

THÈSE / ENS CACHAN - BRETAGNE

sous le sceau de l'Université européenne de Bretagne

pour obtenir le titre de
DOCTEUR DE L'ÉCOLE NORMALE SUPÉRIEURE DE CACHAN
Mention : Informatique
École doctorale MATISSE

présentée par

Antoine Morvan

Préparée à l'Unité Mixte de Recherche 6074
Institut de recherche en informatique
et systèmes aléatoires

Utilisation du modèle polyédrique pour la synthèse d'architectures pipelinées

Thèse soutenue le 28 juin 2013
devant le jury composé de :

Tanguy RISSET
Professeur à l'INSA de Lyon / *président*

Jean-Marc DELOSME
Professeur à l'Université d'Evry Val d'Essonne / *rapporteur*
Philippe CLAUSS
Professeur à l'Université de Strasbourg / *rapporteur*

Erven ROHOU
Directeur de recherche, INRIA-Rennes / *examinateur*
Thierry MICHEL
Concepteur matériel sénior, STMicroelectronics / *examinateur*
Christophe ALIAS
Chargé de recherche, ENS de Lyon / *examinateur*

Patrice QUINTON
Professeur à l'ENS Cachan - Bretagne / *directeur de thèse*
Steven DERRIEN
Professeur à l'Université de Rennes 1 / *co-directeur de thèse*

If you want to go somewhere,
goto is the best way to get there.
Kenneth Thompson

Remerciements

Toujours là pour un bon conseil, une revue agressive de chapitre, un barbecue arrosé (à l'eau !), ou un commit destructeur, Steven a été un super encadrant durant ma thèse. Je le remercie énormément pour tout ce qu'il a fait durant ces ~~3~~ 4 ans.

Retrouver mon directeur autour d'un café n'était pas vraiment aisé, au vu des tâches administratives qui lui incombaient. Néanmoins, Patrice a toujours répondu à mes questions, avec un regard certes plus distant, mais toujours avisé. Je le remercie pour la disponibilité qu'il a su créer, ses remarques, ses conseils, et ses encouragements lors de la rédaction.

Offrant leur temps, les membres du jury m'ont fait l'honneur de venir juger mes travaux. Je les remercie pour avoir accepté cette tâche et d'avoir fait le déplacement.

L'équipe Cairn a su m'accueillir dès mes débuts en tant que stagiaire. Je remercie tous ses membres, permanents et contractuels, présents ou anciens, pour tous les conseils et toute l'aide qu'ils ont pu m'apporter.

Le Cairn Spirit est un fléau qui s'abat contre le sérieux. Je remercie tous ceux qui contribuent à perpétrer cet esprit, indispensable pour obtenir une bonne ambiance dans une équipe et dans une communauté.

Faire de la recherche se fait généralement dans un laboratoire. J'ai eu la chance de réaliser mes travaux à l'Irisa, un grand labo qui connecte des gens de tous les horizons. Merci au personnel et aux visiteurs, chercheurs ou non, pour les conseils sur mes travaux et pour m'avoir ouvert l'esprit et les yeux sur notre monde.

On est souvent enfoncé jusqu'au cou dans sa thèse après 4 ans ~~d'esclavage~~ de travail intensif. J'avoue, ce n'est pas moi qui est éliminé la majorité des typos du manuscrit. Plus concentré sur la forme que sur le fond, c'est ma famille qui en a eu la charge. Je les remercie pour ce travail, mais surtout pour avoir cru en moi, et m'avoir encouragé et soutenu durant ces 4 ans.

Rester sur Rennes le weekend est devenu plus facile quand j'ai eu ma machine à laver. Mais c'est surtout devenu beaucoup plus sympa avec toutes les personnes que j'ai rencontrées, et qui me sortent ~~de ma caverne~~ la tête de mes travaux. Merci aux rennais-es.

Entre deux semaines de boulot, mes escapades m'ont permis de buller pendant certains weekends. Merci aux costarmoricain-e-s, breton-ne-s, parisien-e-s, toulousain-e-s, clermontois-es, avignonnais-es, américain-e-s, indien-ne-s, russes et à tous ceux que j'oublie.

Vers le début de matinée, peu après 10h42, et en fin d'après-midi, vers 16h42, les pauses café régulent la vie de l'équipe. C'est le moment idéal pour se moquer des blagues pourries, du Ouest-France, ou des thésards en pleine rédaction... Merci pour ces moments de détente.

Entre midi et 2h, les stagiaires, les ingénieurs, les doctorants, les docteurs, et les gros se retrouvent pour casser la croûte. C'est le moment idéal pour se changer les idées avec un ~~débat~~ troll super intéressant, pour savoir si le nucléaire est vraiment si bien que ça, si c'est vraiment Gohan le héros de Dragon Ball Z, ou encore si la patate de l'espace c'est vraiment la meilleure *drone bay*. Merci pour tous ces ~~débats~~ trolls.

Rétrospectivement, ce long tunnel de labeur qu'est la thèse a été énormément éclairé par toutes les rencontres que j'ai pu voir autour d'un café, d'un ~~débat~~ troll, d'une bière, d'un jeu... Encore une fois, je remercie toutes ces personnes.

Table des matières

Table des matières **i**

Introduction **1**

1 Accélérateurs matériels spécialisés pour systèmes embarqués **7**
1.1 Introduction . . . 7
1.2 Applications et architectures embarquées . . . 7
1.2.1 Quelques exemples d'applications embarquées . . . 8
1.2.2 Critères de performance . . . 8
1.2.3 Solutions architecturales . . . 9
1.2.4 Contraintes économiques . . . 12
1.3 Conception de systèmes embarqués . . . 13
1.3.1 Description comportementale de l'application . . . 13
1.3.2 Conception conjointe logicielle-matérielle . . . 15
1.3.3 Conception d'accélérateurs matériels spécialisés . . . 15
1.4 Conclusion . . . 17

2 Synthèse de haut niveau d'accélérateurs spécialisés **19**
2.1 Introduction . . . 19
2.2 Synthèse d'accélérateur matériel à partir d'une description comportementale . . . 20
2.2.1 Compilation de la description comportementale . . . 20
2.2.2 Ordonnancement, allocation et placement . . . 21
2.2.3 Génération de l'architecture . . . 23
2.3 Optimisations pour une synthèse d'accélérateurs performants . . . 24
2.3.1 Réduction de la complexité des calculs . . . 24
2.3.2 Transformation des structures de données, des ressources de stockage et des accès à la mémoire . . . 25
2.3.3 Exploitation du parallélisme . . . 28
2.4 Conclusion . . . 31

3 Modèle polyédrique et synthèse de haut niveau **35**
3.1 Introduction . . . 35
3.2 Flot global de transformation . . . 36

3.2.1 Représentation des programmes dans le modèle polyédrique 37
3.2.2 Spécification et application des transformations 38
3.2.3 Génération de code 40
3.3 Légalité des transformations 41
3.3.1 Représentation des dépendances de données 41
3.3.2 Transformations des dépendances et condition de légalité 43
3.3.3 Analyse des dépendances de données 44
3.3.4 Extensions 46
3.4 Transformations automatiques 49
3.4.1 Le cas monodimensionnel 49
3.4.2 Le cas multidimensionnel 50
3.4.3 Transformations optimisantes 51
3.5 Application du modèle polyédrique à la HLS 52
3.5.1 MMalpha 52
3.5.2 Réseau de processus polyédriques 53
3.5.3 Optimisation des accès mémoire 56
3.6 Conclusion 57

4 Génération de contrôle pour le modèle polyédrique 59
4.1 Introduction 59
4.2 Parcours de polyèdres 59
4.2.1 Problématique et approches basiques 59
4.2.2 Génération de nids de boucles efficaces 62
4.2.3 Génération d'une machine à états 64
4.3 Génération de contrôleur matériel 65
4.3.1 ClooGVHDL 65
4.3.2 MMalpha 66
4.3.3 Réseau de processus polyédriques 67
4.3.4 Transformations source-à-source et outils de HLS 67
4.4 Amélioration de la génération de contrôle 68
4.4.1 Structure des boucles 68
4.4.2 Représentation des domaines et relations polyédriques 69
4.4.3 Réduction de force 70
4.4.4 Analyse de taille de type 71
4.5 Conclusion 72

5 Amélioration de l'applicabilité du pipeline de nid de boucles 73
5.1 Introduction 73
5.2 Pipeline de nid de boucles en HLS 74
5.2.1 Pipeline de boucles 75
5.2.2 Le surcoût de la latence du pipeline 76
5.2.3 Pipeline de nid de boucles : principe 76

5.3 Approches existantes . . . 78
5.3.1 Pipeline de boucle en synthèse de matériel . . . 78
5.3.2 Pipeline de nid de boucles . . . 79
5.3.3 Correction de transformations de boucles illégales . . . 80
5.4 Analyse de la légalité . . . 80
5.4.1 Modèle de pipeline et condition de légalité . . . 80
5.4.2 Vérifier la légalité en estimant la distance de réutilisation . . . 81
5.4.3 Construction de la fonction $next^{\Delta}_{\mathcal{D}}(\vec{x})$ . . . 83
5.4.4 Construction de l'ensemble des violations de dépendances . . . 86
5.4.5 L'algorithme de vérification de légalité . . . 87
5.5 Insertion de bulles polyédriques . . . 87
5.5.1 Complétion simple . . . 89
5.5.2 Complétion optimisée . . . 89
5.5.3 Complétion itérative . . . 92
5.5.4 Extension à l'insertion de bulles sur la boucle interne . . . 93
5.6 Conclusion . . . 94

6 Implémentation et résultats **95**
6.1 Introduction . . . 95
6.2 GeCoS : Generic Compiler Suite . . . 95
6.2.1 Nano2012-S2S4HLS . . . 96
6.2.2 Compilateur source-à-source . . . 98
6.2.3 Ingénierie dirigée par les modèles . . . 98
6.2.4 `ompVerify` . . . 99
6.2.5 Environnement de manipulation de boucles . . . 100
6.3 Résultats expérimentaux . . . 104
6.3.1 Protocole expérimental . . . 104
6.3.2 Résultats qualitatifs . . . 105
6.3.3 Résultats quantitatifs . . . 107
6.4 Conclusion . . . 109

Conclusion **113**

Glossaire **119**

Publications personnelles **121**

Bibliographie **123**

Introduction

Contexte général

Depuis leur apparition, les systèmes électroniques n'ont cessé d'être miniaturisés. Sur une même surface de silicium on a ainsi pu accroitre le nombre de transistors. Ce progrès technologique permet non seulement d'implanter plus de fonctions sur un même circuit, mais aussi d'accélérer l'exécution des calculs et de diminuer la consommation énergétique. À l'heure actuelle, de nombreux systèmes informatiques sont embarqués dans des appareils mobiles produits à grande échelle, pour lesquels des contraintes très fortes s'appliquent en termes de coûts de production et de conception, de performance et de consommation.

Trois critères déterminent essentiellement la viabilité d'un système embarqué. Il faut qu'ils soient performants (vitesse, consommation, fonctionnalités) ; leurs coûts de production doivent être les plus faibles possibles ; enfin leur temps de conception doit permettre une mise sur le marché dans un temps très réduit.

Pour satisfaire ces contraintes, les fabricants de systèmes disposent d'un vaste choix de technologies, allant du processeur programmable au circuit spécialisé. En utilisant des plateformes hétérogènes, intégrées sur des Soc (*System on Chip* : Système sur Puce), il est possible de tirer parti des avantages de chacune des architectures tels que la flexibilité des processeurs programmables ou l'efficacité énergétique des accélérateurs spécialisés.

