un CSDFG : l'exécution symbolique et l'expansion. Cependant la complexité de ces deux méthodes est exponentielle.

Dans le chapitre 4 nous avons présenté la caractérisation des contraintes de précédence induites par les buffers d'un CSDFG pour un ordonnancement périodique. Contrairement à un ordonnancement *au plus tôt*, le nombre de contraintes nécessaires pour définir cet ordonnancement est polynomial.

Maintenant, nous allons étudier le débit maximal d'un CSDFG en limitant notre espace des solutions aux ordonnancements périodiques. La méthode que nous proposons est une méthode d'évaluation du débit approchée, mais de complexité polynomiale.

6.3.1 Calcul du débit périodique maximal d'un CSDFG

Si l'on intègre les conditions d'ordonnancement périodiques du chapitre 4 dans un système linéaire et si l'on considère la période normalisée comme une variable, il est possible de calculer un ordonnancement périodique qui minimise cette période.

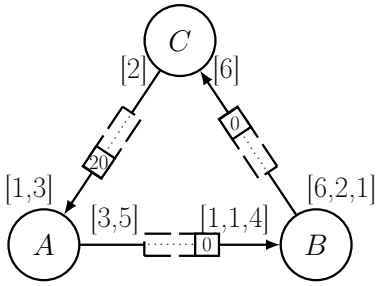


FIGURE 6.7 – Un exemple de CSDFG dont les durées de ses tâches sont $d(A) = [3, 1]$, $d(B) = [2, 1, 2]$, et $d(C) = [1]$.

Soit $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ un CSDFG. Pour tout arc $a \in \mathcal{A}$, on définie l'ensemble de couples $\mathcal{Y}(a)$ tel que

$$\mathcal{Y}(a) = \{(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\},\ \ \alpha_a^{\min}(k, k') \leq \alpha_a^{\max}(k, k')\}.$$

D'après le théorème 6 page 65, déterminer la période minimale $\Omega_{\mathcal{G}}^{\star}$ d'un ordonnancement périodique peut se modéliser par le programme linéaire suivant :

Minimiser $\Omega_{\mathcal{G}}^{\mathcal{S}}$ avec

$$\begin{cases} \forall a = (t, t') \in \mathcal{A}, \forall (k, k') \in \mathcal{Y}(a), \\ \quad \mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \dfrac{\alpha_a^{\max}(k, k')}{i_a \times N_t^{\mathcal{G}}} \\ \forall t \in \mathcal{T}, \forall k \in \{1, \cdots, \varphi(t)\}, \mathcal{S}\langle t_k, 1\rangle \in \mathbb{R}^+ \\ \Omega_{\mathcal{G}}^{\mathcal{S}} \in \mathbb{R}^+ - \{0\} \end{cases}$$

Ce programme linéaire peut être efficacement résolu par des solveurs standard [59], mais de par sa structure, il peut aussi être transformé en un problème de *Max Cost to Time Ratio* (MCR), ou Cycle de Ratio Coût-Temps Maximum.

En effet, considérons le graphe bi-pondéré $H = (V, E)$ défini comme suit :

- $V = \{t_k, t \in \mathcal{T}, k \in \{1, \cdots, \varphi(t)\}\}$ l'ensemble des nœuds ;
- $E = \{(t_k, t'_{k'}), a = (t, t') \in \mathcal{A}, (k, k') \in \mathcal{Y}(a)\}$ l'ensemble des arcs ; chaque arc $e = (t_k, t'_{k'}) \in E$ est bi-pondéré par

$$(H(e), L(e)) = (d(t_k), -\frac{\alpha_a^{\max}(k, k')}{i_a \times N_t^{\mathcal{G}}}).$$

Soit $\mathcal{C}(H)$ l'ensemble des circuits de H. Pour chaque circuit $c \in \mathcal{C}(H)$, un ordonnancement périodique valide doit alors vérifier

$$0 \geq \sum_{e \in c} H(e) + \Omega_{\mathcal{G}}^{\star} \cdot \sum_{e \in c} L(e).$$

Si l'on note le ratio d'un circuit $c = (e_1, e_2, \cdots, e_p) \in \mathcal{C}(H)$ comme

$$R(c) = \frac{\sum_{i=1}^{p} H(e_i)}{\sum_{i=1}^{p} L(e_i)}.$$

Déterminer la période normalisée minimale $\Omega^{\star}_{\mathcal{G}}$ est alors équivalent à calculer le ratio maximum du graphe, c.-à-d. $\Omega^{\star}_{\mathcal{G}} = \max_{c \in \mathcal{C}(H)} R(c)$.

La Figure 6.8 représente le graphe bi-pondéré H associé au CSDFG de la Figure 6.7 page précédente. Son ratio maximum est égal à 24 et il est atteint par le circuit $c = \{A_2, B_3, C_1\}$ avec $L(c) = \frac{1}{6}$ et $H(c) = 4$. Cela implique que la période minimale d'un ordonnancement périodique valide pour ce CSDFG est $\Omega^{\star}_{\mathcal{G}} = 24$.

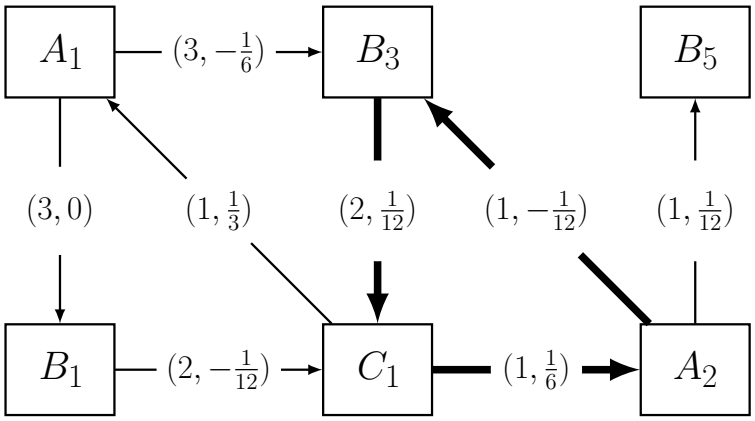


FIGURE 6.8 – Le graphe bi-pondéré $H = (V, E)$ associé au CSDFG de la Figure 6.7 page précédente. Le MCR de ce graphe est atteint par le circuit $c = \{A_2, B_3, C_1\}$; il est égal à $\Omega^{\mathcal{S}}_{\mathcal{G}} = 24$.

Considérant $M = \max_{e \in E}(\max\{H(e), L(e)\})$, un algorithme polynomial dont la complexité théorique est bornée par $\mathcal{O}(|V||E|(\log|V| + \log M))$ existe [44]. Une expérimentation exhaustive des différents algorithmes résolvant le MCR est aussi disponible [30].

L'algorithme d'évaluation du débit que nous proposons ici fournit le débit maximal atteignable par l'ordonnancement périodique d'un CSDFG ; ce débit n'est pas nécessairement optimal pour tout ordonnancement, nous souhaitons alors en évaluer les performances.

6.3.2 Résultats expérimentaux

Le débit maximal périodique d'un CSDFG est une borne inférieure de l'optimal, mais nous ne disposons d'aucune information sur l'écart qui existe entre ces deux solutions. Nous souhaitons alors étudier l'écart qui peut exister entre ces solutions.

D'autre part, l'ordonnancement périodique ayant été envisagé pour sa faible complexité, ces expérimentations devraient nous confirmer de forts écarts de temps de calcul entre ordonnancement périodique et ordonnancement *au plus tôt*.

Pour réaliser ces expérimentations, nous utilisons nos jeux de test ΣC et nous comparons notre méthode avec l'outil d'évaluation du débit des CSDFG fourni par Stuijk et al. [82]. Cet outil s'appuie sur l'exécution symbolique d'un CSDFG.

Tous nos résultats sont résumés sur le Tableau 6.4. La partie supérieure de ce tableau concerne le jeu de test sans borne mémoire. Cette hypothèse est souvent considérée dans les premières étapes de conception, lorsqu'il est nécessaire de connaître les performances maximales d'une application. À cette étape, une solution exacte est préférable, mais cela n'est pas toujours possible (voir l'exemple du *H264 Encoder*).

Dans ce contexte, notre méthode fournie une borne inférieure du débit et la garantie qu'une application est vivante.

	Application	Débit périodique		Débit maximal	
buffers non bornés	BlackScholes	4.75e-09Hz	13ms	4.75e-09Hz	42ms
	Echo	2.45e-11Hz	10ms	2.45e-11Hz	65ms
	JPEG2000	1.51e-11Hz	69ms	1.51e-11Hz	6sec
	Pdetect	5.12e-10Hz	80ms	5.12e-10Hz	216ms
	H264	2.56e-10Hz	544ms	2.56e-10Hz	17j
buffers bornés	BlackScholes	3.06e-09Hz	24ms	3.06e-09Hz	7sec
	Echo	6.84e-12Hz	18ms	2.45e-11Hz	499sec
	JPEG2000	N/S	148ms	-	>60h
	Pdetect	2.56e-10Hz	93ms	1.69e-09Hz	4928sec
	H264	2.39e-10Hz	1sec	-	>60h

N/S : Ordonnancement périodique impossible.

TABLE 6.4 – Évaluation du débit maximal à l'aide de notre méthode périodique et d'un algorithme optimal [82]. La partie haute considère des mémoires non bornées, tandis que la partie basse considère des mémoires dimensionnées par une heuristique gloutonne [80] garantissant la vivacité.

La partie inférieure du Tableau 6.4 concerne les mêmes applications, mais avec des buffers de taille fixe et calculée à l'aide d'une heuristique gloutonne ne cherchant qu'à garantir la vivacité [80].

Dans ce cas, les temps de calcul de SDF3 sont supérieurs. Même s'il peut encore être considéré pour évaluer le débit maximal de certains cas isolés, l'exécution symbolique ne peut plus être utilisée dans un processus d'optimisation itératif.

Quant à l'évaluation du débit maximal périodique, elle demeure particulièrement rapide pour ces applications, principalement parce que la taille d'un l'ordonnancement périodique n'est pas impactée par les contraintes mémoires. Cependant, il existe un écart particulièrement important entre débit périodique et débit optimal. Qui plus est, pour le JPEG2000, on remarque l'absence de solution périodique. Ces résultats confirment le besoin d'améliorer nos méthodes périodiques à l'aide de nouvelles techniques comme les ordonnancements K-périodiques, présentés dans le chapitre 5.

6.4 Dimensionnement mémoire

Après compilation, un buffer entre deux tâches ΣC va être considéré comme un espace mémoire de taille fixe. Il s'agira d'une file circulaire : les écritures se succèdent dans cette file, et lorsqu'une écriture atteint la fin de la zone mémoire, elle continue en début de zone. Les lectures se font de la même manière.

Sans aucune protection, des données peuvent alors être écrasées avant même d'avoir été lues. Heureusement, l'exécutif dispose du modèle CSDFG de l'application et d'un dimensionnement mémoire. Si ce dimensionnement est trop petit, une situation d'interblocage est alors détectée, et l'application sera arrêtée.

Le dimensionnement mémoire d'une application ΣC est donc certainement l'une des étapes les plus cruciales du compilateur AccessCore.

6.4.1 Dimensionnement minimal garantissant la vivacité

Afin d'éviter des situations d'interblocage, la chaîne de compilation AccessCore va produire un premier dimensionnement garantissant la vivacité de l'application. L'algorithme qui est utilisé aujourd'hui n'est pas optimal ; il s'agit d'une heuristique gloutonne basée sur une exécution symbolique et inspirée par [80]. Nous savons que l'exécution symbolique peut être particulièrement longue pour les dataflows les plus complexes.

Pour ces raisons, nous avons proposé une nouvelle méthode de dimensionnement minimal des CSDFG. Elle s'appuie sur les condition SC1 et SC2 de Benazouz [11] et prend la forme d'un programme linéaire en nombre entier. Pour le résoudre, nous effectuons une relaxation du problème.

Détail de la méthode

En plus de vérifier la vivacité d'un CSDFG, la condition SC1 peut aussi être utilisée pour produire un dimensionnement mémoire minimal garantissant la vivacité d'un CSDFG.

En effet, supposons que chaque buffer $b(a)$ associé à un arc $a = (t, t')$ soit borné. Comme suggéré dans le chapitre 3, cette contrainte peut alors être modélisée en utilisant un arc retour $a' = (t', t)$.

Maintenant, notons $\mathcal{G}' = (\mathcal{T}, \mathcal{A}')$ le graphe obtenu en ajoutant ces arcs retour. Comme la capacité d'un buffer $b(a)$ est égale à $M_0(a) + M_0(a')$, et que chacun des marquages initiaux $M_0(a)$ sont déjà fixés, pour dimensionner le CSDFG il est donc nécessaire de calculer les marquages initiaux $M_0(a')$ des arcs retour de sorte que le graphe $\mathcal{G}'$ soit vivant. Ainsi, cela revient à vérifier que $\mathcal{G}'$ respecte la condition SC1.

Notons $H'_1 = (V'_1, E'_1)$ le graphe bi-pondéré associé à la condition SC1 pour le graphe $\mathcal{G}'$. Pour tout arc $e = (t_k, t'_{k'}) \in E_1$ correspondant à un arc $a = (t, t') \in \mathcal{A}$, alors on note $W'_1(e)$ le coût de cet arc.

$$W'_1(e) = O_a\langle t'_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - \text{step}_a - M_0(a)$$

Si H'_1 ne contient que des circuits c tels que la somme $\sum_{e\in c} W'_1(e)$ est strictement négative, alors SC1 est vérifié.

Pour vérifier cette condition on associe à chaque sommet $t_k \in V'_1$ un entier γ_{t_k} tels que $\gamma_{t_k} - \gamma_{t'_{k'}} > W'_1(e)$ pour tout arc $e = (t_k, t'_{k'}) \in E_1$. Les valeurs γ_{t_k} ne pourront alors être définies que si et seulement si pour tout circuit du graphe $\sum_{e\in c} W'_1(e) < 0$.

Le système correspondant est alors

Minimiser $\sum_{a\in\mathcal{A}'} M_0(a)$ avec

$$\begin{cases} \gamma_{t_k} - \gamma_{t'_{k'}} > W'_1(e), & \forall e = (t_k, t'_{k'}) \in E_2 \\ M_0(a) \in \mathbb{N}, & \forall a \in \mathcal{A}' \\ \gamma_{t_k} \in \mathbb{R}, & \forall t \in \mathcal{T}, \forall k \in \{1, \cdots, \varphi(t)\} \end{cases}$$

La résolution de ce programme linéaire en nombre entier est réalisée par relaxation avec l'outil open source GLPK [59] ; les tailles de buffer sont ensuite calculées par arrondie.