Accélérateurs matériels spécialisés pour systèmes embarqués

Les tâches consommatrices de ressources de calcul (par exemple le décodage d'un flux H264) nécessitent plus qu'un processeur programmable pour respecter les contraintes de performance. En effet leur coût de production élevé ainsi que leur faible efficacité énergétique sont le prix à payer pour conserver la flexibilité d'un jeu d'instruction. Pour être compétitifs, les systèmes sur puce intègrent donc des accélérateurs spécialisés, dédiés à certaines tâches très consommatrices en ressources de calcul. Par exemple la plupart des *Smartphones* embarquent des décodeurs H264 implémentés sous forme d'accélérateurs spécialisés.

Ces accélérateurs spécialisés offrent une efficacité énergétique nettement supérieure à ce que proposent les processeurs à jeu d'instructions, allant jusqu'à trois ordres de grandeur [1]. Cette efficacité est cependant le résultat d'une étape de conception longue et fastidieuse, pendant laquelle une spécification algorithmique, décrite dans des langages de haut niveau (tels que

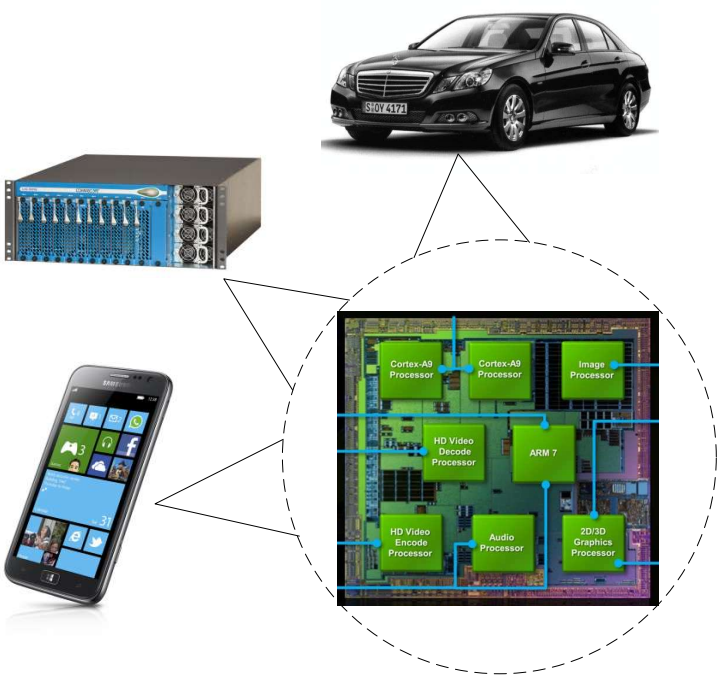


FIGURE 1 – Exemples de systèmes embarqués (Smartphone, nœud réseau, automobile), et illustration d'un système sur puce hétérogène (Tegra 2 de NVidia).

C/C++), est traduite manuellement dans un langage de description matérielle, tel que VHDL (*Very-high-speed integrated circuits Hardware Description Language*) ou Verilog.

Flot de conception d'un accélérateur spécialisé

Le processus de conception se décompose principalement en deux étapes. Comme le présente la figure 2, la première consiste à traduire la spécification algorithmique en une description matérielle au niveau RTL (*Register Transfert Level* : Niveau de Transferts de signaux entre des Registres). La seconde étape, aujourd'hui systématiquement réalisée de manière automatique par les outils de synthèse logique, traduit la description matérielle en un réseau de portes ou transistors.

Cette première étape est encore souvent réalisée manuellement par des experts. Elle ne consiste pas simplement en une traduction des algorithmes, mais aussi en une transformation de leurs spécifications, de manière à permettre une exécution parallèle, améliorer la localité des accès mémoire, ou réduire la complexité de certaines opérations. À l'heure actuelle la complexité des applications incite les concepteurs à utiliser des outils d'aide à la conception afin de respecter les délais de mise sur le marché.

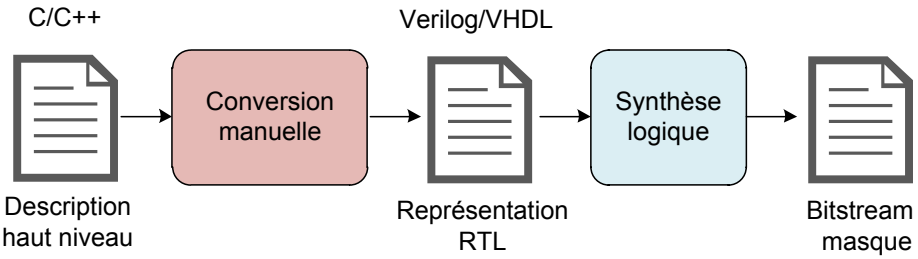


FIGURE 2 – Flot de conception pour la synthèse de matériel à partir d'une description dans un langage de haut niveau.

Synthèse de haut niveau d'accélérateurs spécialisés

Ces outils d'aide à la conception opèrent directement sur des programmes C/C++, et produisent une description matérielle en VHDL ou Verilog. Cette opération est appelée synthèse de haut niveau. Ces outils de HLS (*High-Level Synthesis* : Synthèse de Haut Niveau) permettent des gains de productivité importants (d'un facteur allant de 5 à 10 d'après les industriels), et sont aujourd'hui utilisés en production. Même s'ils offrent aux concepteurs la possibilité de choisir finement les caractéristiques de l'architecture générée, ces outils ont une efficacité qui dépend énormément de la façon dont les algorithmes en C/C++ sont spécifiés : ils sont bien souvent incapables de transformer efficacement les programmes pour exploiter parallélisme et localité.

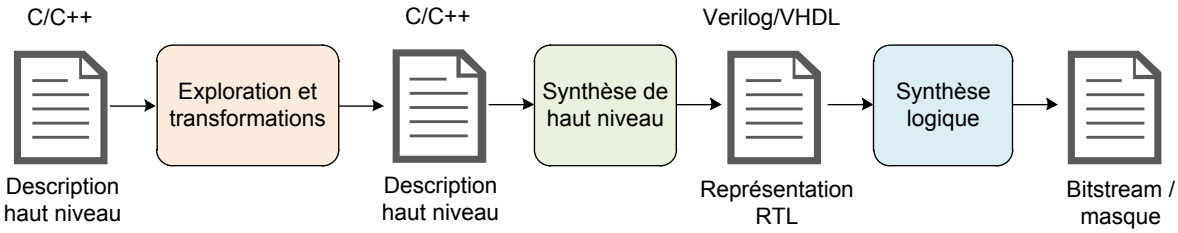


FIGURE 3 – Lors de l'intégration d'outils de HLS dans le flot de conception, l'exploration de l'espace de conception se fait sur la description à haut niveau.

La conception d'un accélérateur matériel spécialisé passe donc souvent par une étape d'exploration, réalisée en réécrivant le programme C/C++. L'objectif de cette exploration est de simplifier la tâche de l'outil de HLS et d'obtenir ainsi un accélérateur matériel plus performant.

Exploration de l'espace de conception

À partir d'une spécification à haut niveau, dans des langages tels que C/C++, il faut faire apparaitre du parallélisme de manière explicite et transformer les accès mémoire de manière à améliorer leur localité. Les outils de HLS proposent aux concepteurs des annotations, généralement sous la forme de `#pragmas`, pour ajouter cette sémantique absente dans les langages. On retrouve par exemple des annotations pour dérouller des boucles, qui vont permettre de vectori-

ser l'exécution du corps de la boucle ; indiquer à l'outil de HLS qu'une boucle doit être exécutée de manière pipelinée ; ou encore spécifier qu'un tableau doit être implanté dans des registres au lieu d'une mémoire.

La transformation de pipeline de boucle (*loop pipeline* ou *software pipeline*) est une transformation clé lors de la mise en œuvre matérielle d'une application. Le principe est de découper le corps de la boucle en n étapes, exécutées par n étages de pipeline (*pipeline stages*), chaque étape étant exécutée en un cycle, puis d'entrelacer l'exécution de n itérations du corps de la boucle à un instant donné (cf. exemple de la figure 4). Cette transformation permet de diminuer significativement le temps d'exécution. Il faut cependant s'assurer au préalable qu'aucune dépendance n'est violée lors de l'application de cette transformation.

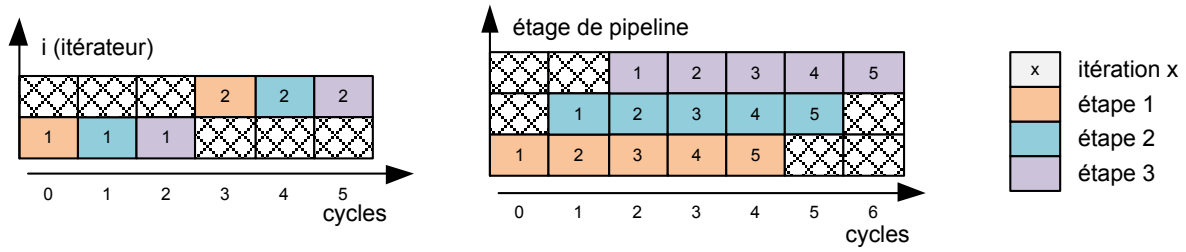


FIGURE 4 – Exécution séquentielle d'une boucle (à gauche). Lors du pipeline de cette boucle (à droite), le corps est découpé en trois étapes. Il est alors possible d'entrelacer l'exécution de trois itérations simultanément.

Compilation source-à-source pour la synthèse de haut niveau

Les industriels utilisent aujourd'hui ces outils de HLS en production. Cependant, les limitations de ces outils obligent les concepteurs à restructurer et annoter les programmes sources en C/C++. Dans le cadre du programme de recherche Nano2012, le projet S2S4HLS (*Source-To-Source For High-Level Synthesis* : source-à-source pour la synthèse de haut niveau), soutenu par Inria et STMicroelectronics, vise à automatiser cette étape d'exploration en utilisant des transformations source-à-source, comme le présente la figure 5

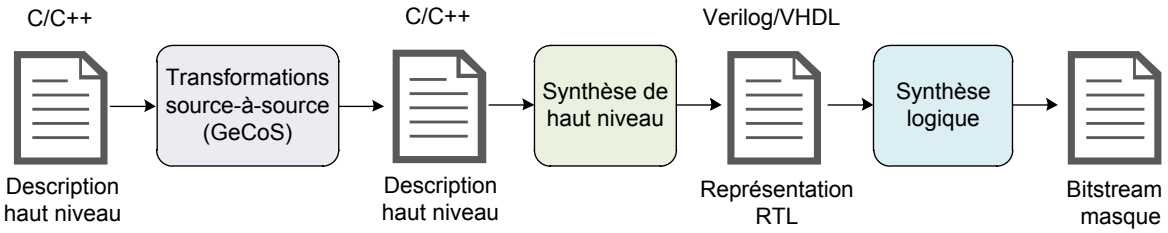


FIGURE 5 – Afin de réduire les temps de conception, des transformations peuvent être appliquées automatiquement sur la description à haut niveau de l'application. Dans un flot source-à-source, l'outil de HLS travaille sur une description optimisée pour la synthèse.