Notons enfin qu'un programme équivalent existe pour résoudre le dimensionnement mémoire minimal avec la condition SC2.

Expérimentations

Les résultats expérimentaux obtenus avec cette méthode de dimensionnement sont comparés avec l'algorithme en place dans la chaîne de compilation AccessCore ; ils sont visibles sur le Tableau 6.5 page suivante.

La première colonne indique le nom de l'application concernée, les deux suivantes rapportent le temps de calcul et la taille mémoire calculée par la méthode de dimensionnement

la plus rapide entre SC1 et SC2. Les deux dernières donnent le temps de calcul et la taille de buffer obtenus avec l'algorithme glouton.

	Dimensions SC1/SC2		Dimensions SE	
Application	durée	taille mem.	durée	taille mem.
BlackScholes	**8 ms**	16 KB	9 ms	16 KB
JPEG2000	3089 ms	3807 KB	**2055 ms**	**3651 KB**
Echo	**5 ms**	**28 KB**	315 ms	52 KB
Pdetect	**26 ms**	3959 KB	61 ms	3959 KB
H264	4808 ms	1368 KB	**937 ms**	1368 KB
autogen1	**169 ms**	**1849 KB**	3043 ms	2009 KB
autogen2	**1704 ms**	**227 MB**	7 min	244 MB
autogen3	**2407 ms**	**1080 MB**	36 min	1296 MB
autogen4	**16605 ms**	47 KB	20522 ms	**34 KB**
autogen5	**2 min**	**1555 KB**	3 min	3069 KB

TABLE 6.5 – Experimentation de différentes méthodes de dimensionnement minimal garantissant la vivacité.

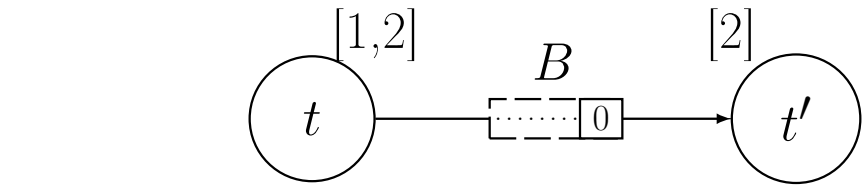
À en juger par ces données, bien qu'une avance soit donnée à la condition SC1/SC2, la qualité des deux méthodes est comparable. Pour des applications complexes, le temps de calcul de l'algorithme glouton reste cependant bien plus élevé.

Néanmoins, il est important de rappeler que tous les CSDFG n'admettent pas d'ordonnancement périodique. Dans ce cas, notre algorithme ne sera pas en mesure de fournir une solution de dimensionnement.

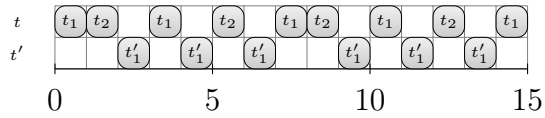
6.4.2 Dimensionnement minimal sous contrainte de débit

Après les premières phases de conception d'une application ΣC, il est courant que le dimensionnement minimal ne permettant pas d'atteindre le débit maximal d'une application.

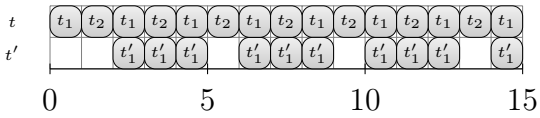
Un exemple simple de cette situation est visible sur la Figure 6.9 page suivante. Il s'agit de deux ordonnancements *au plus tôt* pour une même application, mais l'un dispose d'une quantité de mémoire bornée à trois jetons, tandis que pour l'autre cette quantité est limitée à quatre jetons.



(a) un simple CSDFG



(b) 3 jetons maximum



(c) 4 jetons maximum

FIGURE 6.9 – (b) et (c) sont des ordonnancements *au plus tôt* du CSDFG (a), mais avec des tailles mémoire différentes.
Pour le premier, le buffer B est borné à trois jetons maximum tandis que pour le deuxième cette limite est de quatre jetons.

On s'aperçoit alors que la période normalisée du premier ordonnancement est de 7 unités de temps tandis que celle du deuxième est de 4 unités de temps. Pour éviter cette flagrante perte de performance, un dimensionnement avec contrainte de débit doit être calculé.

En nous aidant des résultats sur les ordonnancements périodiques du chapitre 4, nous présentons deux algorithmes de dimensionnement minimal avec contrainte de débit, PSizing-ILP et PSizing-LP. Le premier fournit une solution optimale du dimensionnement avec contrainte de débit pour un ordonnancement périodique, et le second ne fournit qu'une solution approchée de ce problème.

Nous comparons par la suite ces algorithmes avec la méthode du Min-Max proposée par Benazouz et Munier-Kordon [12]. Cependant, nous n'avons pas été en mesure de comparer nos résultats avec SDF3 [81], son temps de calcul dépassant l'ordre de la semaine.

Définition des algorithmes PSizing-LP et PSizing-ILP

Soit le CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Chaque buffer $b(a)$ pour $a \in \mathcal{A}$ est supposé borné. On considère donc $\mathcal{G}' = (\mathcal{T}, \mathcal{A}')$ le graphe obtenu en ajoutant des arcs retour. Par souci de clarté, on note $Fb(\mathcal{A})$ l'ensemble de ces arcs retours, $\mathcal{A}' = \mathcal{A} \cup Fb(\mathcal{A})$. On rappelle que la taille d'un buffer $b(a)$ associé à un couple d'arcs $(a, a') \in \mathcal{A} \times Fb(\mathcal{A})$ est égale à $M_0(a) + M_0(a')$. Et l'on considère enfin la période $\Omega_{\mathcal{G}}^{\mathcal{S}}$ fixée à l'avance.

D'après le théorème 6 page 65, déterminer un dimensionnement minimal garantissant l'existence d'un ordonnancement périodique dont la période $\Omega_{\mathcal{G}}^{\star}$ est fixée peut alors se modéliser par le système suivant :

Minimiser $\sum_{a \in Fb(\mathcal{A})} M_0(a)$ avec

$$\begin{cases} \forall a = (t, t') \in \mathcal{A}, \forall (k, k') \in \mathcal{Y}(a), \\ \quad \mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \dfrac{\alpha_a^{\max}(k, k')}{i_a \times N_t^{\mathcal{G}}} \\ \forall a' = (t, t') \in Fb(\mathcal{A}), \forall (k, k') \in \mathcal{Y}(a'), \\ \quad \mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \dfrac{\alpha_{a'}^{\max}(k, k')}{i_{a'} \times N_t^{\mathcal{G}}} \\ \forall t \in \mathcal{T}, \forall k \in \{1, \cdots, \varphi(t)\}, \mathcal{S}\langle t_k, 1\rangle \in \mathbb{R}^+ \end{cases}$$

Pour résoudre ce système, on s'aperçoit que deux classes de contraintes doivent alors être considérées.

- Pour tous les arcs $a \in \mathcal{A}$, $M_0(a)$ est fixé. Ainsi pour tout couple de valeur $(k, k') \in \mathcal{Y}(a)$, $\alpha_a^{\max}(k, k')$ est connu.
- Pour tous les arcs $a' = (t, t') \in Fb(\mathcal{A})$, $M_0(a')$ est une variable du problème et $\alpha_{a'}^{\max}(k, k')$ l'est aussi. L'ensemble $\mathcal{Y}(a')$ est donc inconnu. Si l'on considère l'ensemble $\mathcal{K}(a') = \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$, alors tous les couples $(k, k') \in \mathcal{K}(a')$ devront donc être considérés.

D'autre part, les expressions de $\alpha_{a'}^{\max}(k, k')$ ne sont pas linéaires, corrigeons cela.

On remarque que pour tout entier $u \in \mathbb{Z}$,

$$u - \gcd_a + 1 \leq \lfloor u \rfloor^{\gcd_a} \leq u.$$

$\lfloor u \rfloor^{\gcd_a}$ est alors la plus petite valeur à la fois supérieure à $u - \gcd_a + 1$ et divisible par $\gcd_a$.

Notons $u = O_a\langle t_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - M_0(a) - 1$, on obtient alors

$$\alpha_a^{\max}(k, k') = \lfloor u \rfloor^{\gcd_a} \geq u - \gcd_a + 1.$$

Notre problème d'optimisation peut maintenant être formulé comme un système linéaire en nombre entier minimisant la somme des jetons et noté Σ :

Minimiser $\sum_{a \in Fb(\mathcal{A})} M_0(a)$ avec

$$\begin{cases} \forall a \in \mathcal{A}, \forall (k,k') \in \mathcal{Y}(a), \\ \quad \mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \dfrac{\alpha_a^{\max}(k,k')}{i_a \times N_t^{\mathcal{G}}} \\ \forall a \in Fb(\mathcal{A}), \forall (k,k') \in \mathcal{K}(a), \\ \quad \mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \dfrac{f_a(k,k') \times \mathrm{gcd}_a}{i_a \times N_t^{\mathcal{G}}} \\ \quad u_a(k,k') = O_a\langle t_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - M_0(a) - 1 \\ \quad f_a(k,k') \times \mathrm{gcd}_a \geq u_a(k,k') - \mathrm{gcd}_a + 1 \\ \quad f_a(k,k') \in \mathbb{N}, u_a(k,k') \in \mathbb{N} \\ \forall a \in Fb(\mathcal{A}), M_0(a) \in \mathbb{N}^+ \\ \forall t \in \mathcal{T}, \forall k \in \{1, \cdots, \varphi(t)\}, \mathcal{S}\langle t_k, 1\rangle \in \mathbb{R}^+ \end{cases}$$

Les premières inégalités du système expriment les contraintes associées à tout arc $a \in \mathcal{A}$ et comme $\alpha_a^{\max}(k,k')$ est connu, le terme droit de l'inégalité est une constante. Ce n'est plus vrai pour $a \in Fb(\mathcal{A})$. Toutes les valeurs $\alpha_a^{\max}(k,k')$ vont être alors remplacées par $f_a(k,k') \times \mathrm{gcd}_a$. D'après la définition de $u_a(k,k')$, $f_a(k,k') \times \mathrm{gcd}_a$ est alors la plus petite valeur supérieure ou égale à $u_a(k,k') - \mathrm{gcd}_a + 1$ et divisible par gcd_a. Et comme $\mathcal{Y}(a)$ dépend de $M_0(a)$, cet ensemble ne peut être évalué lorsque $a \in Fb(\mathcal{A})$. Tous les couples appartenant à $\mathcal{K}(\mathcal{A})$ doivent alors être considérés.

Nous proposons deux algorithmes pour résoudre Σ : le plus simple, appelons-le PSizing-ILP, consiste à résoudre Σ directement en utilisant un solveur linéaire en nombre entier. Une solution optimale du problème est fournie, il s'agit du dimensionnement minimal garantissant le débit fixé pour un ordonnancement périodique.

La résolution de ce programme linéaire est complexe, et pour de grandes instances, son temps de calcul est trop élevé. Pour pallier cette limite, nous proposons aussi PSizing-LP, une méthode qui résout un système Σ relaxé où les variables $f_a(k,k')$, $u_a(k,k')$ et $M_0(a)$ sont rationnelles.

Une instance réalisable du problème peut alors être obtenue en définissant pour chaque arc $a \in Fb(\mathcal{A})$ les deux fonctions

$$\begin{aligned} \widehat{f_a}(k,k') &= \lfloor f_a(k,k') \rfloor, \\ \widehat{M_0}(a) &= \max_{(k,k') \in \mathcal{K}(a)} (O_a\langle t_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - gcd_a - \widehat{f_a}(k,k') \times \mathrm{gcd}_a). \end{aligned}$$

Pour tout couple $(k, k') \in \mathcal{K}(a)$ l'inégalité

$$\widehat{f_a}(k, k') \times \gcd_a \geq O_a\langle t_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - gcd_a - \widehat{M_0}(a)$$

garantie alors que la solution approchée obtenue est valide.

Résultats expérimentaux

Dans cette section, nous commençons par évaluer la pertinence de nos deux algorithmes de dimensionnement dans un contexte industriel. Nos résultats sont comparés avec l'algorithme Min-Max proposé par Benazouz et Munier-Kordon [12] et l'algorithme optimal de Stuijk et al. [82]. Les programmes linéaires de PSizing-LP et de l'algorithme Min-Max sont résolus en utilisant le solveur open source CLP (Coin-or linear programming) [58] ; les programmes linéaires en nombre entier de PSizing-ILP sont résolus à l'aide du solveur Gurobi Optimizer [49]. Ce dernier est un produit payant, mais il nous permet de résoudre ces problèmes dans un temps raisonnable. La comparaison avec l'algorithme optimal de Stuijk et al. [82] fut réalisée en utilisant l'implémentation publique de SDF3 [81] au travers de l'exécutable `sdf3analysis-csdf`.

Dans un second temps, nous exploitons ces méthodes pour générer le front de Pareto d'une application particulière, le JPEG2000. Dans cette expérience, nous souhaitons visualiser les performances de nos méthodes de dimensionnement en fonction du débit requis. Afin de compléter cette étude, nous les avons aussi comparées à des algorithmes jugés moins performants, le Burst et le Average [13].

Dimensionnement mémoire avec contrainte de débit Le Tableau 6.6 page suivante résume nos premières expérimentations sur le dimensionnement mémoire avec trois contraintes de débit différentes. Les deux premières colonnes indiquent la performance des méthodes Min-Max [12] et PSizing-LP. La dernière colonne concerne la solution entière PSizing-ILP.

Avant toute chose, une période de fonctionnement minimale a été calculée en utilisant l'algorithme d'évaluation du débit périodique de la section précédente (on note cette période $\Omega^\star_\mathcal{G}$). Des contraintes de débit ont ensuite été fixées successivement à $10 \times \Omega^\star_\mathcal{G}$, $2 \times \Omega^\star_\mathcal{G}$ et $\Omega^\star_\mathcal{G}$. Pour chaque algorithme, le temps de calcul des trois contraintes est fourni.