Les compilateurs source-à-source génèrent du code dans le même langage que celui utilisé en entrée. Ils permettent d'appliquer des transformations de haut niveau tout en restant indépendants des outils de compilation ou de HLS utilisés en aval. Plusieurs outils (Pluto [2], CETUS [3], Rose [4]) ont déjà montré l'intérêt des compilateurs source-à-source dans le cadre de la parallélisation automatique pour machines parallèles.

L'objectif de la compilation source-à-source, dans le cadre de la HLS, est de transformer le programme original pour faire apparaitre automatiquement des structures que l'outil de HLS peut facilement exploiter.

Le projet S2S4HLS regroupe trois sous-projets, visant à :

- transformer les boucles pour faciliter une mise en œuvre matérielle performante ;
- déterminer des tailles idéales pour les types à virgule fixe à partir d'une spécification en virgule flottante ;
- détecter des motifs dans les graphes de données, de manière à accélérer l'étape d'ordonnancement et utiliser des macro-opérateurs.

Les travaux de cette thèse interviennent dans le sous-projet 1, dédié aux transformation de boucles. En particulier, les travaux présentés permettent d'étendre l'applicabilité du pipeline de boucle mentionné ci-dessus.

Amélioration de l'applicabilité du pipeline de nid de boucles

En utilisant la représentation des nids de boucles dans le modèle polyédrique, nous avons mis au point une technique d'analyse de la légalité du pipeline de nids de boucles. Cette technique permet, en trois temps :

- d'évaluer rapidement de manière imprécise (mais conservative) la légalité du pipeline de nid de boucle ;
- de vérifier précisément la légalité lorsque la première étape échoue ;
- de corriger un pipeline a priori illégal, en insérant des états d'attente.

De plus, en se basant sur la technique de génération de code polyédrique proposée par BOULET et FEAUTRIER [5], il est possible de générer une boucle mise à plat qui facilite l'application du pipeline de nid de boucles par les outils de HLS.

Pour les expérimentations et pour rester autant que possible indépendant des outils de HLS, nos méthodes ont été implémentées dans l'infrastructure de compilation source-à-source GECOS [6].

Plan de la thèse

Ce mémoire est structuré en six chapitres qui développent les idées présentées ci-dessus. Les chapitres 1 à 3 décrivent le contexte de nos travaux ainsi que les connaissances nécessaires à leur compréhension. Les chapitres 4 à 6 détaillent les contributions.

Le chapitre 1 présente le contexte des systèmes embarqués et les contraintes qui portent sur la conception de leur plateforme matérielle. Ce chapitre présente les raisons qui poussent les concepteurs des plateformes matérielles pour systèmes embarqués à utiliser des accélérateurs matériels dédiés à l'exécution des tâches critiques. Le chapitre 2 présente quant à lui les intérêts liés à l'utilisation de la HLS dans un flot de conception d'une plateforme matérielle, ainsi que les limitations des outils de HLS.

Les recherches décrites dans cette thèse sont basées sur la représentation des nids de boucles dans le modèle polyédrique. Le chapitre 3 rappelle les principes de ce modèle, ainsi que les techniques d'analyse de dépendances qui sont au cœur de nos travaux.

Le chapitre 4 est consacré à la génération de code à partir d'une représentation polyédrique d'un nid de boucles. Cette étape du flot de compilation détermine la façon dont le contrôleur matériel parcourt le nid de boucles, en particulier la façon de calculer les indices des boucles. Deux approches [7, 5] à ce problème sont présentées dans ce chapitre. La plus utilisée dans le contexte de la compilation logicielle génère un programme sous la forme de nids de boucles. La deuxième dérive une machine à états qui détermine la valeur des indices pour la prochaine itération en fonction de l'itération courante. Nous montrons comment ces techniques peuvent être utilisées pour générer des contrôleurs matériels efficaces.

La technique de génération de code qui permet de dériver une machine à état offre l'avantage de mettre à plat le nid de boucles, c'est-à-dire que le programme ne comporte qu'un seul niveau de boucle. Avec un nid de boucle aplati, les outils de HLS peuvent facilement mettre en œuvre un pipeline de nid de boucle. Il faut cependant s'assurer que le pipeline n'introduit pas de violation de dépendance de données.

Le chapitre 5 présente le test de légalité qui permet de déterminer, à la compilation, si une mise en œuvre pipelinée d'un nid de boucles introduit de telles violations de dépendances. Après avoir rappelé les principes du pipeline de boucles en HLS, nous détaillons les trois étapes de notre approche.

Le chapitre 6 présente les développements logiciels effectués pendant cette thèse. On y présente l'infrastructure de compilation source-à-source GeCoS, l'intérêt d'un compilateur source-à-source et l'utilisation des techniques d'ingénierie dirigée par les modèles dans sa mise en œuvre. Les deux principales réalisations sont les suivantes :

- `ompVerify` : un greffon Eclipse qui permet de vérifier la légalité des annotations OpenMP insérées dans les programmes C/C++, via des retours instantanés dans l'éditeur ;
- S2S4HLS-SP1 : le sous-projet 1 (SP1) du projet source-à-source pour la HLS (S2S4HLS) du programme de recherche Nano2012, qui consiste à mettre en œuvre un environnement de transformation de boucles source-à-source.

Les méthodes présentées dans les chapitres 4 et 5 ont été implémentées dans l'environnement S2S4HLS-SP1, et les résultats obtenus sont exposés à la fin de ce dernier chapitre.

Chapitre 1

Accélérateurs matériels spécialisés pour systèmes embarqués

1.1 Introduction

Les systèmes embarqués sont présents partout dans notre quotidien. Les applications qu'ils exécutent fournissent des services qui couvrent des domaines de plus en plus variés, allant du décodage d'une vidéo sur un téléphone mobile à la vérification de paquets dans les infrastructures réseaux. Avec le temps, la demande en puissance de calcul de ces applications embarquées a fortement augmenté et continue sa progression. Par exemple, les vidéos sont encodées dans des définitions de plus en plus élevées (HD, Full HD, Quad HD), parfois avec plusieurs points de vue (3D). Pour répondre à ces besoins applicatifs, les systèmes embarqués doivent faire appel à des plateformes matérielles de plus en plus puissantes.

Ce chapitre a pour but de décrire plus en détail ce qu'est un accélérateur matériel pour un système embarqué. Dans la section 1.2 nous présentons les caractéristiques des domaines d'application de l'embarqué, ainsi que les choix architecturaux possibles pour leurs mises en œuvre en matériel. En particulier nous précisons les raisons qui motivent l'utilisation d'accélérateurs matériels spécialisés pour mettre en œuvre certaines fonctionnalités du système. La section 1.3 s'intéresse quant à elle aux moyens utilisés pour spécifier le comportement de l'application, identifier les parties de l'application qui pourraient nécessiter une accélération matérielle, et spécifier la structure de la plateforme matérielle qui va exécuter l'application.

1.2 Applications et architectures embarquées

Dans cette section, nous présentons quelques exemples significatifs d'applications embarquées, puis nous énumérons les critères qui doivent être pris en compte pour leurs performances. Les choix qui s'offrent aux concepteurs en termes d'architectures matérielles sont présentés à la fin de cette section.

1.2.1 Quelques exemples d'applications embarquées

Les applications typiques des systèmes embarqués couvrent des domaines variés. Les besoins des applications sont spécifiques à chaque domaine, mais possèdent tout de même certaines caractéristiques communes. Voici quelques exemples de domaines d'applications.

Multimédia : celles-ci consistent par exemple à encoder et décoder des flux audio et vidéo. Ce domaine d'applications requiert de plus en plus de performance à cause de la croissance de la résolution des flux vidéo [8], de l'augmentation du nombre d'images par seconde, de l'utilisation de la stéréovision [9], ou encore de la multiplication des flux audio.

Communication : avec l'essor d'internet, ainsi que le succès des *datacenters*, les infrastructures de télécommunication doivent fournir de plus en plus de débit, avec une latence de plus en plus faible.

Sécurité : l'essor d'internet soulève aussi des questions concernant la sécurité des informations qui y transitent. Étant donné l'accroissement des débits dans les réseaux, la demande en ressources de calcul pour le chiffrement et le déchiffrement des données s'accroît proportionnellement.

1.2.2 Critères de performance

Les trois domaines d'applications décrits précédemment ont des besoins élevés en ressources de calcul. De plus, et en particulier dans un contexte mobile, la plateforme matérielle qui exécute l'application doit fournir ces ressources de calcul avec une consommation énergétique la plus faible possible. Ces plateformes matérielles doivent aussi respecter certaines contraintes commerciales, de maintenance, ou encore certaines normes. Les cinq principales contraintes sont décrites ci-dessous.

Puissance de calcul : les applications demandent une grande puissance de calcul pour respecter des débits ou des échéances dans le temps réel [10]. Par exemple, une vidéo doit être décodée à 24 images par seconde. La plateforme matérielle doit être correctement dimensionnée pour soutenir ce débit. La puissance de calcul est généralement mesurée en nombre maximal de Mops (*Mega Operations Per Second* : Million d'Opérations Par Seconde) atteignable théorique.

Consommation d'énergie : la consommation d'énergie de la plateforme matérielle définit son autonomie dans un contexte mobile. Par exemple, un appareil mobile permettant de décoder une vidéo doit pouvoir le faire pendant au moins la durée de la vidéo avant que ses batteries ne soient vides. L'environnement est aussi une préoccupation de plus en plus importante et sa préservation passe par des économies d'énergie.

Coûts de production : dans un contexte commercial, le coût de production doit être limité pour que le produit final soit compétitif. Le coût de production de la plateforme matérielle est lié au nombre de ressources (calcul, contrôle, stockage, ...) mises en œuvre. Le coût est aussi indirectement influencé par la consommation énergétique du système. En effet, une consommation élevée demande une source d'énergie adaptée (batterie de plus grande capacité) et impacte la dissipation thermique (système de refroidissement plus conséquent).

Évolutivité : les applications sont sujettes à évolution. Par exemple les normes des protocoles réseaux évoluent pour fiabiliser les connexions ou améliorer les débits ; les standards multimédias changent dans l'optique d'améliorer la qualité des flux ou corriger des défauts. Reconstruire un système complet pour suivre l'évolution d'une norme n'est pas envisageable.

Fiabilité : dans certains domaines d'applications, un défaut peut avoir des conséquences très importantes (vies humaines). La fiabilité de la plateforme matérielle est alors un critère important.

1.2.3 Solutions architecturales

En pratique, l'application embarquée est installée sur une puce unique, appelée Soc (*System on Chip* : Système sur Puce), sur laquelle un système complet est installé (cf. figure 1.1). Le SoC intègre tous les composants matériels et logiciels nécessaires à l'execution de l'application et à la fourniture des services.

Parmi ces composants se trouvent des unités de calcul (cf. ci-dessous), des mémoires, les interfaces, ainsi que les mécanismes de communication internes (bus, *crossbar*, NoC [1]) et externes (communications réseau, contrôleurs pour mémoires ou périphériques externes).

Un Soc peut être implémenté à l'aide d'une des deux technologies suivantes :

- la technologie Asic (*Application-Specific Integrated Circuit* : Circuit Intégré Spécifique à une Application) : ce sont des circuits intégrés pour lesquels les transistors sont gravés dans le silicium. Ils offrent la meilleure densité en termes de surface silicium par MOPS, mais une fois gravé le circuit n'est pas modifiable.
- la technologie FPGA (*Field Programmable Gate Array* : Réseau de Portes Programmables) : ce sont des circuits dans lesquels la logique est reprogrammable. Cette caractéristique permet d'avoir des circuits flexibles, mais les performances sont moindres que celles obtenues par un Asic [11].