Application	Min-Max		PSizing-LP		PSizing-ILP	
BlackScholes	16332 16332 22572	77ms	16332 16332 22572	155ms	16332 16332 22572	120ms
Echo	28098 28101 28115	40ms	28098 28101 28115	79ms	28098 28101 28113	60ms
H264	aucun résultat aucun résultat aucun résultat	3sec	1369256 1369256 1369271	5sec	1369257 1369256 1369271	32sec
JPEG2000	3936365 4027536 4153356	1sec	3635687 3733367 3864471	2sec	3502087 3600295 3725351	5min
Pdectect	4264311 4327687 5375166	5sec	3959031 4123187 5191721	223sec	3958311 3958351 5068006	10h

TABLE 6.6 – Experimentation du dimensionnement mémoire pour trois contraintes de débit ($10 \times \Omega^\star_{\mathcal{G}}$, $2 \times \Omega^\star_{\mathcal{G}}$ et $\Omega^\star_{\mathcal{G}}$).

L'algorithme Min-Max reste toujours le plus rapide ; ce résultat (prévisible) est dû au faible nombre de contraintes considérées dans le programme linéaire généré pour le Min-Max. L'écart de temps avec notre méthode est particulièrement élevé pour l'application Pdetect ; c'est elle qui compte le plus grand nombre de phases (4045 phases pour 58 tâches).

Néanmoins, on remarque que la méthode PSizing-LP fournit dans de nombreux cas des solutions de dimensionnement plus précises que le Min-Max. En fixant localement les exécutions de phases, le Min-Max n'est par exemple plus capable de traité l'application H264, alors qu'un ordonnancement périodique existe (il est reconnu par notre méthode).

Pour finir, notons que l'écart de performance entre les solutions de PSizing-LP et PSizing-ILP reste toujours inférieur à 10%. La relaxation choisie semble donc particulièrement efficace au regard des gains sur le temps de calcul.

Génération d'un front de Pareto Lorsque l'évaluation du dimensionnement est répétée pour différentes contraintes de débit, un front de Pareto approché peut être généré. Dans cette section, nous nous intéressons à la forme de ce front pour l'application JPEG2000 avec les algorithmes Min-Max, PSizing-LP et PSizing-ILP.

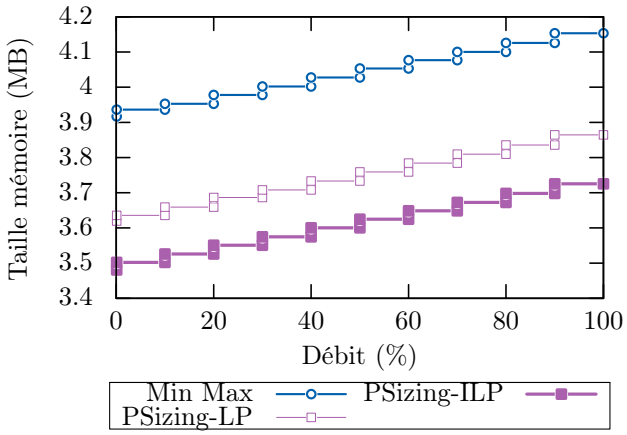


FIGURE 6.10 – Approximation d'un front de Pareto Fréquence/Mémoire pour l'application JPEG2000.

Sur le front de Pareto de la Figure 6.10 l'abscisse correspond au débit (en pourcentage par rapport à l'optimal période), et l'ordonnée indique la quantité de mémoire nécessaire pour atteindre ce débit. Onze points de tests allant de 0 à 100% sont considérés.

Les meilleures solutions sont toujours obtenues par l'algorithme PSizing-ILP, lequel calcule la taille mémoire optimale pour un ordonnancement périodique.

L'algorithme PSizing-LP s'est lui avéré toujours plus performant que l'algorithme Min-Max, ce qui confirme nos expérimentations précédentes. On remarque aussi que cet écart ne semble pas dépendre du débit requis.

Nous regrettons néanmoins de ne pas disposer des dimensionnements optimaux. En effet, toutes ces méthodes sont basées sur des ordonnancements périodiques et la croissance linéaire de ces courbes reste surprenante. Nous sommes persuadés que le front de Pareto optimal pour cette application aurait une forme très différente.

Pour conclure, notons que chacune des méthodes étudiées ici dispose de ses propres avantages. L'algorithme Min-Max est la technique la plus rapide. Pour atteindre de meilleures solutions, tout en restant dans un temps de calcul raisonnable, PSizing-LP peut être envisagée. Ces deux méthodes sont rapides, et les solutions qu'elles proposent sont cohérentes ; leur intégration dans une chaîne de compilation industrielle comme AccessCore est donc tout à fait envisageable.

Enfin, l'outil de dimensionnement SDF3 et PSizing-ILP sont deux techniques non polynomiales, mais elles nous permettent d'atteindre des solutions plus précises.

Conclusion

Nous avons étudié ici l'application des ordonnancements périodiques à l'analyse statique des CSDFG. Trois cas d'intégration furent considérés : l'étude de la vivacité, l'évaluation du

débit maximal et le dimensionnement mémoire. Pour chacun de ces cas, nous avons proposé des algorithmes originaux et nous avons évalué expérimentalement leurs performances. Des expérimentations qui furent d'ailleurs réalisées en partie à l'aide de notre générateur d'instances aléatoires.

Dans ces résultats, on constate deux effets imputables aux ordonnancements périodiques :

- Ces algorithmes sont rapides, bien plus rapides que les méthodes exactes.
- Et ils sont approchés, et peuvent parfois fournir des solutions inadaptées ou insuffisantes.

Pour remédier à cela, dans le chapitre 5, nous avions suggéré d'étudier les ordonnancements K-périodiques. Au travers de quelques expérimentations, nous avons confirmé l'intérêt de ce choix.

Plusieurs de ces algorithmes sont aujourd'hui en production dans la chaîne de compilation AccessCore. Cependant, pour réaliser avec succès cette intégration, les différentes contributions des chapitres 4 et 5 ont été révisées. En effet, le langage ΣC évolue continuellement, et certaines de ses fonctionnalités ont entrainé des modifications du modèle initial. Dans le chapitre suivant, nous étudions ces nouvelles fonctionnalités ainsi que les fonctionnalités futures.

Chapitre 7

Support du langage ΣC

Introduction

Au cours de ces trois dernières années, la chaîne de compilation AccessCore a beaucoup évolué. Le langage ΣC s'est vu accordé de nouvelles fonctionnalités, et le modèle dataflow qui lui est associé a aussi changé. La modélisation dataflow d'une application ΣC est désormais plus proche d'un modèle CSDFG auquel l'on aurait appliqué des seuils d'exécution similaires à ceux des *Computation Graphs*. Qui plus est, d'autres améliorations du langage sont en cours d'étude, des phases d'initialisation sont par exemple envisagées.

Afin d'intégrer nos différentes contributions à la chaîne de compilation AccessCore, il est nécessaire d'étendre nos méthodes d'analyse à ces modèles plus expressifs.

Dans ce chapitre, nous étudions deux nouvelles fonctionnalités, le seuil d'exécution et les phases d'initialisation. Nous commençons par présenter les *Computation Graphs* et l'impact du seuil sur l'ordonnançabilité de ce modèle (en comparaison avec les SDFG). Nous généralisons ensuite les travaux d'ordonnancement périodique du chapitre 4 aux *Phased Computation Graph*. Pour cela, nous étudions deux modèles particuliers : les *Thresholded CSDFG*, un modèle CSDFG disposant de seuils d'exécution distincts de ses taux de consommation ; et les *Initialized CSDFG*, un modèle CSDFG dont certaines tâches disposent de phases préliminaires ne s'exécutant qu'une fois seulement.

7.1 Les seuils d'exécution

Le seuil d'exécution précise une quantité de données nécessaire à l'exécution d'une tâche, pour chacun de ses buffers d'entrée. Lorsqu'il est défini, le seuil d'exécution se distingue alors de la quantité de données effectivement consommées par la tâche. Dans la suite de cette section, nous considérons le modèle *Thresholded CSDFG* (TCSDFG), un modèle CSDFG avec des seuils d'exécution.

Karp et Miller [51] ont défini le modèle *Computation Graph* avec des seuils d'exécution (appelés *Threshold*). On retrouve aussi les seuils d'exécution dans le langage StreamIt [86] au travers de la fonction `peak`. Thies et al. [87] proposent aussi le modèle *Phased Computation Graphs* afin d'exprimer une extension des *Cyclo-Static Dataflow Graph* avec, entre autres, des seuils d'exécutions.

Le principal atout du seuil est qu'il permet d'exprimer naturellement une *fenêtre de lecture glissante*. Une fenêtre glissante est un motif d'accès où une partie des données lues pendant une exécution peuvent être lus à nouveau durant exécution suivante. Sans seuil, pour effectuer une fenêtre de lecture glissante, une tâche doit conserver en mémoire un historique de ses lectures. Ce contournement est valide, mais il signifie aussi que cette tâche doit disposer d'*un état* (une zone mémoire qui lui est propre). Certains modèles

(comme StreamIt) autorisent la réentrance des tâches, et permettent ainsi d'augmenter le parallélisme d'une application en effectuant des exécutions concurrentes d'une même tâche. Une tâche avec un état ne peut pas être réentrante. D'autre part, dans certains contextes applicatifs, il est tout simplement impossible qu'une tâche ait un état.

Comme le langage ΣC précise aussi des seuils d'exécutions (appelés *caches*), nous souhaitons adapter nos méthodes d'analyse aux TCSDFG. Denolf et al. [34] fournit alors une transformation des TCSDFG, vers un modèle CSDFG classique. Cependant, cette transformation se limite à des cas particuliers de TCSDFG où toutes les données lues à la fin de l'itération d'une tâche doivent avoir été consommées. Formellement, pour un buffer $a = (t, t')$, cette condition s'écrit :

$$\forall k \in \varphi(t) \quad o_a \leq O_{Pr\langle t_k,1\rangle} + thr_a(k).$$

La transformation de Denolf et al. [34] n'est alors pas suffisante pour adapter nos techniques aux TCSDFG. Dans la suite de cette section, nous allons étudier les seuils d'exécution au travers des *Computation Graphs*. Nous adaptons ensuite nos contributions aux TCSDFG.

7.1.1 *Computation Graph*

Les *Computation Graphs* (CG) sont une extension des *Synchronous Dataflow Graphs* où la condition d'exécution d'une tâche ne dépend pas seulement de la quantité de données consommée par cette tâche. Pour un buffer $a = (t, t')$, en plus du taux de production in_a et du taux de consommation out_a, le modèle CG définit un seuil d'exécution thr_a, la quantité de données nécessaires à l'exécution de la tâche t'. Cette caractéristique a une influence sur l'ordonnançabilité d'un graphe, et plus précisément sur les relations de précédence induites par ses buffers. Nous étudions ces changements en comparaison avec les SDFG, puis nous présentons une transformation valide d'un CG vers un SDFG avec marquage négatif.

Influence du seuil sur la définition du vecteur de répétition

Le vecteur de répétition d'un SDFG est défini comme le nombre minimal d'exécutions nécessaires pour qu'un graphe puisse retrouver son état initial. Il se calcule à partir des productions et des consommations de ses tâches et son existence atteste de la consistance d'un SDFG.

Le calcul s'applique toujours aux CG, mais la définition initiale n'est plus vérifiée. En effet, certains CG ne sont pas en mesure de retrouver leur état initial au cours de leur exécution. C'est le cas du CG de la Figure 7.1 page suivante.

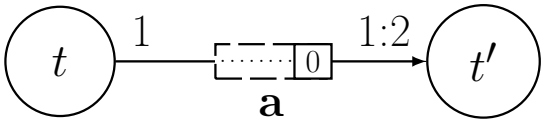


FIGURE 7.1 – Un exemple de *Computation Graph*, on note $in_a = out_a = 1$ et $thr_a = 2$.

Pour cet exemple, deux exécutions de la tâche t sont nécessaires avant de pouvoir exécuter la tâche t' une première fois. Et comme la tâche t' requièrt toujours au moins un jeton supplémentaire pour s'exécuter, on s'aperçoit qu'il est impossible pour le buffer a de retrouver son état initial.

Étant donné que le vecteur de répétition atteste de la consistance d'un graphe, nous pouvons toujours calculer le vecteur de répétition d'un CG. Mais ses propriétés sont différentes.

Lemme 9 *Soit un CG consistant $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. Considérons un buffer $a = (t, t')$ dont le marquage courant est noté $M(a)$. Il existe un ordre d'exécution des tâches t et t' permettant de retrouver ce marquage courant, si et seulement s'il vérifie*

$$M(a) \geq thr_a - out_a.$$

PREUVE : Considérons un marquage $M(a)$.

- Comme l'exécution de la tâche t produit toujours des jetons, la quantité de jetons dans le buffer a après une exécution de t sera toujours strictement supérieure à $M(a)$.
- D'autre part, chaque exécution de la tâche t' requièrt thr_a jetons, mais n'en consomme que out_a. Après une exécution de cette tâche, la quantité de jetons dans le buffer a sera toujours au minimum de $thr_a - out_a$ jetons.

On s'aperçoit donc, quelle que soit la tâche exécutée, que la quantité de jetons dans le buffer a sera au moins supérieure ou égale à

$$\max\{M(a) + 1, thr_a - out_a\}.$$

Alors si $M(a) < thr_a - out_a$, la quantité de jetons dans le buffer après l'exécution d'une tâche sera toujours strictement supérieure à $M(a)$.

Réciproquement, considérons que la quantité de jetons dans le buffer vérifie

$$M(a) \geq thr_a - out_a.$$

Comme le graphe est consistant, nous savons qu'une série d'exécutions successives de $N_t^{\mathcal{G}}$ exécution de la tâche t et $N_{t'}^{\mathcal{G}}$ exécutions de la tâche t', nous permet de revenir à l'état de marquage courant. ∎

Partant du lemme 9 page ci-contre, bien que le vecteur de répétition ne caractérise pas la séquence minimale d'exécution nécessaire pour retrouver l'état initial d'un CG, il permet de retrouver un état de marquage si pour l'ensemble des buffers la condition de lemme 9 page précédente est respectée.

Ce résultat nous permet de déterminer un ordonnancement *au plus tôt* de taille minimale pour un CG, et de mettre en œuvre la plupart des techniques d'analyse basées sur une exécution symbolique du graphe comme pour les SDFG.