Un Soc implémenté sur Asic peut aussi intégrer un composant reconfigurable, appelé eFPGA (*embedded* FPGA). Pour atteindre les performances recherchées, il est utile de combiner différentes approches architecturales que nous détaillerons dans la suite (cf. figure 1.2) :

- les eCPU (*Embedded Central Process Unit* : Processeur Programmable Embarqué) ;
- les architectures *many-cores* embarquées ;

1. *Network on Chip* : réseau sur puce.

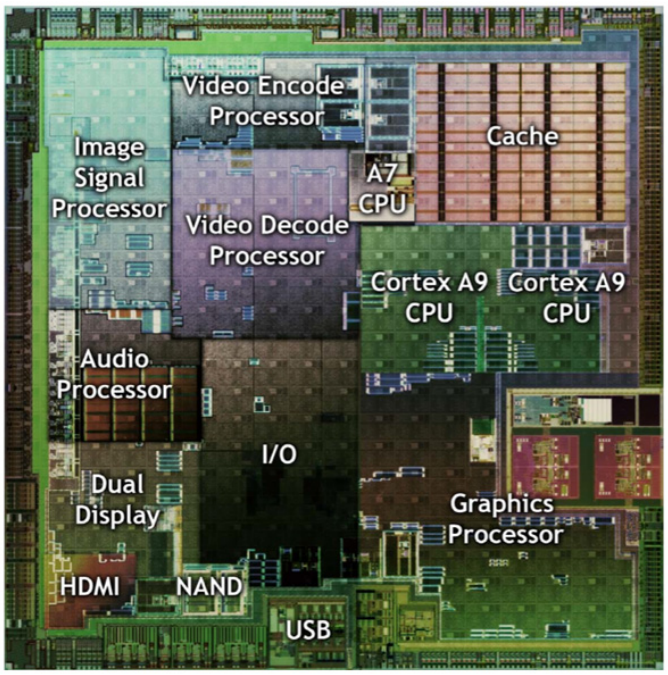


FIGURE 1.1 – Illustration d'un système sur puce basse consommation Tegra 2 de NVidia, destiné au marché des *Smartphones*. La même puce contient deux ARM Cortex A9, un ARM Cortex A7, associés à plusieurs blocs matériels spécialisés, un cache et des interfaces pour un total de 260 millions de transistors.

- les Asip (*Application-Specific Instruction-set Processor* : Processeur à jeu d'Instructions Spécialisées pour une Application) ;
- les CGRA (*Coarse-Grain Reconfigurable Architecture* : Architecture Reconfigurable à Gros Grain) ;
- les blocs IP (*semiconductor Intellectual Property core/block* : bloc IP matériel)

Les processeurs programmables embarqués (eCPU, par exemple les ARM Cortex A9 [12]), aussi appelés microprocesseurs, sont des composants dont le calcul est déterminé par un programme codé dans un jeu d'instructions et stocké dans une mémoire. Ils sont très flexibles car il suffit de modifier le programme en mémoire pour modifier l'application, mais c'est au prix d'une faible puissance de calcul. Bien que certaines solutions permettent d'accélérer l'exécution d'un programme dans les processeurs dits super scalaires, la consommation énergétique augmente considérablement, ainsi que le coût en ressources matérielles.

À l'instar du HPC (*High-Performance Computing* : Calcul Haute Performance), les systèmes embarqués peuvent tirer parti du caractère massivement parallèle de certaines applications généralistes pour exploiter plusieurs processeurs, d'une manière similaires aux architectures *many-*

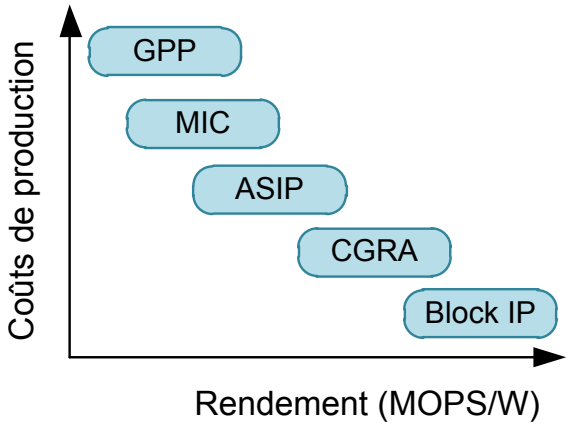


FIGURE 1.2 – Positionnement des différentes solutions architecturales en fonction de leur coût de production (fonction de la flexibilité de conception) et de leur efficacité énergétique (MOPS/W). Plus l'architecture est spécialisée, moins elle est flexible, mais plus son rendement est élevé.

cores (par exemple l'architecture STHORM de STMicroelectronics [13]). Leur flexibilité est inférieure à celle des eCPU, car leur conception requiert de porter une attention particulière au parallélisme et aux accès mémoire, mais leur efficacité énergétique est bien meilleure.

Les processeurs spécialisés (Asip, par exemple le processeur *soft-core*[2] Nios II de Altera) permettent de garder la flexibilité des processeurs généralistes tout en augmentant l'efficacité énergétique. Ces processeurs peuvent être adaptés à une application spécifique, proposer des jeux d'instructions spécialisés, par exemple les processeurs Dsp (*Digital Signal Processing* : Traitement du Signal Numérique) pour le traitement du signal, ou proposer des unités de calcul parallèle tels que les processeurs VLIW (*Very Long Instruction Word* : Mot d'Instruction Très Long). Les instructions spécialisées proposées par ces processeurs ont souvent une efficacité énergétique bien supérieure à leur émulation dans un jeu d'instructions standard. Déterminer quelles instructions doivent être ajoutées dans le processeur est un problème difficile [14]. De tels processeurs demandent aussi au compilateur d'être capable de générer du code exploitant le jeu d'instructions spécialisées.

Les architectures reconfigurables à gros grain (CGRA, par exemple l'architecture ROMA [15]) se situent entre les Asic, les FPGA, et les eCPU. Elles proposent en effet un circuit reconfigurable, au niveau opérateur, voire au niveau des blocs fonctionnels, et non plus au niveau des portes logiques comme les FPGA. Cette particularité leur permet d'atteindre une meilleure densité que les FPGA, d'être plus flexibles que les Asic, tout en offrant une meilleure efficacité énergétique que les processeurs programmables. L'implantation d'un calcul sur un CGRA est rendue difficile par la granularité des opérations offertes par l'architecture, sur laquelle l'application doit se

2. Les processeurs soft-core sont généralement destinés à une mise en œuvre sur FPGA.

« déployer ». En effet le compilateur doit identifier les parties de l'application correspondant aux blocs fonctionnels disponibles sur le CGRA, et maximiser l'utilisation de ces blocs [16].

Enfin, les blocs IP sont des composants dédiés à un seul traitement. Pour cette raison, leur efficacité énergétique est excellente, et elle peut atteindre 20 à 50 fois celle des processeurs programmables [17]. Cependant le temps de conception d'un circuit spécialisé est nettement plus important que celui nécessaire au développement d'un programme pour processeurs programmables (cf. section 1.3.3). Il est possible de gagner en flexibilité en implémentant les blocs IP sur des FPGA, mais au prix d'une baisse de performance importante [11].

1.2.4 Contraintes économiques

Les contraintes qui pèsent sur la conception de systèmes embarqués incluent non seulement un niveau de performance à atteindre, mais elles sont aussi liées au contexte commercial du système, c'est-à-dire le délai de mise sur le marché, la durée de vie du produit final, et les moyens mis en œuvre lors de sa conception. La durée de vie d'un système embarqué est généralement considérée comme suffisamment longue pour que le produit soit dépassé avant d'être hors service. Il faut donc minimiser les coûts de conception pour obtenir une plateforme matérielle respectant les contraintes de performances tout en minimisant les délais de mise sur le marché.

Les concepteurs de systèmes embarqués s'orientent aujourd'hui vers des plateformes matérielles hétérogènes. Une plateforme matérielle hétérogène est composée d'un ou plusieurs processeurs plus ou moins spécialisés, pour assurer la flexibilité du système. À ces processeurs généralistes sont associés plusieurs accélérateurs matériels pour améliorer les performances des tâches critiques (cf. section 1.3.2).

Les capacités des processus d'intégration des semi-conducteurs ne cessent de croitre, suivant la loi de MOORE [18]. Cependant, depuis 2004, la consommation énergétique des transistors ne diminue plus proportionnellement à leur taille. La conséquence directe est que pour une surface silicium donnée, il n'est plus possible de maintenir une consommation énergétique similaire d'une génération à la suivante [1].

Pour que l'augmentation de la consommation ne produise une surchauffe du circuit, les concepteurs sont aujourd'hui contraints de limiter l'utilisation des différents composants de l'architecture à un instant donné, phénomène appelé *utilization wall* (mur de l'utilisation). La partie d'une puce qui est active à un moment donné ne peut dépasser aujourd'hui environ 20% de sa surface totale, et pourrait se réduire à 7% d'ici 8 ans (cf. figure 1.3). Le reste de la puce, appelé *dark silicon* (silicium sombre), est maintenu au ralenti pour limiter la consommation globale.

Une des solutions proposées pour améliorer le taux d'utilisation matérielle d'un circuit est de spécialiser au maximum la puce afin d'atteindre une meilleure efficacité énergétique. Cependant spécialiser le matériel coûte cher en termes d'effort de conception (temps ou ressources humaines). En effet les étapes qui mènent d'un cahier des charges à un Soc nécessitent des développements importants, ainsi qu'une expertise pour atteindre les seuils de performance. Les temps de conception, lorsqu'ils sont longs, retardent la mise sur le marché du produit. Ce retard n'est pas toujours acceptable dans un contexte commercial.

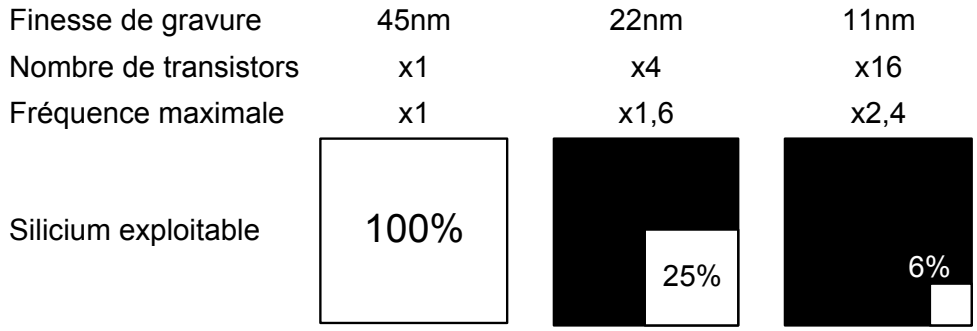


FIGURE 1.3 – Évolution du nombre de transistors et de la fréquence maximale d'un circuit en fonction de la finesse de gravure, pour une même surface de silicium. En augmentant le nombre de transistors, l'énergie consommée par la puce augmente. Pour maintenir une puissance similaire lorsque la finesse de gravure évolue, il faut diminuer le nombre de transistors actifs simultanément [19].