Voyons désormais comment généraliser aux CG l'ordonnancement périodique considéré par Benabid et al. [10].

Caractérisation d'une relation de précédence

Nous commençons ici par définir une condition d'existence nécessaire et suffisante d'une relation de précédence entre deux exécutions pour les *Computation Graphs*.

Lemme 10 *Dans un CG, un buffer $a = (t, t')$ induit une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si :*

$$\max(0, in_a - out_a) \leq M_0(a) + n \cdot in_a - n' \cdot out_a - thr_a + out_a < in_a$$

PREUVE : D'après la définition 6 page 59, il existe une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si

1. $\langle t', n'\rangle$ peut s'exécuter après $\langle t, n\rangle$;

$$M_0(a) + n \cdot in_a - (n' - 1) \cdot out_a \geq thr_a$$

2. $\langle t', n' - 1\rangle$ peut s'exécuter avant $\langle t, n\rangle$;

$$M_0(a) + (n - 1) \cdot in_a - (n' - 2) \cdot out_a \geq thr_a$$

3. $\langle t', n'\rangle$ ne peut pas s'exécuter avant $\langle t, n\rangle$.

$$M_0(a) + (n - 1) \cdot in_a - (n' - 1) \cdot out_a < thr_a$$

La combinaison de ces inéquations prouve le lemme. ∎

Cette condition va maintenant nous permettre d'étendre la notion de jeton utile aux CG.

Notion de jetons utiles

La notion de jeton utile d'un SDFG est une transformation de graphe simplifiant son marquage sans influence sur les relations de précédence induites par ses buffers. Elle identifie, pour chaque buffer a, une valeur $\gcd_a$ telle que la quantité de données dans un buffer soit toujours multiple de $\gcd_a$. Nous étendons cette propriété aux CG.

Soit $\gcd_a = \gcd(in_a, out_a)$, le plus grand commun diviseur de in_a et out_a.

Théorème 14 *Considérons un CG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, et le buffer $a = (t, t') \in \mathcal{A}$. Notons maintenant $R_{M_0(a)}$ le reste de la division euclidienne du marquage initial $M_0(a)$ par gcd_a et R_{thr_a} le reste de la division euclidienne du seuil thr_a par gcd_a. Si on note*

- *$M_0^*(a) = \lfloor M_0(a) \rfloor_{gcd_a}$ le marquage utile*
- *et thr_a^* le seuil de consommation utile tels que*

$$thr_a^* = \begin{cases} \lfloor thr_a \rfloor_{gcd_a} & si\ R_{M_0(a)} \geq R_{thr_a} \\ \lceil thr_a \rceil_{gcd_a} & sinon \end{cases}$$

On peut alors remplacer simultanément $M_0(a)$ et thr_a par les valeurs $M_0^*(a)$ et thr_a^* sans modifier les relations de précédence induites par le buffer a.

Preuve :

S'il existe une relation de précédence entre deux exécutions $\langle t, n \rangle$ et $\langle t', n' \rangle$ alors

$$in_a > M_0(a) + n \cdot in_a - n' \cdot out_a - thr_a + out_a \geq \max\{0, in_a - out_a\}.$$

Si $R_{M_0(a)} \geq R_{thr_a}$ on note $R_1 = R_{M_0(a)} - R_{thr_a}$,

$$\begin{aligned} in_a > \quad & M_0^*(a) + R_{M_0(a)} + n \cdot in_a - n' \cdot out_a - thr_a^* - R_{thr_a} + out_a \quad \geq \max\{0, in_a - out_a\} \\ in_a > \quad & M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + R_1 + out_a \quad \geq \max\{0, in_a - out_a\} \end{aligned}$$

D'une part

$$in_a > M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + out_a$$

est vérifié, et comme $R_{thr_a} \leq R_{M_0(a)} < gcd_a$ alors $0 \leq R_1 < gcd_a$. Ainsi, comme $\max\{0, in_a - out_a\}$ et $M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + out_a$ sont divisible par $\gcd_a$ alors

$$M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + out_a \geq \max\{0, in_a - out_a\}.$$

Dans le cas contrainre, si $R_{M_0(a)} < R_{thr_a}$ on note $R_2 = R_{thr_a} - R_{M_0^*(a)}$.

$$\begin{aligned} in_a > \quad & M_0^*(a) + R_{M_0(a)} + n \cdot in_a - n' \cdot out_a - thr_a^* - R_{thr_a} + \gcd_a + out_a \quad \geq \max\{0, in_a - out_a\} \\ in_a > \quad & M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + \gcd_a - R_2 + out_a \quad \geq \max\{0, in_a - out_a\} \end{aligned}$$

D'une part

$$in_a > M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + out_a$$

est vérifié, et d'autre part comme $R_{M_0(a)} < R_{thr_a} < \gcd_a$ alors $0 < R_2 < \gcd_a$ et donc $\gcd_a > \gcd_a - R_2 > 0$. Ainsi, comme $\max\{0, in_a - out_a\}$ et $M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + out_a$ sont multiples de $\gcd_a$ alors

$$M_0^*(a) + n \cdot in_a - n' \cdot out_a - thr_a^* + out_a \geq \max\{0, in_a - out_a\}$$

∎

La propriété de jeton utile pour un CG nous permet maintenant de caractériser l'ordonnancement périodique d'un CG.

Ordonnancement périodique

De la même manière que Benabid et al. [10] pour les SDFG, nous sommes maintenant en mesure de caractériser une condition nécessaire et suffisante pour que l'ordonnancement périodique d'un CG soit valide. Premièrement, nous caractérisons une condition nécessaire et suffisante d'existence d'une relation de précédence. Nous définissons ensuite une contrainte vérifiant la validité de l'ordonnancement périodique d'un CG.

Lemme 11 *Considérons un buffer $a = (t, t')$. Il existe une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ si et seulement si il existe un entier positif k tel que*

$$k_{min}(a) \leq k \cdot gcd_a \leq k_{max}(a),$$

avec

$$\begin{aligned} k \cdot gcd_a &= n \cdot in_a - n' \cdot out_a \\ k_{min}(a) &= \max\{0, in_a - out_a\} - M_0(a) + thr_a \\ k_{max}(a) &= in_a - M_0(a) + thr_a - gcd_a. \end{aligned}$$

PREUVE : Prouvons tout d'abord que s'il existe une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ alors

$$k_{min}(a) \leq n \cdot in_a - n' \cdot out_a \leq k_{max}(a).$$

Comme $\gcd_a = gcd(out_a, in_a)$, pour tout couple $(n, n') \in \mathbb{N}^2$, il existe $k \in \mathbb{Z}$ tel que

$$n \cdot in_a - n' \cdot out_a = k \times \gcd_a.$$

Si une relation de précédence existe entre $\langle t, n\rangle$ et $\langle t', n'\rangle$, d'après le lemme 10 page 123

$$in_a - M_0(a) + thr_a > n \cdot in_a - n' \cdot out_a \geq \max\{0, in_a - out_a\} - M_0(a) + thr_a.$$

Comme le théorème 14 page 124 prouve que $M_0(a)$ et thr_a peuvent être multiples de $gcda$ sans influence sur les relations de précédence,

$$in_a - M_0(a) + thr_a - \gcd_a \geq k \times \gcd_a \geq \max\{0, in_a - out_a\} - M_0(a) + thr_a$$

On obtient bien $k_{min}(a) \leq n \cdot in_a - n' \cdot out_a \leq k_{max}(a)$.

Réciproquement, il existe toujours $(x, y) \in \mathbb{Z}^2$ tels que $x \cdot in_a - y \cdot out_a = \gcd_a$. Alors pour toute valeur $k \times \gcd_a \in \{k_{min}(a), \cdots, k_{max}(a)\}$, et pour tout entier $q \geq 0$, le couple d'entiers $(n, n') = (k \cdot x + q \cdot out_a, k \cdot y + q \cdot in_a)$ vérifie $in_a \cdot n - out_a \cdot n' = k \times \gcd_a$. Le buffer b induit donc une relation de précédence entre $\langle t, n\rangle$ et $\langle t', n'\rangle$ ce qui conclut cette preuve. ∎

Partant de cette condition d'existence d'une relation de précédence, nous sommes en mesure d'exprimer le théorème suivant.

Théorème 15 *Un ordonnancement périodique $\mathcal{S}$ est un ordonnancement périodique valide pour le* Computation Graph $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ *si et seulement si pour tout buffer $a \in \mathcal{A}$,*

$$\mathcal{S}\langle t', 1\rangle - \mathcal{S}\langle t, 1\rangle \geq d(t) + \Omega_{\mathcal{G}}^{\mathcal{S}} \cdot \frac{k^{\max}(a)}{N_t^{\mathcal{G}} \cdot in_a}.$$

PREUVE : Pour qu'un ordonnancement périodique $\mathcal{S}$ soit valide, il doit respecter les relations de précédence entre chacune des exécutions de ses tâches.

Ainsi, pour tout arc $a = (t, t') \in \mathcal{A}$, s'il existe une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$, l'ordonnancement doit vérifier

$$\mathcal{S}\langle t, n\rangle + d(t) \leq \mathcal{S}\langle t', n'\rangle.$$

D'après la définition d'un ordonnancement périodique, $\mathcal{S}\langle t, n\rangle = \mathcal{S}\langle t, 1\rangle + (n-1)\mu_t^{\mathcal{G}}$ et donc

$$\mathcal{S}\langle t', 1\rangle - \mathcal{S}\langle t, 1\rangle \geq d(t) + (n-1)\mu_t^{\mathcal{G}} - (n'-1)\mu_{t'}^{\mathcal{G}}.$$

D'après le lemme 11 page 125, il existe une valeur k telle que

$$k \cdot \gcd_a = n \cdot in_a - n' \cdot out_a$$

et $k_{min}(a) \leq k \cdot \gcd_a \leq k_{max}(a)$. Alors,

$$n' = \frac{n \cdot in_a - k \cdot \gcd_a}{out_a}$$

Rappelons maintenant que lorsqu'un buffer est de taille bornée, la définition 8 page 72 est vérifiée, alors

$$\mu_t^{\mathcal{S}} = \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_t^{\mathcal{G}}}.$$

Par substitution, d'après la définition d'une période normalisée, on obtient la contrainte suivante,

$$\mathcal{S}\langle t', 1\rangle - \mathcal{S}\langle t, 1\rangle \geq d(t) + \frac{\Omega_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} \cdot \frac{k \cdot \gcd_a}{in_a}$$

Maintenant, comme il existe nécessairement un couple d'exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ pour lequel la valeur $k \cdot \gcd_a = k_{max}(a)$ on retrouve la contrainte du théorème.

$$\mathcal{S}\langle t', 1\rangle - \mathcal{S}\langle t, 1\rangle \geq d(t) + \frac{\Omega_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} \cdot \frac{k_{max}(a)}{in_a}$$

Inversement, si cette contrainte est vérifiée, par l'argument inverse, comme toutes les relations de précédence sont vérifiées par cette inéquation, nous prouvons que l'ordonnancement $\mathcal{S}$ est valide. ∎

Avec le théorème 15 page précédente, nous généralisons les travaux d'ordonnançabilité de Benabid et al. [10] aux *Computation Graphs*.

Une autre façon d'envisager les seuils d'exécution d'un CG serait de fournir une transformation d'un CG vers un SDFG. Nous proposons alors dans la section suivante cette solution.

Transformation d'un CG vers un SDFG équivalent

Au travers des travaux précédents, nous nous sommes aperçus que le seuil d'exécution d'un *Computation Graph* se positionne toujours comme une pénalité supplémentaire au marquage initial d'un graphe. Pour cette raison, nous sommes en mesure de proposer une

transformation d'un CG vers un SDFG tout en conservant les propriétés comportementales du graphe.

Théorème 16 *Soit le* Computation Graph $\mathcal{G} = (\mathcal{T}, \mathcal{A})$. *Il existe un SDFG* $\mathcal{G}^E = (\mathcal{T}^E, \mathcal{A}^E)$ *équivalent, pour lequel les mêmes relations de précédence sont respectées.*

- *La structure du graphe est identique,* $\mathcal{T}^E = \mathcal{T}$ *et pour chaque arc* $a \in \mathcal{A}$, *on considère un arc équivalent* $a^E \in \mathcal{A}^E$.
- *Les taux sont identiques,* $\forall a \in \mathcal{A}$, $in_{a^E} = in_a$ *et* $out_{a^E} = out_a$.
- *Le marquage des buffers est désormais* $\forall a \in \mathcal{A}$, $M_0(a^E) = M_0(a) - thr_a + out_a$.

On note que le marquage du SDFG produit peut alors être négatif (par définition inaccessible).

PREUVE : Considérons tout d'abord la condition nécessaire et suffisante d'existence d'une relation de précédence entre deux exécutions $\langle t, n\rangle$ et $\langle t', n'\rangle$ pour un arc $a = (t, t')$ du CG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$,

$$\max(0, in_a - out_a) \leq M_0(a) + n \cdot in_a - n' \cdot out_a - thr_a + out_a < in_a.$$

On considère maintenant la condition nécessaire et suffisante d'existence d'une relation de précédence induite par le buffer a^E équivalent dans le SDFG $\mathcal{G}^E = (\mathcal{T}^E, \mathcal{A}^E)$,

$$\max(0, in_{a^E} - out_{a^E}) \leq M_0(a^E) + n \cdot in_{a^E} - n' \cdot out_{a^E} + out_{a^E} < in_{a^E}.$$

La substitution de $M_0(a^E)$ par $M_0(a) - thr_a + out_a$ montre l'équivalence des deux conditions, et prouve le théorème. ∎

Si l'on prend l'exemple du CG de la Figure 7.1 page 122, le SDFG équivalent est un buffer avec un marquage négatif (voir la Figure 7.2). Compte tenu de ce marquage, deux exécutions de la tâche t sont necessaire pour exécuter une premiere fois la tâche t et d'après la régle d'execution des tâches d'un SDFG, il est toujours impossible de retrouver son état de marquage initial (un marquage négatif).

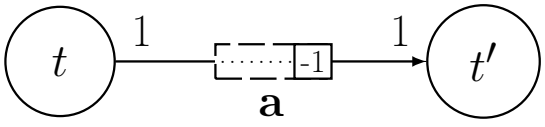


FIGURE 7.2 – Résultat de la transformation du CG de la Figure 7.1 page 122 en un SDFG équivalent, on note $in_a = out_a = 1$ et $M_0(a) = -1$.

Nous ne pensons pas qu'une telle transformation ait pour vocation d'être utilisée. Elle permet cependant d'entrevoir la relation qui existe entre marquage initial et seuil

d'exécution. Plus encore, après avoir pris connaissance de cette transformation, nous comprenons mieux l'existence des états de marquage inaccessibles ; ils sont modélisés par le marquage négatif.

Cette étude porte sur les CG, des SDFG avec un seuil d'exécution pour chaque tâche. Nous allons maintenant étudier les *Thresholded CSDFG*, un modèle CSDFG avec un seuil d'exécution pour chaque phase de chacune de ses tâches.

7.1.2 *Thresholded CSDFG*

De la même manière que pour les *Computation Graph*, nous allons caractériser l'ordonnancement périodique d'un CSDFG avec seuil d'exécution. Nous commençons par définir les *Thresholded CSDFG* (TCSDFG) et la notation que nous lui associons. Nous reprendrons ensuite les travaux du chapitre 4 pour les étendre aux TCSDFG.

Définition

Les *Thresholded CSDFG* (TCSDFG) sont introduits pour envisager les seuils d'exécutions dans un modèle CSDFG. Nous utilisons les mêmes notations que pour les CSDFG. Dans ce modèle, l'exécution des tâches n'est cependant plus conditionnée par la quantité de jetons consommés par une phase, mais par le seuil d'exécution de cette phase. Soit le buffer $a = (t, t')$, on note $thr_a(k')$ avec $k' \in \{1, \cdots, \varphi(t')\}$ la quantité de données nécessaire à l'exécution de la k'^{e} exécution de la tâche t'. Pour simplifier les notations qui suivent, nous considérerons également $thr_a(0) = thr_a(\varphi(t'))$.

Caractérisation d'une relation de précédence

Lemme 12 *Dans un TCSDFG, un buffer $a = (t, t')$ induit une relation de précédence stricte entre deux exécutions $\langle t_k, n\rangle$ et $\langle {t'}_{k'}, n'\rangle$ si et seulement si :*

$$up_a(k, k') > M_0(a) + I_a\langle t_k, n\rangle - O_a\langle {t'}_{k'}, n'\rangle \geq low_a(k, k')$$

avec

$$\begin{aligned} up_a(k, k') &= in_a(k) + thr_a(k') \\ low_a(k, k') &= \max\{thr_a(k'), thr_a(k'-1) + in_a(k) - out_a(k')\} \end{aligned}$$

Preuve : Considérons les trois conditions pour qu'il existe une relation de précédence stricte entre deux exécutions,

1. $\langle t'_{k'}, n' \rangle$ peut s'exécuter après $\langle t_k, n \rangle$:

$$M_0(a) + I_a\langle t_k, n\rangle \geq O_a\langle t'_{k'}, n'\rangle + thr_a(k')$$

2. $Pr\langle t'_{k'}, n' \rangle$ peut s'exécuter avant $\langle t_k, n \rangle$:

$$\begin{aligned} M_0(a) + I_a Pr\langle t_k, n\rangle &\geq O_a Pr\langle t'_{k'}, n'\rangle + thr_a(k'-1) & (7.1)\\ M_0(a) + I_a\langle t_k, n\rangle - O_a\langle t'_{k'}, n'\rangle &\geq in_a(-)out_a(+)thr_a(k'-1) & (7.2)\end{aligned}$$

3. $\langle t'_{k'}, n' \rangle$ ne peut pas s'exécuter avant $\langle t_k, n \rangle$:

$$\begin{aligned} M_0(a) + I_a Pr\langle t_k, n\rangle &< O_a\langle t'_{k'}, n'\rangle + thr_a(k') & (7.3)\\ M_0(a) + I_a\langle t_k, n\rangle - O_a\langle t'_{k'}, n'\rangle &< in_a(+)thr_a(k') & (7.4)\end{aligned}$$

La combinaison de ces inéquations prouve le lemme. ∎

Ordonnancement 1-Périodique

Afin de caractériser un ordonnancement périodique valide pour un TCSDFG, nous commencons par définir le lemme 13, une condition nécessaire et suffisante d'existence d'une relation de précédence entre deux phases d'un TCSDFG.

Lemme 13 *Soit un arc $a = (t, t') \in \mathcal{A}$ et un couple $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$. Il existe une infinité de couples $(n, n') \in (\mathbb{N} - \{0\})^2$ tels que l'arc a induit une relation de précédence entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ si et seulement si*

$$\alpha_a^{\min}(k, k') \leq \alpha_a(n, n') \leq \alpha_a^{\max}(k, k'),$$

avec

$$\begin{aligned} \alpha_a(n, n') &= (n-1) \times i_a - (n'-1) \times o_a\\ \alpha_a^{\min}(k, k') &= \lceil O_a\langle t'_{k'}, 1\rangle - I_a\langle t_k, 1\rangle - M_0(a) + low_a(k, k')\rceil^{gcd_a}\\ et\ \ \alpha_a^{\max}(k, k') &= \lfloor thr_a(k') + O_a\langle t'_{k'}, 1\rangle - I_a Pr\langle t_k, 1\rangle - M_0(a) - 1\rfloor^{gcd_a}. \end{aligned}$$

Preuve :

Losqu'il existe une relation de précédence entre deux exécutions, $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$, alors l'inéquation suivante est vérifiée :

$$in_a(k) + thr_a(k') > M_0(a) + I_a\langle t_k, n\rangle - O_a\langle t'_{k'}, n'\rangle \geq \mathrm{low}_a(k, k').$$

Prouvons alors qu'il existe $\alpha_a(n,n') = (n-1)i_a - (n'-1)o_a$ tel que

$$\alpha_a^{\min}(k,k') \leq \alpha_a(n,n') \leq \alpha_a^{\max}(k,k'). \tag{7.5}$$

Après réécriture des valeurs $I_a\langle t_k, n\rangle = I_a\langle t_k, 1\rangle + (n-1)\times i_a$ et $O_a\langle t'_{k'}, n'\rangle = O_a\langle t'_{k'}, 1\rangle + (n'-1)\times o_a$,

$$in_a(k) + thr_a(k') > M_0(a) + \alpha_a(n,n') + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle \geq \mathrm{low}_a(k,k').$$

La partie gauche de l'inéquation nous permet alors de vérifier

$$thr_a(k') + O_a\langle t'_{k'}, 1\rangle - I_aPr\langle t_k, 1\rangle - M_0(a) > \alpha_a(n,n').$$

Comme tous les termes sont multiples de gcd_a, on obtient

$$\lfloor thr_a(k') + O_a\langle t'_{k'}, 1\rangle - I_aPr\langle t_k, 1\rangle - M_0(a) - 1\rfloor^{\mathrm{gcd}_a} \geq \alpha_a(n,n').$$

Et la partie droite implique

$$\alpha_a(n,n') \geq \lceil O_a\langle t'_{k'}, 1\rangle - I_a\langle t_k, 1\rangle - M_0(a) + \mathrm{low}_a(k,k')\rceil^{\mathrm{gcd}_a}.$$

Alors la contrainte (7.5) est vérifiée.

Réciproquement, prouvons l'existence d'une relation de précédence $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$ lorsque la contrainte (7.5) est vérifiée.

D'après Bezout, il existe $(x,y) \in (\mathbb{Z} - \{0\})^2$ tels que $x \times i_a - y \times o_a = \mathrm{gcd}_a$.

Pour tout entier z, les valeurs

$$\begin{aligned} n(z) &= 1 + x \cdot \frac{\alpha_a^{\max}(k,k')}{\mathrm{gcd}_a} + z \cdot o_a \\ \text{et} \quad n'(z) &= 1 + y \cdot \frac{\alpha_a^{\max}(k,k')}{\mathrm{gcd}_a} + z \cdot i_a \end{aligned}$$

vérifient alors

$$\begin{aligned} (n(z)-1)i_a - (n'(z)-1)o_a &= (x \cdot \frac{\alpha_a^{\max}(k,k')}{\mathrm{gcd}_a} + z \cdot o_a)i_a - (y \cdot \frac{\alpha_a^{\max}(k,k')}{\mathrm{gcd}_a} + z \cdot i_a)o_a \\ (n(z)-1)i_a - (n'(z)-1)o_a &= \frac{x \cdot i_a - y \cdot o_a}{\mathrm{gcd}_a}\alpha_a^{\max}(k,k') \\ &= \alpha_a^{\max}(k,k'). \end{aligned}$$

Soit z suffisamment grand tel que $n(z)$ et $n'(z)$ soient positifs.

Soit $A(z) = M_0(a) + I_a\langle t_k, n(z)\rangle - O_a\langle t'_{k'}, n'(z)\rangle$. Prouvons maintenant que

$$\text{low}_a(k,k') \leq A(z) < in_a(k) + thr_a(k').$$

D'après les définitions de I_a et O_a,

$$I_a\langle t_k, n(z)\rangle = I_a\langle t_k, 1\rangle + (n(z)-1)i_a$$
$$O_a\langle t'_{k'}, n'(z)\rangle = O_a\langle t'_{k'}, 1\rangle + (n'(z)-1)o_a,$$

et $A(z)$ peut-être réécrit

$$\begin{aligned} A(z) &= M_0(a) + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle + (n(z)-1)i_a - (n'(z)-1)o_a \\ &= M_0(a) + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle + \alpha_a^{\max}(k,k'). \end{aligned}$$

– D'après la définition de $\alpha_a^{\max}(k,k')$,

$$\alpha_a^{\max}(k,k') < -M_0(a) - I_aPr\langle t_k, 1\rangle + O_a\langle t'_{k'}, 1\rangle + thr_a(k').$$

Alors,

$$A(z) < M_0(a) + I_a\langle t_k, 1\rangle - O_a\langle t'_{k'}, 1\rangle - M_0(a) - I_aPr\langle t_k, 1\rangle + O_a\langle t'_{k'}, 1\rangle + thr_a(k').$$

Comme $I_aPr\langle t_k, 1\rangle + in_a(k) = I_a\langle t_k, 1\rangle$, l'inégalité $A(z) < in_a(k) + thr_a(k')$ est vérifiée.

– Maintenant, d'après l'hypothèse, $\alpha_a^{\min}(k,k') \leq \alpha_a^{\max}(k,k')$, alors

$$\alpha_a^{\max}(k,k') \geq \text{low}_a(k,k') - M_0(a) - I_a\langle t_k, 1\rangle + O_a\langle t'_{k'}, 1\rangle$$

et donc $A(z) \geq \text{low}_a(k,k')$.

Nous prouvons que $\text{low}_a(k,k') \leq A(z) < in_a(k) + thr_a(k')$. L'inégalité du lemme 12 page 129 est vérifiée : il existe une relation de précédence entre $\langle t_k, n(z)\rangle$ et $\langle t'_{k'}, n'(z)\rangle$, ce qui complète cette preuve. ∎

Partant du lemme 13 page 130, nous sommes en mesure de proposer le théorème 17. Il s'agit d'une condition nécessaire et suffisante pour qu'un ordonnancement soit l'ordonnancement périodique valide d'un TCSDFG.

Théorème 17 *Soit $\mathcal{G}$ un CSDFG consistant dont les mémoires sont bornées. Pour tout ordonnancement périodique $\mathcal{S}$ de période normalisée $\Omega_{\mathcal{G}}^{\mathcal{S}}$, les relations de précédence induites par un arc $a = (t, t')$ sont vérifiées si et seulement si, pour tout couple*

$(k,k') \in \{1,\cdots,\varphi(t)\} \times \{1,\cdots,\varphi(t')\}$ *avec* $\alpha_a^{\min}(k,k') \leq \alpha_a^{\max}(k,k')$,

$$\mathcal{S}\langle t'_{k'},1\rangle - \mathcal{S}\langle t_k,1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \frac{\alpha_a^{\max}(k,k')}{N_t^{\mathcal{G}} \times i_a}.$$

PREUVE : Supposons que l'arc a induise une relation de précédence entre deux exécutions $\langle t_k,n\rangle$ et $\langle t'_{k'},n'\rangle$, alors l'ordonnancement $\mathcal{S}$ doit vérifier

$$\mathcal{S}\langle t_k,n\rangle + d(t_k) \leq \mathcal{S}\langle t'_{k'},n'\rangle.$$

Comme $\mathcal{S}$ est périodique, il vérifie

$$\mathcal{S}\langle t_k,1\rangle + (n-1)\mu_t^{\mathcal{S}} + d(t_k) \leq \mathcal{S}\langle t'_{k'},1\rangle + (n'-1)\mu_{t'}^{\mathcal{S}}.$$

Avec $\alpha = (n-1)i_a - (n'-1)o_a$, on obtient $(n-1) = \dfrac{(n'-1)o_a + \alpha}{i_a}$ et alors

$$\mathcal{S}\langle t'_{k'},1\rangle - \mathcal{S}\langle t_k,1\rangle \geq d(t_k) - (n'-1)\mu_{t'}^{\mathcal{S}} + \frac{(n'-1)o_a+\alpha}{i_a}\mu_t^{\mathcal{S}},$$

Ce qui revient à

$$\mathcal{S}\langle t'_{k'},1\rangle - \mathcal{S}\langle t_k,1\rangle \geq d(t_k) + (n'-1)(\frac{o_a\mu_t^{\mathcal{S}}}{i_a} - \mu_{t'}^{\mathcal{S}}) + \frac{\alpha}{i_a}\mu_t^{\mathcal{S}}.$$

Maintenant, d'après le théorème 5 page 58, dans un CSDFG à mémoire bornée la relation entre les périodes de fonctionnement des tâches t et t' s'exprime par $\dfrac{N_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} = \dfrac{\mu_t^{\mathcal{S}}}{\mu_{t'}^{\mathcal{S}}}$. D'après la définition du vecteur de répétition (voir la section 3.2.2 page 39), $\dfrac{N_{t'}^{\mathcal{G}}}{N_t^{\mathcal{G}}} = \dfrac{i_a}{o_a}$ et donc $\dfrac{o_a\mu_t^{\mathcal{S}}}{i_a} = \mu_{t'}^{\mathcal{S}}$.

$$\mathcal{S}\langle t'_{k'},1\rangle - \mathcal{S}\langle t_k,1\rangle \geq d(t_k) + \mu_{\mathcal{G}}^{\mathcal{S}}\frac{\alpha}{i_a}.$$