1.3 Conception de systèmes embarqués

L'implantation d'une application sur une plateforme matérielle embarquée s'effectue en plusieurs étapes. Un flot simplifié est présenté sur la figure 1.4. La première étape consiste en une description comportementale de l'application dans un langage de haut niveau (C/C++ ou MATLAB par exemple) exécutable sur un ordinateur. Il s'agit dans un premier temps de définir le comportement et les fonctions du système. La deuxième étape consiste à identifier des tâches exhibant du parallélisme à partir de la description comportementale. La troisième étape consiste à déterminer quelles sont les tâches qui demandent le plus de puissance de calcul, dont l'exécution sur un processeur généraliste n'est pas adaptée et pour lesquelles il faut envisager une exécution sur du matériel spécialisé. Ensuite la définition du matériel et du logiciel, et jusqu'à la vérification, sont réalisées conjointement.

L'objectif de cette section est de détailler ces différentes étapes, qui mènent d'une description comportementale d'une application à la synthèse d'un composant matériel dédié à une tâche critique. Nous nous intéressons en priorité aux trois premières étapes du flot de conception d'un Soc, jusqu'à la conception conjointe logicielle-matérielle, afin de mettre en évidence l'importance des choix réalisés lors de ces étapes lorsqu'on cherche à optimiser les performances globales de la plateforme matérielle. Pour finir, cette section présente les difficultés liées à la conception d'accélérateurs matériels spécialisés pour une tâche critique.

1.3.1 Description comportementale de l'application

La conception d'une architecture commence par une étape de description du comportement de l'application. L'objectif est d'obtenir un prototype « logiciel » de l'application. Ce prototype permet de valider l'aspect fonctionnel de l'application.

Parmi les langages les plus fréquemment utilisés pour la description comportementale, on trouve notamment C/C++, SystemC [20], et MATLAB Simulink [21]. Les langages C/C++ do-

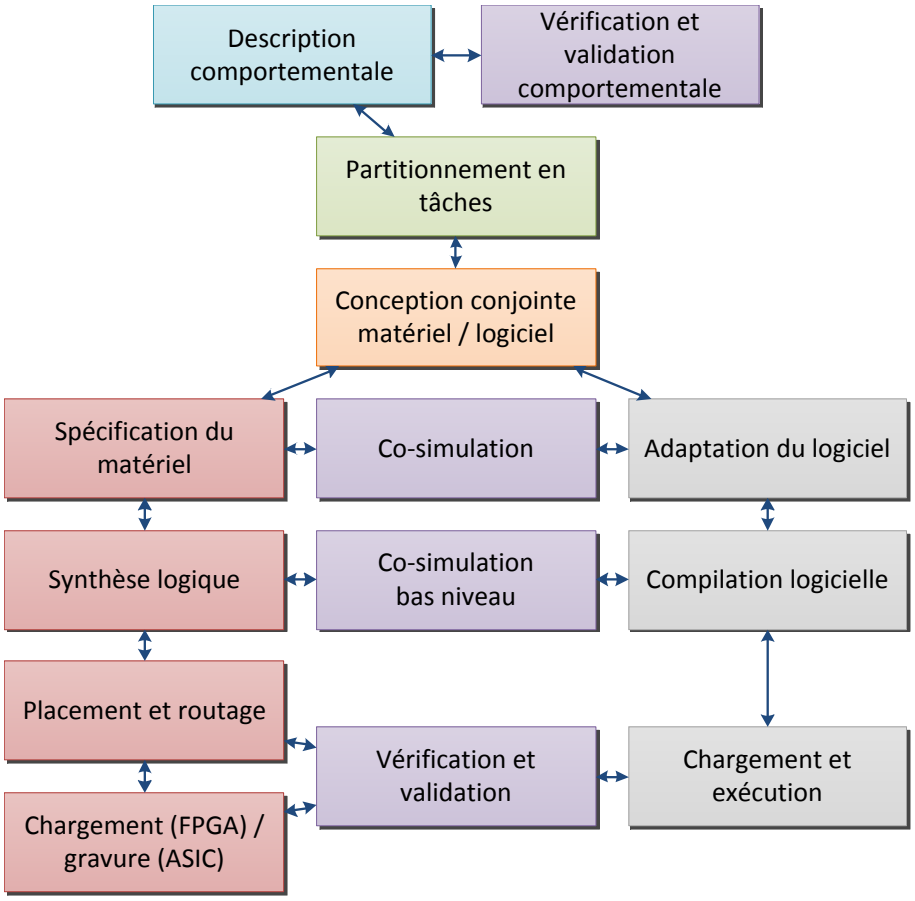


FIGURE 1.4 – Vue générale du flot de conception pour un Soc.

minent le secteur parce que de nombreux outils permettent de développer, analyser et transformer des programmes exprimés dans ces langages.

Afin de se rapprocher d'une description matérielle de l'application, SystemC permet de décrire le système sous forme d'un ensemble de blocs fonctionnels communiquants par des canaux. Chaque bloc fonctionnel et chaque canal est exprimé en C/C++, et les interactions entre les blocs via les canaux sont modélisées grâce à une surcouche de classes C++.

De la même manière, l'environnement Simulink intégré à MATLAB [21] permet de modéliser des blocs fonctionnels, et de les faire communiquer via des canaux. L'intérêt de cet environnement vient de l'utilisation d'une représentation visuelle du système modélisé. De plus, le comportement de chaque bloc fonctionnel peut être décrit dans plusieurs langages, y compris en C/C++, et autorise l'utilisation de toutes les fonctions mathématiques disponibles en MATLAB. Enfin, Simulink est fourni avec un grand nombre de modules facilitant le prototypage des applications.

Pour mettre en œuvre une application embarquée sur un Soc, il faut identifier les différents composants logiciels et les isoler dans des blocs fonctionnels, ce qu'on appelle le partitionnement de l'application. Afin de satisfaire les critères de performances tout en simplifiant la traduction

de la description comportementale vers une description structurelle, un bon partitionnement doit identifier les tâches indépendantes. Pour une application décrite dans un langage procédural tel que C, cela revient à isoler des procédures ou des structures de contrôle qui peuvent effectuer des calculs indépendants.

Dans des langages de programmation autres que C, ce partitionnement peut se faire de manière plus évidente. Le langage C++ permet par exemple d'utiliser la notion d'objet, et une tâche pourra s'exprimer par une classe. Dans un modèle de calcul *dataflow* (flot de données), le partitionnement en tâches est déjà exprimé dans le langage [22, 23]. C'est le cas des langages SystemC ou MATLAB Simulink. Cependant programmer dans des langages exploitant ce modèle de calcul est généralement considéré comme plus difficile.

1.3.2 Conception conjointe logicielle-matérielle

Comme cela a été présenté dans la section précédente, la plateforme matérielle d'un système embarqué doit fournir une vitesse d'exécution élevée pour un faible niveau de consommation énergétique. Dans le cas du décodage d'un flux vidéo HD, les solutions logicielles nécessitent des processeurs très puissants, similaires à ceux que l'on trouve dans les ordinateurs personnels. Une telle solution conduit à une consommation énergétique déraisonnable, et les concepteurs utilisent plutôt des architectures hétérogènes, où des accélérateurs matériels dédiés prennent en charge les tâches critiques.

L'objectif de la conception conjointe logicielle-matérielle [24, 25] est de déterminer, parmi l'ensemble des tâches identifiées lors du partitionnement, quelles sont les tâches critiques, pour lesquelles dédier un accélérateur matériel est nécessaire. On parle alors d'allocation de ressources, et de placement de tâches sur ces ressources (cf. figure 1.5).

L'identification des tâches critiques est délicate : il faut estimer la complexité de chaque tâche, mais aussi prendre en compte les communications éventuelles entre l'accélérateur matériel et le logiciel afin d'optimiser les performances du système, en limitant au maximum l'utilisation d'un bus de communication commun par exemple.

Pour identifier automatiquement les tâches critiques au sein d'une application, on peut par exemple instrumenter l'application et observer son comportement à l'exécution. Parallèlement, on peut estimer le WCET (*Worst Case Execution Time* : Temps d'Exécution dans le Pire Cas) pour s'assurer que certaines contraintes de temps réel sont respectées sur une architecture donnée [26].

1.3.3 Conception d'accélérateurs matériels spécialisés

Une fois les tâches critiques identifiées et placées sur des ressources matérielles spécialisées, il faut ensuite concevoir ce matériel spécialisé (cf. figure 1.6).

Dans un premier temps, la conception du matériel pour une tâche critique consiste en une traduction de la description comportementale de cette tâche en une description structurelle du matériel qui l'exécute. Cette description structurelle s'exprime dans des langages HDL (*Hardware Description Language* : Langage de Description Matérielle), tels que VHDL (*Very-high-speed integrated circuits Hardware Description Language*) ou Verilog. Ces langages permettent aux concepteurs de travailler à un niveau d'abstraction dans lequel il est possible de décrire un

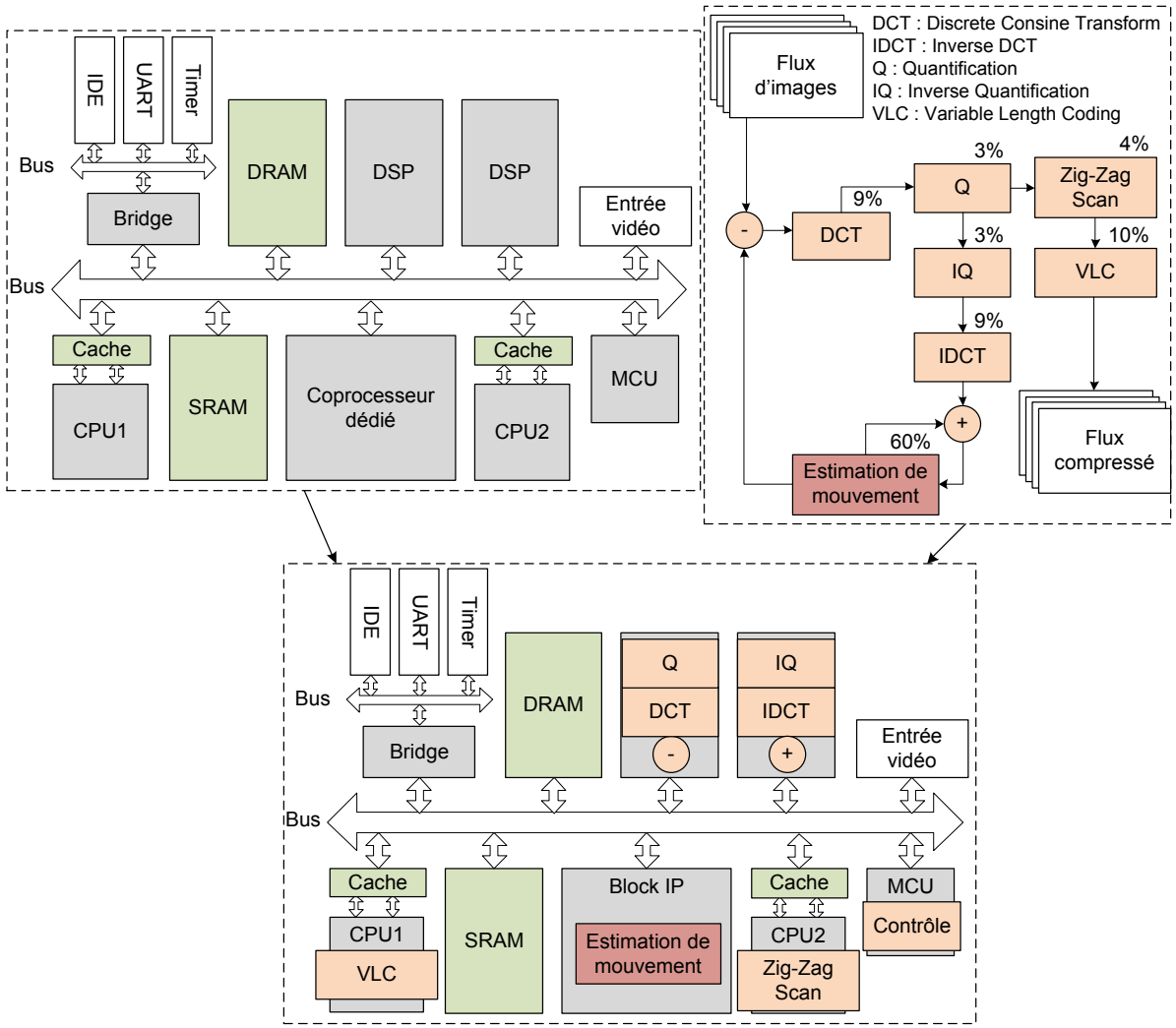


FIGURE 1.5 – Représentation conceptuelle d'un Soc (en haut à gauche), et d'une application d'encodage MPEG (en haut à droite) partitionnée en tâches, et pour laquelle chaque tâche a été profilée. L'objectif de la conception conjointe est de déterminer quelle ressource exécute quelle tâche, en tenant compte des ressources disponibles, de leurs performances, des besoins en ressources de calculs de chaque tâche et des communications.

circuit électronique au niveau RTL (*Register Transfert Level* : Niveau de Transferts de signaux entre des Registres), c'est-à-dire en termes d'opérateurs et de mémoires.

Une fois l'architecture décrite dans un langage de description matérielle, la seconde étape consiste à traduire cette description structurelle en une implémentation en termes de primitives technologiques (par exemple des LUTs[3] pour les FPGA ou des portes logiques pour les Asic). Dans cette implémentation il faut prendre en compte le câblage entre les différents éléments du circuit, et traduire les opérateurs en transistors ou portes programmables, suivant la technologie

3. Look Up Tables, l'élément de logique de base des FPGA.

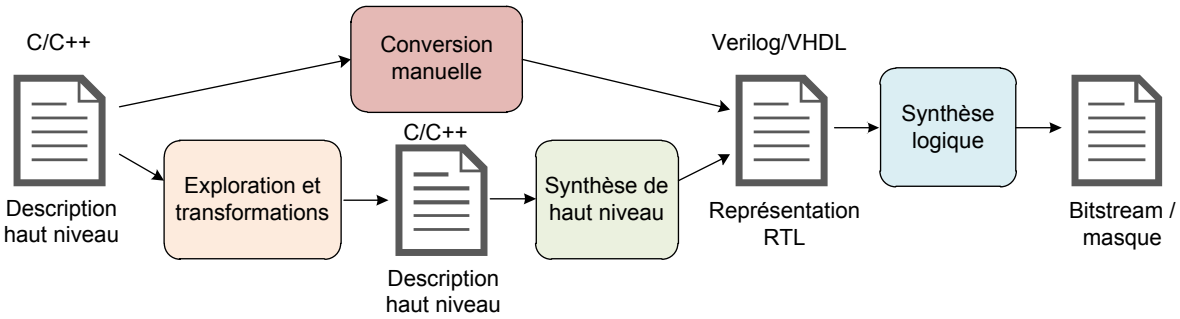


FIGURE 1.6 – Flot de conception pour la synthèse de matériel à partir d'une description dans un langage de haut niveau.

utilisée.

Effectuée par des concepteurs experts, l'étape de traduction vers une description matérielle est très coûteuse (temps ou ressources), et plusieurs cycles de traduction ou vérification sont souvent nécessaires avant d'obtenir une description matérielle efficace, et sans erreurs. Cependant la taille et la complexité des applications embarquées ne cessent de croître, et malgré l'expérience des concepteurs, les temps de conception s'allongent en conséquence. Cela pose un problème en termes de temps de mise sur le marché, mais aussi en termes de coûts de conception.

Pour accélérer ce processus de traduction et en améliorer la productivité, les concepteurs s'orientent aujourd'hui vers des outils qui traduisent automatiquement une description comportementale en une description structurelle directement utilisable par un outil de synthèse logique. Ces outils de HLS (*High-Level Synthesis* : Synthèse de Haut Niveau) sont développés depuis les années 1990 [27], et font l'objet du chapitre 2. En automatisant une grande partie du processus de synthèse, ils permettent d'obtenir des gains en productivité d'un facteur allant de 5 à 10 [28].

1.4 Conclusion

Les accélérateurs matériels spécialisés sont essentiels pour obtenir les seuils de performance requis. Cependant leur flot de conception est très long, en particulier l'étape de traduction de la description comportementale des tâches critiques vers une description structurelle. Des outils sont nécessaires pour faire face à la taille croissante des applications tout en conservant des délais de mise sur le marché acceptables. La synthèse d'accélérateurs matériels spécialisés pour systèmes embarqués est par conséquent une étape clé dans le processus de conception.

Chapitre 2

Synthèse de haut niveau d'accélérateurs spécialisés

2.1 Introduction

Ainsi qu'on l'a vu dans le chapitre 1, la croissance de la capacité des processus d'intégration des semi-conducteurs conduit les concepteurs de Soc (*System on Chip* : Système sur Puce) à spécialiser de plus en plus les plateformes matérielles. Cependant l'augmentation du nombre de transistors et de la complexité des applications a pour conséquence une explosion des coûts de conception. Pour rester compétitives, les entreprises doivent être capables de maintenir des temps de mise sur le marché les plus faibles possible et ont par conséquent besoin d'outils améliorant la productivité lors du processus de conception d'accélérateurs matériels spécialisés.

Jusque dans les années 1960, les circuits étaient conçus au niveau transistor manuellement [29]. À partir des années 1970 sont apparus les premiers outils de simulation au niveau porte logique, avant les outils de schématique dans les années 1980. Les langages de description matérielle, tels que Verilog (1986) ou VHDL (*Very-high-speed integrated circuits Hardware Description Language*) (1987), se sont répandus dans les années 1990. Ces langages ont permis aux concepteurs de spécifier des architectures au niveau RTL (*Register Transfert Level* : Niveau de Transferts de signaux entre des Registres), pour lesquels des outils de simulation, d'analyse et de synthèse sont disponibles.

Aujourd'hui, les concepteurs s'orientent vers des outils opérant à un niveau d'abstraction plus élevé, directement dans les langages de description comportementale (C/C++ par exemple) et appelés outils de HLS (*High-Level Synthesis* : Synthèse de Haut Niveau). Ces outils de HLS sont développés depuis les années 1990 et sont aujourd'hui utilisés en production par les entreprises [28]. En synthétisant des circuits directement à partir de leur description comportementale, les outils de HLS permettent ainsi d'obtenir des gains en productivité d'un facteur allant de 5 à 10. En outre, ils permettent un prototypage rapide, facilitent l'exploration de l'espace de conception et rendent la conception d'accélérateurs spécialisés accessible aux non experts.

Ce chapitre présente ce qu'est la synthèse de haut niveau. La section 2.2 décrit un flot « classique » de synthèse de haut niveau, permettant la génération d'une description structurelle

d'un accélérateur spécialisé à partir d'une description comportementale. La section 2.3 détaille certaines des optimisations utilisées pour obtenir des circuits plus performants. Ces explications mettent en évidence l'importance de la structure du code source pour obtenir un circuit performant, et n'ont pas vocation à constituer l'état de l'art du domaine. Le lecteur est invité à lire les références pour plus de détails [28].

2.2 Synthèse d'accélérateur matériel à partir d'une description comportementale

La figure 2.1 présente les principales étapes de la synthèse de haut niveau d'un circuit. Le flot commence par extraire une représentation intermédiaire à partir de la description comportementale, dans un langage de haut niveau (C/C++). Ensuite, en fonction de la technologie ciblée et des contraintes de conception, il faut ordonnancer les opérations, allouer des ressources et y placer les opérations. On obtient alors une description structurelle de l'architecture, proche du niveau RTL. Enfin, il faut générer l'architecture. Les différentes étapes de ce flot sont présentées dans les sous-sections suivantes et les transformations optimisantes dans la section 2.3.

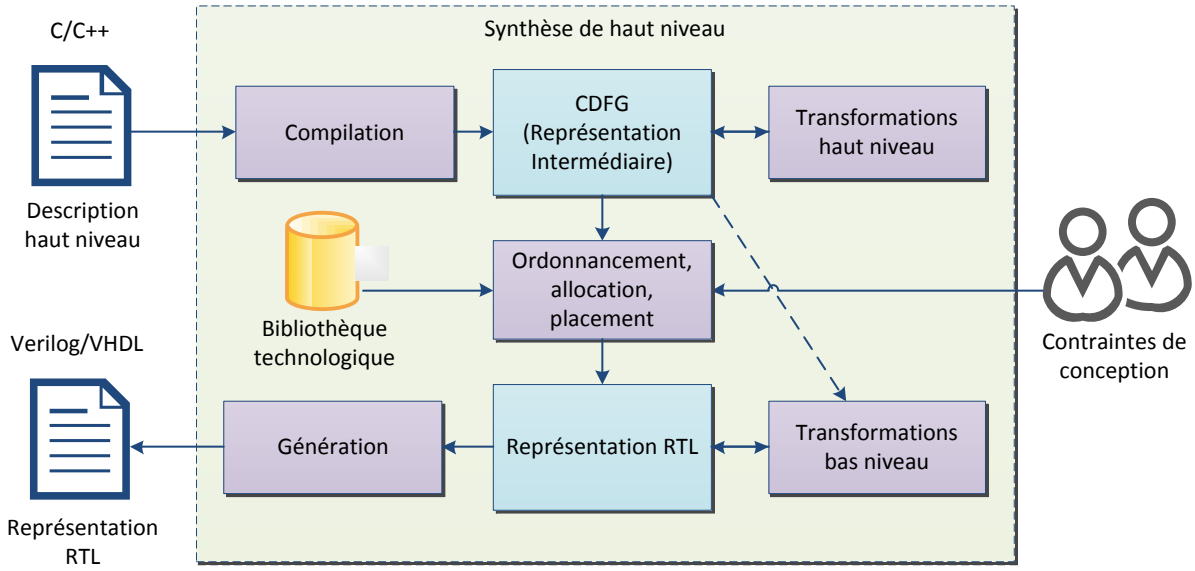


FIGURE 2.1 – Flot de synthèse de haut niveau.

2.2.1 Compilation de la description comportementale

La compilation de la description comportementale permet d'obtenir une représentation intermédiaire de la tâche à synthétiser, généralement sous la forme d'un CDFG (*Control DataFlow Graph* : Graphe de Contrôle et de Flot de Données) [30]. Cette représentation permet de modéliser le contrôle entre les blocs de base, et les dépendances de données à l'intérieur des blocs de base [1].

1. Un bloc de base représente un ensemble d'opérations ne contenant aucun saut, sauf la dernière, et n'étant cible d'aucun saut, sauf la première.

Au sein d'un bloc de base, les calculs sont généralement représentés sous la forme d'un graphe de flot de données, par exemple un Dag (*Directed Acyclic Graph* : Graphe Acyclique Orienté). Il est alors possible de transformer ces représentations grâce à des optimisations (propagation de constante, élimination de code mort, etc.) utilisées dans les compilateurs. Par exemple, la figure 2.2 décrit une boucle compilé en un CDFG, puis déroulée par un facteur 4.