De plus, d'après la définition 5 page 58, $\mu_t^{\mathcal{S}} = \dfrac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{N_t^{\mathcal{G}}}$, l'inégalité devient

$$\mathcal{S}\langle t'_{k'},1\rangle - \mathcal{S}\langle t_k,1\rangle \geq d(t_k) + \frac{\Omega_{\mathcal{G}}^{\mathcal{S}}}{i_a N_t^{\mathcal{G}}}\alpha.$$

Enfin, le terme de droite augmente selon la valeur de α et d'après le lemme 13 page 130 la valeur $\alpha = \alpha_a^{\max}(k,k')$ est atteinte. La relation de précédence est assurée si

$$\mathcal{S}\langle t'_{k'},1\rangle - \mathcal{S}\langle t_k,1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \frac{\alpha_a^{\max}(k,k')}{N_t^{\mathcal{G}} \cdot i_a}. \tag{7.6}$$

Réciproquement, si l'on fait l'hypothèse que l'ordonnancement périodique $\mathcal{S}$ vérifie les inégalités exprimées par ce théoreme. Soit $a = (t, t') \in \mathcal{A}$ et le couple d'entiers (n, n') et $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ tels qu'il existe une relation de précédence entre $\langle t_k, n\rangle$ et $\langle t'_{k'}, n'\rangle$. Vérifions que cette relation de précédence est bien vérifiée par l'équation

$$\mathcal{S}\langle t'_{k'}, 1\rangle - \mathcal{S}\langle t_k, 1\rangle \geq d(t_k) + \Omega_{\mathcal{G}}^{\mathcal{S}} \times \frac{\alpha_a^{\max}(k, k')}{N_t^{\mathcal{G}} \cdot i_a}. \tag{7.7}$$

D'après le lemme 13 page 130, $\alpha = (n-1) \times i_a - (n'-1) \times o_a$ est bornée par $\alpha_a^{\max}(k, k')$. Donc lorsque la contrainte (7.7) est vérifiée, la contrainte (7.6) page précédente l'est aussi. D'après l'argument inverse, on prouve alors que $\mathcal{S}\langle t_k, n\rangle + d(t_k) \leq \mathcal{S}\langle t'_{k'}, n'\rangle$. ∎

Nous pouvons maintenant dire que les techniques d'analyse du chapitre 6 s'appliquent aussi aux TCSDFG, et donc au modèle ΣC dans sa définition actuelle.

Maintenant, afin d'anticiper les futures évolutions du langage ΣC, intéressons-nous aux CSDFG avec phases d'initialisation.

7.2 Les phases d'initialisation

Les phases d'initialisation sont des phases particulières, définies pour certaines tâches, et n'ayant lieu qu'une seule fois au début de l'exécution d'un programme. Le modèle *Initialized CSDFG* (ICSDFG) est un modèle CSDFG avec des phases d'initialisation.

Peu de modèles existants considèrent cette fonctionnalité. Notons cependant le modèle n-synchrone, principalement utilisé avec le langage lucy-n [61]. Il s'apparente fortement à un modèle ICSDFG, mais pour lequel les taux de production et de consommation sont binaires uniquement. Il existe alors, pour ce modèle, des techniques d'analyse appliquées à des problèmes similaires aux notres, comme le dimensionnement mémoire [60]. Une extension de ce langage est aussi en cours de développement et correspond plus précisément aux ICSDFG [47].

Nous présentons maintenant le modèle ICSDFG.

7.2.1 *Initialized CSDFG*

Un *Initialized Cyclo-Static Dataflow Graph* (ICSDFG) $\mathcal{G} = (\mathcal{T}, \mathcal{A})$ respecte toutes les propriétés d'un CSDFG. Chaque tâche $t \in \mathcal{T}$ dispose cependant d'une caractéristique supplémentaire, un ensemble de $\phi(t)$ phases d'initialisation. Ces phases ne s'exécutent qu'une unique fois lors de la première itération de la tâche t et leurs exécutions seront notées $\langle t_k, 1\rangle$ avec k allant de $1 - \phi(t)$ à 0.

On note $Pr\langle t_k, n\rangle$ l'exécution de la tâche t qui précède l'exécution $\langle t_k, n\rangle$. Plus formellement,

$$Pr\langle t_k, n\rangle = \begin{cases} \langle t_{k-1}, n\rangle & si\ k > 1 \\ \langle t_{k-1}, n\rangle & si\ k > 1 - \phi(t)\ et\ si\ n = 1 \\ \langle t_{\varphi(t)}, n-1\rangle & sinon \end{cases}$$

L'exécution $\langle t_{\varphi(t)}, 0\rangle$ est fictive, et n'est introduite que pour simplifier la définition de Pr.

Ainsi, le CSDFG est un cas particulier des ICSDFG où $\forall t \in \mathcal{T}, \phi(t) = 0$.

Caractérisation de l'ordonnancement périodique d'un ICSDFG

La définition de l'ordonnancement périodique d'un ICSDFG est sensiblement identique à celle d'un CSDFG. Pour toutes les tâches d'un ICSDFG, on fixe une période et une date de départ par phase (phases d'initialisation incluses). La définition de la période est inchangée, elle ne porte que sur les phases cycliques.

Comme les phases d'initialisation sont absorbées par notre notation, comparée aux CSDFG, la caractérisation d'une relation de précédence est inchangée :

$$in_a(k) > M_0(a) + I_a\langle t_k, n\rangle - O_a\langle t'_{k'}, n'\rangle \geq \max\{0, in_a(k) - out_a(k')\}.$$

Pour un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$, et pour un arc $a = (t, t') \in \mathcal{A}$, les contraintes périodiques vérifient qu'un ordonnancement périodique respecte les relations de précédence induites par ce buffer. On compte alors autant de contraintes que de couples $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ tels que $\alpha_a^{\min}(k, k') \leq \alpha_a^{\max}(k, k')$.

Pour un ICSDFG, des contraintes périodes portent alors sur deux catégories de couples de phases :

- Des couples de phases du même type :
  - Il peut s'agir de deux phases cycliques $(k, k') \in \{1, \cdots, \varphi(t)\} \times \{1, \cdots, \varphi(t')\}$ telles que $\alpha_a^{\min}(k, k') \leq \alpha_a^{\max}(k, k')$. La contrainte est alors identique aux contraintes périodiques d'un CSDFG.
  - Il peut aussi s'agir de deux exécutions de phases d'initialisation. La contrainte utilisée sera donc classiquement

$$\mathcal{S}\langle t_k, n\rangle + d(t_k) \leq \mathcal{S}\langle t'_{k'}, n'\rangle.$$

- Sinon, il s'agit d'un couple mixte, une contrainte entre une phase d'initialisation et une phase cyclique. Dans ce cas, une phase d'initialisation ne peut intervenir dans une relation de précédence qu'avec une unique phase cyclique. Déterminer la phase et l'itération de l'exécution en dépendance avec une phase d'initialisation est réalisable

en temps constant, nous définirons également cette contrainte sous la forme

$$\mathcal{S}\langle t_k, n\rangle + d(t_k) \leq \mathcal{S}\langle t'_{k'}, n'\rangle.$$

Nous sommes donc en mesure de définir une caractérisation d'un ordonnancement périodique pour les ICSDFG, et d'appliquer nos méthodes d'analyse à ce type de modèle.

Conclusion

L'intégration de nos travaux d'analyse à la chaîne de compilation AccessCore n'était possible que par l'extension de nos méthodes à des modèles dataflows moins conventionnels. En effet, tandis que les CSDFG disposent d'une large littérature, on ne compte que très peu d'études sur les CG, ou sur des modèles similaires aux TCSDFG et ICSDFG présentés ici.

Dans ce chapitre nous avons présenté quelques propriétés originales des CG, puis nous avons étendu certains de ces raisonnements à un modèle dataflow défini à cette occasion, les *Thresholded CSDFG* (TCSDFG). Dans une seconde partie, nous avons ensuite étudié une extension des modèles CSDFG avec des phases d'initialisation, le modèle *Initialised CSDFG* (ICSDFG).

Ces résultats nous ont permis d'intégrer les différents algorithmes proposés dans le chapitre 6 à la chaîne de compilation AccessCore.

Chapitre 8

Conclusion

Résumé

Les travaux présentés dans cette thèse ont pour principal intérêt d'améliorer une chaîne de compilation industrielle : la chaîne de compilation AccessCore. Cette chaîne de compilation est développée par la société Kalray, elle permet de compiler des applications ΣC vers le processeur MPPA, une architecture multicœur comptant plus de 256 cœurs de calcul. La particularité du langage ΣC est qu'il est fidèle à une modélisation proche du *Cyclo-Static Dataflow Graph*, un modèle dataflow dont les caractéristiques permettent d'obtenir les garanties de fonctionnement nécessaires aux applications embarquées.

Dans ce contexte d'étude, nous nous intéressons à trois problèmes d'analyse :

- vérifier la vivacité d'une application,
- évaluer sa fréquence de fonctionnement maximale
- et calculer un espace mémoire nécessaire à ce fonctionnement.

Nous faisons le choix d'étudier ces problèmes sous un angle encore peu considéré, l'ordonnancement périodique.

Les premières contributions de cette thèse visent tout d'abord à caractériser l'ordonnancement périodique d'un CSDFG. Ils font suite aux travaux de Benazouz [11] sur la formulation des relations de précédence dans un CSDFG et généralisent ceux de Benabid et al. [10] sur l'ordonnancement périodique d'un SDFG. Nous étudions ensuite les ordonnancements K-périodiques des SDFG avec un vecteur de périodicité fixé. Au travers de ces ordonnancements, nous souhaitons améliorer les résultats obtenus avec un ordonnancement périodique, tout en évitant la complexité d'une méthode exacte.

Dans une seconde partie, nous abordons des sujets plus appliqués, principalement tournés vers l'analyse statique de programmes ΣC. Nous présentons un ensemble de jeux de test CSDFG, nous les utilisons pour évaluer les algorithmes de nos contributions. Nous proposons aussi un générateur d'instances aléatoires afin d'enrichir ces jeux de test. Ce générateur comble un vide, aucun générateur existant ne nous ayant permis de produire des CSDFG de grande taille. La première étape d'analyse que nous traitons concerne le problème de vivacité pour un CSDFG. Nous proposons alors un algorithme inspiré des conditions suffisantes de vivacité présentées par Benazouz [11]. Nous traitons ensuite de l'évaluation du débit maximal d'un CSDFG, et du dimensionnement mémoire d'un CSDFG afin de garantir sa vivacité, ou de garantir une fréquence de fonctionnement minimale. Tous les algorithmes que nous proposons sont des applications directes de nos résultats sur les ordonnancements périodiques des CSDFG. Elles ne fournissent alors que des résultats approchés, mais les résultats expérimentaux que nous avons réalisés nous laissent penser que ces méthodes restent particulièrement efficaces.

Dans un dernier chapitre, nous étudions l'intégration de nos techniques d'analyse à la chaîne de compilation AccessCore. Nous présentons alors deux fonctionnalités du langage

ΣC, jusqu'ici ignorées. Pour chacune de ces extensions, nous prouvons que nos méthodes s'y appliquent.

Dans l'ensemble de nos contributions, on peut identifier deux caractéristiques majeures. Premièrement, en faisant le choix d'une modélisation périodique du problème, toutes les solutions que nous proposons sont rapides, bien plus rapides que celles basées sur des ordonnancements *au plus tôt*. En contrepartie, les solutions obtenues avec ces méthodes sont approchées. Parfois même, il se peut qu'aucune solution ne puisse être produite.

Suite à ces travaux, plusieurs perspectives apparaissent, nous les divisons en trois catégories. D'une part l'amélioration de l'approche périodique par l'utilisation des techniques K-périodiques. D'autre part l'application de notre approche à d'autres problèmes d'analyse. Nous envisageons enfin la conception d'un exécutif périodique.

Perspectives

Amélioration de l'approche périodique

Dans le chapitre 5, nous proposons une alternative aux ordonnancements périodiques, les ordonnancements K-périodiques avec un vecteur de périodicité fixé. Cependant, ces travaux restent inachevés en deux points. D'une part, bien que nous soyons fortement persuadés de l'existence de vecteur de périodicité strictement dominant, nous ne disposons pas encore des preuves suffisantes pour nous en assurer. D'autre part, si ces solutions dominantes existent, nous souhaitons connaître une méthode efficace pour les découvrir.

Une fois ces travaux réalisés, nous serions alors en mesure de proposer des algorithmes d'analyse plus efficaces, et pour lesquels des garanties de performance sont envisageables.

Application à d'autres domaines

Dans cette thèse, seulement trois étapes de compilation dataflow sont étudiées. Il reste donc des étapes d'analyse non explorées et pour lesquelles notre méthodologie pourrait fournir des solutions pertinentes.

Partitionnement d'un CSDFG sur une architecture multiprocesseur

La plupart des outils de partitionnement actuels s'appuient sur une exploration des solutions à l'aide de différentes métaheuristiques. Nous sommes persuadés que notre algorithme d'évaluation du débit périodique d'un CSDFG pourrait alors être utilisé, dans ce contexte, comme fonction de décision.

Vérification de garanties supplémentaires comme la latence

La description de nos ordonnancements sous forme de contraintes linéaires nous permet aisément d'intégrer des contraintes supplémentaires, tout particulièrement des contraintes de latence. Cependant, la description périodique d'un ordonnancement peut induire un impact négatif sur lesdites latences. Il serait judicieux d'étudier cet impact, et de proposer des solutions adaptées. Nous pensons par exemple à l'utilisation du *retiming*. Le *retiming* consiste à produire un ordonnancement préliminaire d'une application avant de respecter une définition périodique.

Considération d'un *runtime* périodique « asynchrone »

Jusqu'ici nous n'utilisons l'ordonnancement périodique que comme un outil d'analyse. Nous n'avons alors aucune considération sur l'exécutif des applications traitées. Il s'agit pourtant d'un champ de recherche à part entière et pour lequel il existe encore de nombreux problèmes de performance.

Bien souvent l'exécutif est centralisé. Il existe des dépendances de données entre les tâches d'un programme et une autorité supérieure décide de leur ordonnancement. Ce principe entraine des surcoûts d'exécution, principalement dus à une congestion du système. Des solutions décentralisées existent, mais la taille du code généré est souvent plus élevée. Cet excédent est lié à l'utilisation d'ordonnancements *au plus tôt*. Lorsqu'un système décentralisé ne subit pas ce surcoût, des solutions de synchronisation doivent tout de même être utilisées par garantir la présence de donnée dans chaque buffer.