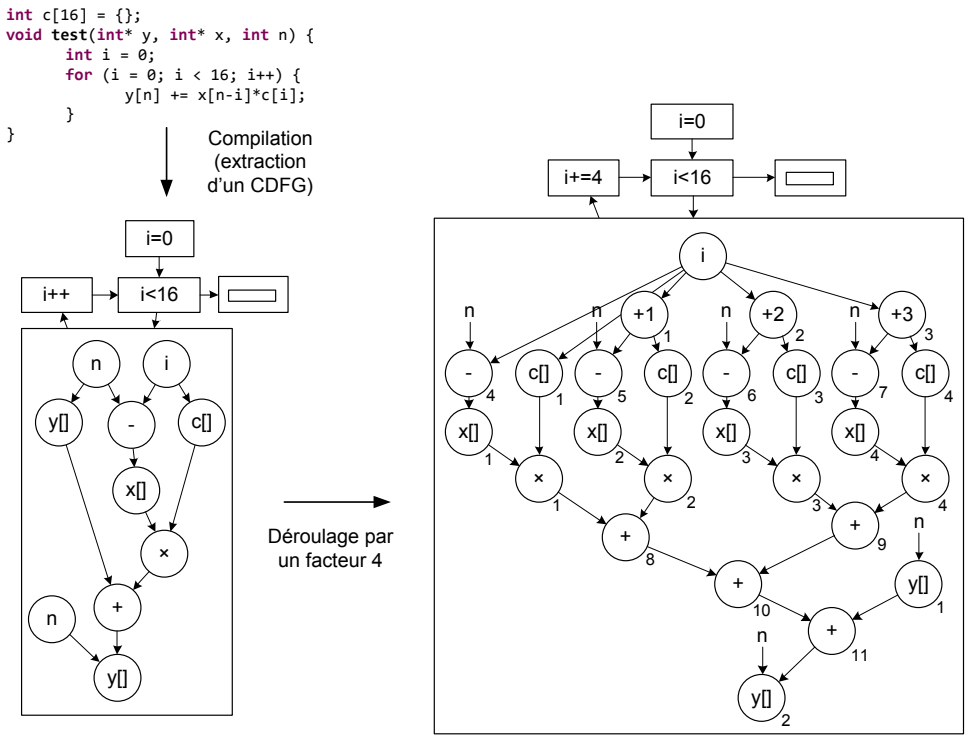


FIGURE 2.2 – Exemple de nid de boucle (en haut à gauche), compilé en un CDFG (en bas à gauche), qui est ensuite déroulé par un facteur 4 (en bas à droite).

2.2.2 Ordonnancement, allocation et placement

À partir de la représentation intermédiaire, il faut ordonnancer les opérations, allouer des ressources de calcul, placer les opérations sur ces ressources et stocker les valeurs dans des registres ou mémoires. Idéalement, ces étapes doivent être effectuées simultanément de manière à optimiser conjointement les performances et la surface. En pratique elles sont réalisées séquentiellement de manière à réduire la complexité du flot de synthèse. L'objectif est de minimiser les ressources matérielles tout en respectant les contraintes de conception. Cette étape implique de résoudre plusieurs problèmes dont la complexité est exponentielle. Dans le contexte de la HLS, cette étape est cruciale, car elle détermine la qualité de l'architecture générée, et on peut accepter que le résultat ne soit pas immédiat. On peut dès lors se permettre d'explorer l'espace des solutions possibles, avec pour objectif de trouver le meilleur compromis (cf. figure 2.3). Ces étapes sont très proches de la compilation pour processeurs VLIW (*Very Long Instruction Word* : Mot d'Instruction Très Long), à la différence près que l'architecture n'est pas figée.

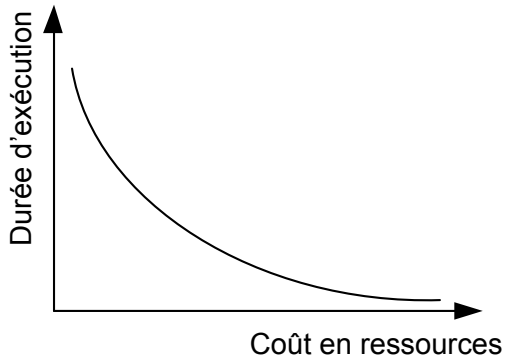


FIGURE 2.3 – Lors de l'étape d'ordonnancement, allocation et placement, il faut maximiser les performances et minimiser les ressources, tout en respectant certaines contraintes de conception. Le rapport n'est pas linéaire et suit une courbe de PARETO.

Ordonnancement L'ordonnancement consiste à déterminer un ordre d'exécution pour les opérations, tout en respectant des contraintes de latence et de ressources de calcul. Plusieurs méthodes ont été mises au point pour déterminer des ordonnancements qui minimisent la latence dans le cas de ressources illimitées (*ASAP & ALAP* [31, 32]), qui minimisent la latence sous contraintes de ressources (*List scheduling* [33], *Force-Directed scheduling* [34]), ou encore qui se basent sur la programmation par contrainte pour minimiser les ressources et la latence conjointement [35]. À titre d'exemple, la figure 2.4 présente un ordonnancement pour le CDFG déroulé de la figure 2.2.

Lorsque le CDFG comporte plusieurs blocs de base, il est possible de réduire le coût total en ressources matérielles nécessaires à leur mise en œuvre en fusionnant les Dag qui les représentent [16, 36, 37]. Le problème consiste à trouver un Dag fusionné capable d'exécuter plusieurs Dag plus petits, en utilisant des multiplexeurs et des signaux de contrôle. Le problème est résolu en trouvant la clique de poids maximal dans un graphe de compatibilité. Par exemple, le Dag de droite de la figure 2.5 peut exécuter les deux autres Dag.

Allocation et placement L'allocation et le placement consistent à affecter des ressources (de calcul ou de stockage) aux opérations et aux valeurs. En fonction de la technologie ciblée, des opérateurs sont sélectionnés dans la bibliothèque et chaque opération du CDFG ordonnancé doit être placée sur un opérateur capable de la réaliser. L'objectif est alors de maximiser la réutilisation des opérateurs entre chaque cycle, de manière à réduire le coût en ressources matérielles, tout en minimisant la connectivité (en termes de fils) et le stockage (en termes de registres). Quelques travaux résolvent le problème grâce à la programmation linéaire en nombres entiers [38]. Des travaux plus récents modélisent le problème comme une maximisation du nombre de cliques dans un graphe de compatibilité [31, 39]. Par exemple, l'architecture de la figure 2.6 peut exécuter le CDFG ordonnancé de la figure 2.4.

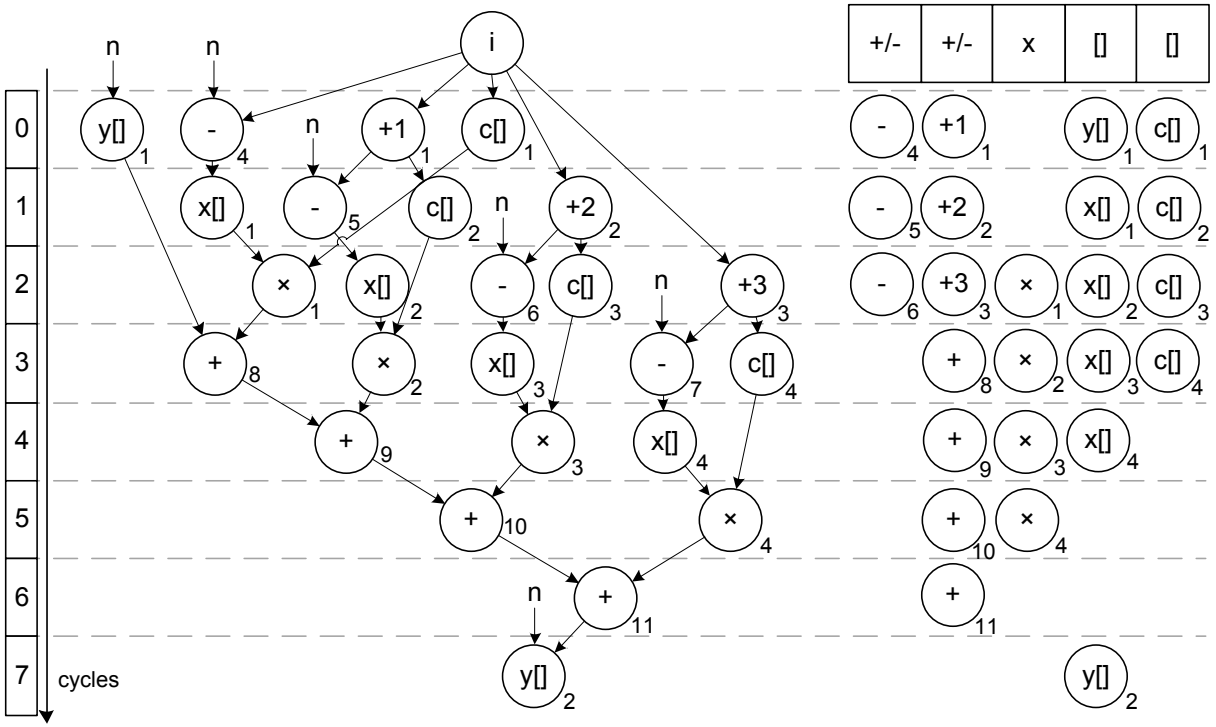


FIGURE 2.4 – Ordonnancement possible pour le CDFG déroulé de la figure 2.2 pour des ressources matérielles constituées de 2 additionneur/soustracteur, 1 multiplicateur et 2 accès au bus par cycle (à gauche). Une table d'allocation, pour cet ordonnancement et les ressources données, est illustrée à droite.

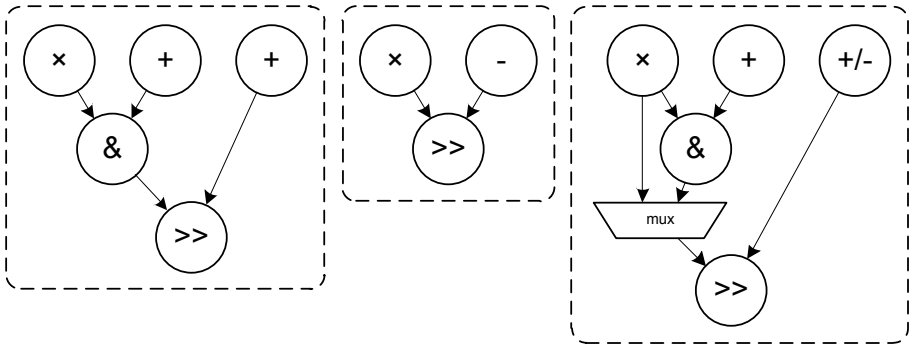


FIGURE 2.5 – Exemple de fusion de chemins de données. Le Dag de droite peut exécuter les deux autres Dag. Cette fusion permet d'économiser un additionneur et un multiplicateur, au prix d'un multiplexeur.