Il serait intéressant d'envisager un exécutif ou l'exécution d'une application est définie par un ordonnancement périodique. Il existe déjà des solutions de ce type dans un contexte temps réel dur [8]. Les durées d'exécutions des tâches doivent alors être correctement définies sous peine d'un arrêt brutal du système ou d'indéterminisme. Nous souhaitons aller plus loin, et mettre au point une technique similaire dans un contexte plus général, ou des retards d'exécution seraient possibles.

Bibliographie

[1] OpenMP Application Program Interface Version 3.0. Technical report, 2008.

[2] M. Ade, Rudy Lauwereins, et Jean A. Peperstraete. Buffer memory requirements in DSP applications. In *Proceedings of IEEE 5th International Workshop on Rapid System Prototyping*, pages 108–123. IEEE Comput. Soc. Press, 1994.

[3] Sukumar R. Anapalli, Krishna C. Chakilam, et Timothy W. O'Neil. Static Scheduling for Cyclo Static Data Flow Graphs. In *Parallel and Distributed Processing Techniques and Applications, PDPTA 2009*, pages 302–306. CSREA Press, 2009.

[4] Arvind et Kim P. Gostelow. The U-interpreter. *Computer*, 15(2) :42 – 49, 1982.

[5] Arvind et Vinod Kathail. A multiple processor data flow machine that supports generalized procedures. In *Proceedings of the 8th annual symposium on Computer Architecture (ISCA '81)*, pages 291–302, 1981.

[6] Mohamed A. Bamakhrama et Todor Stefanov. Hard-real-time scheduling of data-dependent tasks in embedded streaming applications. In *Proceedings of the 9th ACM international conference on Embedded software - EMSOFT '11*, page 195, New York, New York, USA, 2011. ACM Press.

[7] Mohamed A. Bamakhrama, Jiali T. Zhai, Hristo Nikolov, et Todor Stefanov. Daedalus RT : The System-Level Design Flow for Hard-Real-Time Embedded MPSoCs Platforms. *rd-access.eu*, 27(3), 2011.

[8] Mohamed A. Bamakhrama, Jiali T. Zhai, Hristo Nikolov, et Todor Stefanov. A methodology for automated design of hard-real-time embedded streaming systems. In *Design, Automation & Test in Europe (DATE)*, pages 941–946. Ieee, March 2012.

[9] Marco J.G. Bekooij, Rob Hoes, Orlando Moreira, Peter M. Poplavko, Milan Pastrnak, Bart Mesman, Jan David Mol, Sander Stuijk, Valentin Gheorghita, et Jef van Meerbergen. Dataflow analysis for real-time embedded multiprocessor system design. *Dynamic and Robust*, pages 81–108, 2005.

[10] Abir Benabid, Claire Hanen, Olivier Marchetti, et Alix Munier-Kordon. Periodic Schedules for Bounded Timed Weighted Event Graphs. *IEEE Transactions on Automatic Control*, 57(5) :1222 – 1232, 2012.

[11] Mohamed Benazouz. *Buffer Sizing for Stream Processing Applications*. PhD thesis, Université P. et M. Curie, Paris, France, 2012.

[12] Mohamed Benazouz et Alix Munier-Kordon. Cyclo-static DataFlow phases scheduling optimization for buffer sizes minimization. In *Proceedings of the 16th International Workshop on Software and Compilers for Embedded Systems - M-SCOPES '13*, page 3, New York, New York, USA, 2013. ACM Press.

[13] Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, T Michel, et Pascal Urard. A new method for minimizing buffer sizes for Cyclo-Static Dataflow graphs. In *Embedded Systems for Real-Time Multimedia (ESTIMedia '10)*, pages 11–20, 2010.

[14] Mohamed Benazouz, Olivier Marchetti, Alix Munier-Kordon, et Pascal Urard. A new approach for minimizing buffer capacities with throughput constraint for embedded system design. In *International Conference on Computer Systems and Applications (AICCSA '10)*, 2010.

[15] Mohamed Benazouz, Alix Munier-Kordon, Thomas Hujsa, et Bruno Bodin. Liveness Evaluation of a Cyclo-Static DataFlow Graph. In *Design Automation Conference (DAC'13)*, pages 3–7, Austin, TX, USA, 2013. ACM Press.

[16] Bishnupriya Bhattacharya et Shuvra S. Bhattacharyya. Parameterized Dataflow Modeling for DSP Systems. *IEEE Transactions on Signal Processing*, 49(10) :2408–2421, 2001.

[17] Shuvra S. Bhattacharyya, Elaine Cheong, John Davis II, Mudit Goel, Bart Kienhuis, Christopher Hylands, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, John Reekie, Steve Neuendorffer, Neil Smyth, Jeff Tsay, Brian Vogel, Winthrop Williams, Yuhong Xiong, et Haiyang Zheng. Ptolemy II - Heterogeneous Concurrent Modeling and Design in Java. Technical report, 2002.

[18] Shuvra S. Bhattacharyya, Gordon Brebner, Jörn W. Janneck, Johan Eker, Carl Von Platen, Marco Mattavelli, et Mickaël Raulet. OpenDF : a dataflow toolset for reconfigurable hardware and multicore systems. *ACM SIGARCH Computer Architecture News*, 36(5) :29–35, 2009.

[19] Greet Bilsen, Marc Engels, Rudy Lauwereins, et Jean A. Peperstraete. Cyclo-static data flow. *IEEE Transactions on Signal Processing*, pages 3255–3258, 1995.

[20] Bruno Bodin, Alix Munier-Kordon, et Benoît Dupont de Dinechin. K-Periodic Schedules for Evaluating the Maximum Throughput of a Synchronous Dataflow Graph. In *International Conference on Embedded Computer Systems : Architectures, Modeling, and Simulation, SAMOS XII*, pages 152–159, 2012.

[21] Bruno Bodin, Alix Munier-kordon, et Benoît Dupont de Dinechin. Periodic Schedules for Cyclo-Static Dataflow. In *Embedded Systems for Real-Time Multimedia (ESTIMedia '13)*, pages 105–114, 2013.

[22] Julien Boucaron, Anthony Coadou, Benoit Ferrero, J.V. V Millo, et R. De Simone. Kahn-extended event graphs. *Sophia*, (May), 2008.

[23] Joseph T. Buck et Edward A. Lee. Scheduling dynamic dataflow graphs with bounded memory using the token flow model. In *IEEE International Conference on Acoustics, Speech, and Signal Processing (ICASSP-93)*, number September, pages 429–432. Ieee, 1993.

[24] David R. Butenhof. *Programming With Posix Threads*. 1997.

[25] Philippe Chrétienne. Transient and limiting behavior of timed event graphs. *RAIRO Techniques et Sciences Informatiques*, 4 :127–192, 1985.

[26] Frederic Commoner, Aanatol W. Holt, S. Even, et A. Pnueli. Marked directed graphs. *Journal of Computer and System Sciences*, 5(5) :511–523, October 1971.

[27] Loïc Cudennec et Renaud Sirdey. Parallelism reduction based on pattern substitution in dataflow oriented programming languages. *Procedia Computer Science*, 9 :146–155, 2012.

[28] DE Culler. Managing parallelism and resources in scientific data-flow programs. Technical report. 1990.

[29] James Dabney et Thomas L Harman. *Mastering Simulink*. 1998.

[30] Ali Dasdan, Sandy S. Irani, et Rajesh K. Gupta. Efficient algorithms for optimum cycle mean and optimum cost to time ratio problems. *Design Automation Conference (DAC'99)*, pages 37–42, 1999.

[31] Robert de Groote, Jan Kuper, Hajo Broersma, et Gerard J.M. Smit. Max-Plus Algebraic Throughput Analysis of Synchronous Dataflow Graphs. *38th Euromicro Conference on Software Engineering and Advanced Applications*, pages 29–38, September 2012.

[32] Pablo de Oliveira Castro, Stéphane Louise, et Denis Barthou. Reducing memory requirements of stream programs by graph transformations. In *International Conference on High Performance Computing and Simulation (HPCS)*, pages 171–180. IEEE, 2010.

[33] Jack B. Dennis. First version of a data flow procedure language. *Programming Symposium*, 19 :362–376, 1974.

[34] Kristof Denolf, Marco J.G. Bekooij, Johan Cockx, Diederik Verkest, et Henk Corporaal. Exploiting the Expressiveness of Cyclo-Static Dataflow to Model Multimedia Implementations. *EURASIP Journal on Advances in Signal Processing*, 2007 :1–15, 2007.

[35] Karol Desnos, Maxime Pelcat, Jean-francois Nezan, et Slaheddine Aridhi. Memory Bounds for the Distributed Execution of a Hierarchical Synchronous Data-Flow Graph. *SAMOS '12*, pages 160–167, 2012.

[36] Petru Eles, Krzysztof Kuchcinski, Zebo Peng, Alexa Doboli, et Paul Pop. Scheduling of conditional process graphs for the synthesis of embedded systems. *Proceedings Design, Automation and Test in Europe*, pages 132–138, 1998.

[37] Michael J Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21 :948–960, 1972.

[38] DD D Gajski, DA A Padua, et DJ J Kuck. A second opinion on data flow machines and languages. *Computer*, (1), 1982.

[39] M. R. Garey et D. S. Johnson. Complexity Results for Multiprocessor Scheduling under Resource Constraints. *SIAM Journal on Computing*, 4(4) :397–411, December 1975.

[40] Marc Geilen et Twan Basten. Reactive process networks. In *Proceedings of the 4th ACM international conference on Embedded software (EMSoft '04)*, 2004.

[41] Amir H. Ghamarian et Marc Geilen. Liveness and boundedness of synchronous data flow graphs. *Formal Methods in Computer Aided Design (FMCAD'06)*, 2006.

[42] Amir H. Ghamarian, Marc Geilen, Sander Stuijk, Twan Basten, Arno Moonen, Marco J.G. Bekooij, Bart D. Theelen, et MohammadReza Mousavi. Throughput Analysis of Synchronous Data Flow Graphs. In *International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 25–36, 2006.

[43] Alain Girault, B Lee, et Edward A. Lee. Hierarchical finite state machines with multiple concurrency models. *Computer-Aided Design of*, 18(6) :742–760, 1999.

[44] Michel Gondran et Michel Minoux. *Graphs and algorithms*. John Wiley and sons, first edition, 1984.

[45] Thierry Goubier, Renaud Sirdey, Stéphane Louise, et Vincent David. ΣC : A programming model and language for embedded manycores. *Proceedings of the 11th international conference on Algorithms and architectures for parallel processing (ICA3PP'11)*, 7016 :385–394, 2011.

[46] R Govindarajan et G.R. Gao. A novel framework for multi-rate scheduling in DSP applications. In *Proceedings of International Conference on Application Specific Array Processors (ASAP '93)*, pages 77–88. IEEE Comput. Soc. Press, 1993.

[47] Adrien Guatto, Albert Cohen, Léonard Gérard, Louis Mandel, et Marc Pouzet. Integer clocks. In *Synchron Workshop*, 2012.

[48] IBplusAGB5CSDF. http ://www-soc.lip6.fr/˜bodin/, 2013.

[49] Gurobi Optimization Inc. Gurobi Optimizer Reference Manual, 2013.

[50] Gilles Kahn. The semantics of a simple language for parallel programming. *Information processing*, 1974.

[51] Richard M. Karp et Raymond E. Miller. Properties of a model for parallel computations : Determinancy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6) :1390–1411, 1966.

[52] Samer F. Khasawneh, Michael E. Ritcher, et Timothy W. O'Neil. Static Scheduling for synchronous data flow graphs. *Computers and Their Applications*, 1 :38–43, 2007.

[53] Dong-Ik Ko et Shuvra S. Bhattacharyya. Modeling of block-based DSP systems. *The Journal of VLSI Signal Processing*, 40(3) :289–299, 2005.

[54] Edward A. Lee. The Problem with Threads. *Computer*, 39(5) :33–42, May 2006.

[55] Edward A. Lee et Soonhoi Ha. Scheduling strategies for multiprocessor real-time DSP. *IEEE Global Telecommunications Conference, 1989, and Exhibition. 'Communications Technology for the 1990s and Beyond*, pages 1279–1283, 1989.

[56] Edward A. Lee et David G. Messerschmitt. Synchronous dataflow. *Proceedings of the IEEE*, 75(9) :1235–1245, 1987.

[57] Edward A. Lee, E Goei, H Heine, W H. Ho, Shuvra S. Bhattacharyya, Jeff C. Bier, et E Guntvedt. Gabriel : A Design Environment for Programmable DSPs. *26th ACMIEEE Design Automation Conference*, pages 141–146, 1989.

[58] Robin Lougee-Heimer, Matthew J. Saltzman, et Ted K. Ralphs. The COIN-OR Initiative : Open-source Software Accelerates Operations Research Progress, 2001.

[59] Andrew Makhorin. GLPK (GNU linear programming kit), 2006.

[60] L Mandel et F Plateau. Scheduling and Buffer Sizing of n-Synchronous Systems. *Mathematics of Program Construction*, 2012.

[61] Louis Mandel, Florence Plateau, et Marc Pouzet. Lucy-n : a n-synchronous extension of Lustre. *Mathematics of Program Construction*, 2010.

[62] Olivier Marchetti et Alix Munier-Kordon. Complexity results for bi-criteria cyclic scheduling problems. pages 17–24, 2006.

[63] Olivier Marchetti et Alix Munier-Kordon. A sufficient condition for the liveness of weighted event graphs. *European Journal of Operational Research*, 197(2) :532–540, September 2009.

[64] Message Passing Interface Forum. MPI : A Message-Passing Interface Standard. *International Journal of Supercomputer Applications*, 8 :623, 1994.

[65] Orlando Moreira, Twan Basten, Marc Geilen, et Sander Stuijk. Buffer Sizing for Rate-Optimal Single-Rate Data-Flow Scheduling Revisited. *IEEE Transactions on Computers*, 59(2) :188–201, February 2010.

[66] Alix Munier-Kordon. Régime asymptotique optimal d'un graphe d'événements temporisé généralisé : Application à un problème d'assemblage. *RAIRO-Automatique Productique Informatique Industrielle*, 25(5) :487–513, 1993.

[67] Praveen K. Murthy et Edward A. Lee. Multidimensional synchronous dataflow. *IEEE Transactions on Signal Processing,*, 50(7) :2064–2079, 2002.

[68] Qi Ning et Guang R. Gao. A novel framework of register allocation for software pipelining. *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '93*, pages 29–42, 1993.

[69] Hyunok Oh et Soonhoi Ha. Efficient code synthesis from extended dataflow graphs for multimedia applications. In *Design Automation Conference (DAC '02)*, pages 275–280, 2002.

[70] T.M. M Parks. *Bounded scheduling of process networks*. Ph.d thesis, University of California, 1995.

[71] Maxime Pelcat, Pierrick Menuet, Slaheddine Aridhi, et Jean-François Nezan. Scalable Compile-Time Scheduler for Multi-Core Architectures. In *Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 3–6, 2009.

[72] Sanjay Raina. Virtual Shared Memory : A Survey of Techniques and Systems. (December), 1992.

[73] C V Ramamoorthy et G S Ho. Performance Evaluation of Asynchronous Concurrent Systems Using Petri Nets. *IEEE Transactions on Software Engineering*, SE-6 :440–449, 1980.

[74] Raymond Reiter. Scheduling parallel computations. *Journal of the ACM*, 15(4) : 590–599, 1968.

[75] Sebastian Ritz, M Pankert, et Heinrich Meyr. High level software synthesis for signal processing systems. *Application Specific Array*, 1992.

[76] John P. Shen et Mikko H. Lipasti. *Modern processor design : fundamentals of superscalar processors*. Boston, 2005.

[77] Gilbert C. Sih et Edward A. Lee. A compile-time scheduling heuristic for interconnection-constrained heterogeneous processor architectures. *IEEE Transactions on Parallel and Distributed Systems*, 4(2) :175–187, 1993.

[78] Renaud Sirdey. *Contributions à l'optimisation combinatoire pour l'embarqué : des autocommutateurs cellulaires aux microprocesseurs massivement parallèles*. Hdr, Université de Technologie de Compiègne, 2011.

[79] Renaud Sirdey, Jacques Carlier, et Dritan Nace. Approximate solution of a resource-constrained scheduling problem. *Journal of Heuristics*, 15(1) :1–17, October 2009.

[80] Sundararajan Sriram et Shuvra S. Bhattacharyya. *Embedded multiprocessors : Scheduling and synchronization*. CRC, second edition, 2009.

[81] Sander Stuijk, Marc Geilen, et Twan Basten. SDFˆ3 : SDF For Free. In *Sixth International Conference on Application of Concurrency to System Design (ACSD'06)*, pages 276–278. IEEE, 2006.

[82] Sander Stuijk, Marc Geilen, et Twan Basten. Throughput-Buffering Trade-Off Exploration for Cyclo-Static and Synchronous Dataflow Graphs. *IEEE Transactions on Computers*, 57(10) :1331–1345, 2008.

[83] Michael B. Taylor, Jason Kim, Jason Miller, David Wentzlaff, Fae Ghodrat, Ben Greenwald, Henry Hoffman, Paul Johnson, Walter Lee, Albert Ma, Arvind Saraf, Mark Seneski, Nathan Shnidman, Volker Strumpen, Matt Frank, Saman Amarasinghe, et Anant Agarwal. The Raw microprocessor : a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22(2) :25–35, March 2002.

[84] Enrique Teruel, Piotr Chrzastowski-Wachtel, José-Manuel Colom, et Manuel Silva. On weighted T-systems. *Application and Theory of petri Nets*, 616 :348–367, 1992.

[85] Bart D. Theelen, Marc Geilen, Twan Basten, Jeroen P.M. Voeten, Valentin Gheorghita, et Sander Stuijk. A scenario-aware data flow model for combined long-run average and worst-case performance analysis. In *Fourth ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2006. MEMOCODE '06.*, pages 185–194. IEEE, 2006.

[86] William Thies, Michal Karczmarek, et Saman Amarasinghe. StreamIt : A language for streaming applications. *Compiler Construction*, pages 179–196, 2002.

[87] William Thies, Jasper Lin, et Saman Amarasinghe. Phased Computation Graphs in the Polyhedral Model. Technical Report August 2002, 2002.

[88] A. Turjan. Compiling nested loop programs to process networks. 2007.

[89] John Von Neumann. First Draft of a Report on the EDVAC. Technical Report 1, 1945.

[90] Piet Wauters, Marc Engels, Rudy Lauwereins, et Jean A. Peperstraete. Cyclo-dynamic dataflow. In *The 4th EUROMICRO Workshop on Parallel and Distributed Processing,*, number January, pages 319 – 326, Braga,Portugal, 1996. Published by the IEEE Computer Society.

[91] David Wentzlaff, Patrick Griffin, Henry Hoffmann, Liewei Bao, Bruce Edwards, Carl Ramey, Matthew Mattina, Chyi-Chang Miao, John F. Brown III, et Anant Agarwal. On-Chip Interconnection Architecture of the Tile Processor. *IEEE Micro*, 27(5) : 15–31, September 2007.

[92] Paul G. Whiting et R.S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16(4) :38–59, 1994.

[93] Maarten H. Wiggers, Marco J.G. Bekooij, Pierre G. Jansen, et Gerard J.M. Smit. Efficient Computation of Buffer Capacities for Cyclo-Static Real-Time Systems with Back-Pressure. *13th IEEE Real Time and Embedded Technology and Applications Symposium (RTAS'07)*, pages 281–292, April 2007.

[94] Maarten H. Wiggers, Marco J.G. Bekooij, et Gerard J.M. Smit. Efficient computation of buffer capacities for cyclo-static dataflow graphs. *Proceedings of the 44th annual conference on Design automation - DAC '07*, (1) :658, 2007.

Publications personnelles

Publications internationales avec comité de lecture

K-Periodic schedules for evaluating the maximum throughput of a Synchronous Dataflow graph
Bruno Bodin, Alix Munier-Kordon et Benoît Dupont de Dinechin
International Conference on Embedded Computer Systems (SAMOS), 15-19 Juillet 2012, Samos, Grèce

Liveness evaluation of a cyclo-static DataFlow graph
Mohamed Benazouz, Alix Munier-Kordon, Thomas Hujsa et Bruno Bodin
50$^{\text{th}}$ Annual Design Automation Conference (DAC), 1-5 Juin 2013, Austin TX, Etats-unis

Extended Cyclostatic Dataflow Program Compilation and Execution for an Integrated Manycore Processor
Pascal Aubry, Pierre-Edouard Beaucamps, Frédéric Blanc, Bruno Bodin, Sergiu Carpov, Löıc Cudennec, Vincent David, Philippe Dore, Paul Dubrulle, Benôıt Dupont de Dinechin, Franc Galea, Thierry Goubier, Michel Harrand, Samuel Jones, Jean-Denis Lesage, Stéphane Louise, Nicolas Morey Chaisemartin, Thanh Hai Nguyen, Xavier Raynaud et Renaud Sirdey
Architecture, Languages, Compilation and Hardware support for Emerging ManYcore systems workshop (ALCHEMY), 5-7 juin 2013, Barcelone, Espagne

Periodic Schedule of a Cyclo-Static Dataflow graph
Bruno Bodin, Alix Munier-Kordon et Benoît Dupont de Dinechin
The 11$^{\text{th}}$ IEEE Symposium on Embedded Systems For Real-time Multimedia (ESTIMedia), 3-4 Octobre 2013, Montréal, Canada

Communications nationales avec comité de lecture

Evaluation du débit maximum d'un Synchronous DataFlow
Bruno Bodin, Alix Munier-Kordon et Benoît Dupont de Dinechin
14$^{\text{e}}$ Conférence de la Société Française de Recherche Opérationnelle et Aide à la Décision (ROADEF), 13-15 février 2013, Troyes, France

Notations

$\mathcal{A}$	Ensemble des arcs d'un dataflow tel que $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 23
$b(a)$	Le buffer modélisé par l'arc a	page 109
$\mathcal{Y}(a)$	Ensemble des couples de phases nécessaires pour définir les contraintes périodiques d'un buffer a	page 105
$B(a)$	La taille totale d'un buffer	page 47
$\langle t_k, n \rangle$	Désigne la n^{e} exécution de la k^{e} phase de la tâche $t \in \mathcal{T}$ d'un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 49
$out_a(k')$	Quantité de données consommées par la k'^{e} phase de la tâche t' d'un CSDFG dans le buffer $a = (t, t')$	page 25
$thr_a(k')$	Quantité de données nécessaire à l'exécution de la k'^{e} phase de la tâche t' d'un TCSDFG dans le buffer $a = (t, t')$	page 25
$in_a(k)$	Quantité de données produites par la k^{e} phase de la tâche t d'un CSDFG dans le buffer $a = (t, t')$	page 25
o_a	Total des données lues par la tâche t durant son itération dans le buffer $a = (t, t')$	page 25
i_a	Total des données écrites par la tâche t durant son itération dans le buffer $a = (t, t')$	page 25
$O_a\langle t', n' \rangle$	Total des données lues dans un buffer a jusqu'à l'exécution $\langle t', n' \rangle$ incluse	page 59
$I_a\langle t, n \rangle$	Total des données écrites dans un buffer a jusqu'à l'exécution $\langle t, n \rangle$ incluse	page 59
a	Notation le plus souvent utilisée pour désigner un buffer $a = (t, t')$	page 23
a'	Notation le plus souvent utilisée pour décrire l'arc retour modélisant l'espace disponible dans un buffer de taille bornée. Aussi appelé *feedback buffer*	page 47
k	Notation le plus souvent utilisée pour désigner l'indice d'une phase	page 25
t	Notation le plus souvent utilisée pour désigner la source d'un buffer $a = (t, t')$	page 23
n	Notation généralement utilisée pour désigner l'indice d'exécution d'une tâche d'un SDFG ou de l'itération d'une tâche d'un CSDFG lorsqu'elle est la source d'un buffer $a = (t, t')$	page 35
k	Notation le plus souvent utilisée pour désigner l'indice d'une phase de la source d'un buffer	page 25

t'	Notation le plus souvent utilisée pour désigner la cible d'un buffer $a = (t, t')$	page 23
n'	Notation généralement utilisée pour désigner l'indice d'exécution d'une tâche d'un SDFG ou de l'itération d'une tâche d'un CSDFG lorsqu'elle est la cible d'un buffer $a = (t, t')$	page 35
k'	Notation le plus souvent utilisée pour désigner l'indice d'une phase de la cible d'un buffer	page 25
n	Notation généralement utilisée pour désigner l'indice d'exécution d'une tâche d'un SDFG ou de l'itération d'une tâche d'un CSDFG	page 35
t	Notation le plus souvent utilisée pour désigner une tâche	page 23
$d(t)$	Durée de l'exécution de la tâche t	page 36
$Fb(a)$	ensemble des arcs retour ou *feedback buffers* tels que $\forall a = (t, t') \in \mathcal{A}\ \exists a' = (t', t) \in Fb(\mathcal{A})$	page 47
$\gcd_a$	PGCD entre les taux de production et de consommation des itérations	page 39
$\mathcal{G}$	Notation générique d'un graphe dataflow tel que $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 23
$\Omega_{\mathcal{G}}^{\mathcal{S}}$	Période normalisée d'un ordonnancement $\mathcal{S}$ pour le graphe $\mathcal{G}$	page 58
$\phi(t)$	Désigne le nombre de phases d'initialisation de la tâche $t \in \mathcal{T}$ d'un ICSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 25
$\mathcal{A}^+(t)$	Ensemble des arcs entrants d'une tâche $t \in \mathcal{A}$	page 40
$\mathcal{U}(a)$	Ensemble des couples de phases nécessaires pour définir les contraintes k-périodiques d'un buffer a	page 80
$\mathbb{N}$	Ensemble des entiers positifs	page 40
Z_t	Taux normalisé d'une tâche t	page 40
$\mathcal{A}^-(t)$	Ensemble des arcs sortants d'une tâche $t \in \mathcal{A}$	page 40
$K_t^{\mathcal{G}}$	Facteur de périodicité de la tâche t	page 71
$K^{\mathcal{G}}$	Vecteur de périodicité	page 71
t_k	Désigne la k^{e} phase de la tâche t d'un CSDFG	page 57
$\varphi(t)$	Désigne le nombre de phase de la tâche $t \in \mathcal{T}$ d'un CSDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 25
$Pr\langle t, n\rangle$	Désigne l'exécution précédente de $\langle t, n\rangle$	page 59
$M_0(a)$	Marquage initial d'un buffer a	page 23
$\mathbb{Q}$	Ensemble des rationnels	page 81
$\mathbb{R}$	Ensemble des réels	page 37
$N_t^{\mathcal{G}}$	Facteur de répétition de la tâche t	page 39
$N^{\mathcal{G}}$	Vecteur de répétition	page 39
$\mathcal{S}$	Notation généralement utilisée pour désigner un ordonnancement.	page 35
$\langle t, n\rangle$	Désigne la $n$ieme exécution de la tâche $t \in \mathcal{T}$ d'un SDFG $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 35
out_a	Taux de lecture d'un buffer a pour un SDFG	page 23
thr_a	Seuil d'exécution d'un buffer a	page 24

thr_a^*	Seuil d'exécution utile du buffer a d'un SDFG.	page 124
in_a	Taux d'écriture d'un buffer a pour un SDFG	page 23
$\mathcal{S}\langle t, n\rangle$	Désigne la date de départ de l'exécution $\langle t, n\rangle$ pour un ordonnancement $\mathcal{S}$	page 35
step_a	PGCD entre les taux de production et de consommation des phases	page 40
$\mu_t^{\mathcal{S}}$	Période de fonctionnement de la tâche t pour un ordonnancement $\mathcal{S}$	page 37
$\mathcal{T}$	Ensemble des tâches d'un dataflow tel que $\mathcal{G} = (\mathcal{T}, \mathcal{A})$	page 23
$Th_t^{\mathcal{S}}$	La fréquence de fonctionnement d'une tâche t pour un ordonnancement $\mathcal{S}$	page 35
$\mathbb{1}$	Vecteur de périodicité 1-périodique	page 86
$M_0^*(a)$	Marquage utile du buffer a	page 124
$\mathbb{Z}$	Ensemble des entiers relatifs	page 40