2.2.3 Génération de l'architecture

Une fois les ressources sélectionnées, les opérations ordonnancées et le placement effectué, il faut générer une architecture au niveau RTL (dans les langages VHDL ou Verilog par exemple) qui exécute l'application. Cette architecture se décompose en deux unités : l'unité de contrôle, sous

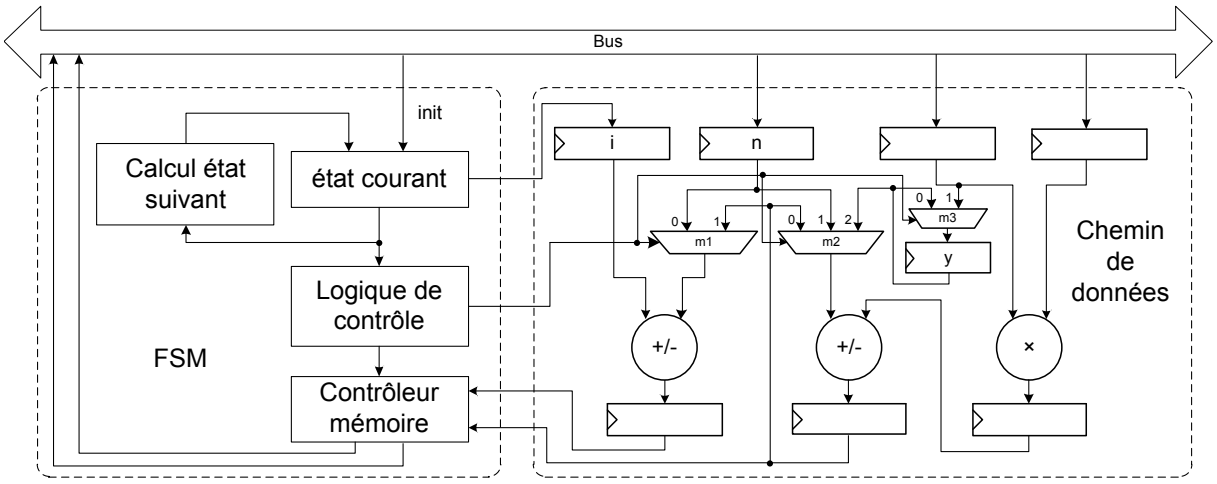


FIGURE 2.6 – Exemple d'architecture capable d'exécuter le nid de boucles de la figure 2.4.

la forme d'une machine à états, et l'unité de traitement, sous la forme d'un chemin de données.

Le rôle de la machine à états est de contrôler :

- l'exécution en déterminant à chaque cycle l'état d'avancement de l'ordonnancement calculé précédemment ;
- le chemin de données via des signaux de contrôle.

Le chemin de données contient quant à lui tous les opérateurs alloués ainsi que les mémoires locales, nécessaires pour exécuter les opérations et stocker les résultats.

2.3 Optimisations pour une synthèse d'accélérateurs performants

Les techniques de base présentées dans la section précédente permettent la synthèse d'un accélérateur matériel dédié à partir d'une description comportementale. Cependant, étant donné que le point de départ de cette synthèse est une procédure dans un langage de haut niveau, de nombreuses optimisations classiques dans les compilateurs logiciels sont applicables [40]. Cette section présente plusieurs optimisations qui interviennent à différents niveaux dans le processus de synthèse, avec différents objectifs de performance.

2.3.1 Réduction de la complexité des calculs

Comme dans un compilateur, de nombreuses optimisations visant à réduire la complexité, en termes de quantité de calculs, sont applicables en HLS. L'objectif est d'enlever, simplifier ou déplacer des opérations pour améliorer les performances, en termes de temps de calculs et de ressources matérielles nécessaires. Parmi les transformations les plus connues peuvent être citées les suivantes, illustrées par la figure 2.7.

- L'évaluation de constantes permet d'évaluer des expressions dont les valeurs sont connues statiquement, c'est-à-dire lors de la compilation, pour éviter de le faire pendant l'exécution. Cela permet d'économiser du temps et des opérateurs.

- La propagation de constante consiste à remplacer un référencement de variable par sa valeur, s'il est possible de la déterminer à la compilation.
- L'élimination de code mort, associé avec les transformations précédentes, évalue les conditions et bornes des structures de contrôle, et supprime les parties du code qui ne sont pas atteignables.
- La recherche d'expressions communes permet d'évaluer une seule fois une expression qui intervient dans plusieurs calculs.

Afin de réduire la complexité des calculs, et par la même occasion le coût des opérateurs impliqués, des techniques permettent de transformer des opérateurs coûteux en une séquence d'opérateurs plus simples. Par exemple, une multiplication par une constante peut être remplacée par des décalages associés à des additions, comme le montre la figure 2.8. Cette transformation de réduction de force est applicable à plusieurs autres opérateurs [41, 42].

Dans le cas des structures itératives, deux transformations sont bien connues, et s'appliquent aussi bien en matériel qu'en logiciel : le déplacement de code invariant et la réduction de force. Le déplacement de code invariant consiste à déplacer en dehors de la boucle tous les calculs dont le résultat ne varie pas pendant l'exécution de la boucle. La figure 2.9 montre un exemple de cette transformation.

La réduction de force s'applique particulièrement bien quand il s'agit de remplacer les opérations dépendantes des indices de boucles par des expressions plus simples [43]. Cette transformation prend tout son sens dans le cas des calculs d'adresse lors d'accès à des tableaux, comme le montre la figure 2.10, mais aussi lorsque le contrôle dépend d'expressions complexes des indices, comme c'est le cas lors de la génération de code dans le modèle polyédrique, présentée dans le chapitre 4. Toutes ces transformations sont effectuées de manière automatique dans les compilateurs.

2.3.2 Transformation des structures de données, des ressources de stockage et des accès à la mémoire

Dans les applications logicielles ou matérielles, les accès mémoire sont souvent le facteur qui limite la vitesse d'exécution, en particulier lors des accès répétitifs dans les structures itératives. En effet on constate que les calculs sont réalisés bien plus rapidement que ne l'est l'approvisionnement en données [44]. Des transformations des structures de données, des ressources de stockage ainsi que le réordonnancement des accès à ces ressources permettent d'en limiter l'impact.

Plusieurs solutions sont possibles pour mettre en œuvre un tableau en matériel. Tout d'abord, la « scalarisation » des tableaux, qui consiste à transformer un tableau en un ensemble de registres, permet un accès parallèle à l'ensemble des cellules du tableau. Le coût en ressources matérielles est prohibitif dans le cas de grands tableaux, et il faut alors utiliser des mémoires de type RAM (*Random-Access Memory* : Mémoire à Accès Aléatoire), implantées sur la puce. Elles permettent l'accès à quelques cellules en parallèle. La mise en œuvre la plus répandue est la *dual port memory* (mémoire à deux ports) qui permet deux accès en parallèle. Une alternative entre les registres et la mémoire unique consiste à partitionner le tableau en plusieurs bancs mémoire, de manière à pouvoir accéder à autant de cellules par cycles qu'il y a de bancs mémoire [45].

```
//original
int func(int a) {
  int b = 3, c = 5, ret;
  if (b > 4)
    ret = a+b*c;
  else {
    if (a < b)
      ret = b*a + c*a;
    else
      ret = b*c + c*a;
  }
  return ret;
}
```

```
// propagation de constantes
int func(int a) {
  int b = 3, c = 5, ret;
  if (3 > 4)
    ret = a+3*5;
  else {
    if (a < 3)
      ret = 3*a + 5*a;
    else
      ret = 3*5 + 5*a;
  }
  return ret;
}
```

```
// évaluation de constantes
int func(int a) {
  int b = 3, c = 5, ret;
  if (0)
    ret = a+15;
  else {
    if (a < 3)
      ret = 3*a + 5*a;
    else
      ret = 15 + 5*a;
  }
  return ret;
}
```

```
// élimination de code mort
int func(int a) {
  int b = 3, c = 5, ret;
  if (a < 3)
    ret = 3*a + 5*a;
  else
    ret = 15 + 5*a;
  return ret;
}
```

```
// recherche d'expressions
// communes
int func(int a) {
  int b = 3, c = 5, ret;
  int expr = 5*a;
  if (a < 3)
    ret = 3*a + expr;
  else
    ret = 15 + expr;
  return ret;
}
```

FIGURE 2.7 – Exemple d'application des transformations permettant de réduire la complexité des calculs.

Lorsqu'il faut accéder à des mémoires de plus grande taille, généralement externes au composant (*off chip*), il faut prendre en compte la spécificité des technologies mémoires utilisées. En effet, les mémoires externes les plus répandues sont les mémoires de type Sdram (*Synchronous Dynamic Random-Access Memory* : Mémoire Dynamique Synchrone à Accès Aléatoire). Ces mémoires offrent une densité élevée en termes de quantité de données par transistor. Cependant il faut accéder aux données de manière consécutive pour obtenir des débits maximum et une

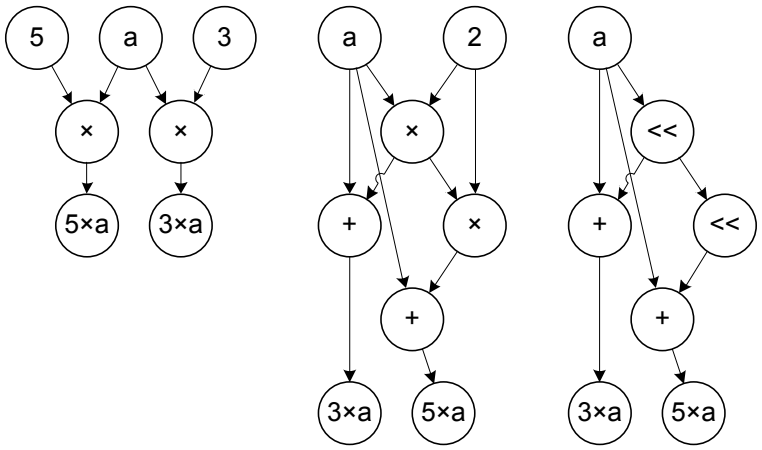


FIGURE 2.8 – Les multiplications par des constantes peuvent être remplacées par des décalages et des additions, beaucoup plus rapides et moins chères que les multiplicateurs à implémenter en matériel.

```
void func2(int tab[256], int N) {
  int i;
  for (i = 0; i < 256; i++)
    if (tab[i] == 0)
      tab[i] = N*N;
}
```

```
void func2(int tab[256], int N) {
  int i, invr = N*N;
  for (i = 0; i < 256; i++)
    if (tab[i] == 0)
      tab[i] = invr;
}
```

FIGURE 2.9 – Exemple de déplacement de code invariant. Au lieu de répéter le calcul de $N \times N$ à chaque itération de la boucle, la valeur est calculée une fois pour toute avant la boucle.

```
void func2(int tab[256], int N) {
  int i, invr = N*N;
  long addr = (long) tab;
  for (i = 0; i < 256; i++)
    if ((*(int*)(addr+(i*4))) == 0)
      (*(int*)(addr+(i*4))) = invr;
}
```

```
void func2(int tab[256], int N) {
  int i, invr = N*N;
  long addr = (long) tab;
  for (i = 0; i < 256; i++)
    addr+=4;
    if ((*(int*)(addr)) == 0)
      (*(int*)(addr)) = invr;
}
```

FIGURE 2.10 – Les accès tableau de l'exemple de gauche de la figure 2.9 peuvent s'exprimer par des opérations sur des adresses (à gauche), en supposant que les entiers sont codés sur 4 octets. Au lieu d'avoir une multiplication de l'indice i par 4 à chaque itération de la boucle, on peut utiliser un compteur qui incrémente l'adresse de 4 à chaque itération de la boucle (à droite).

latence minimum.

Pour optimiser l'accès aux mémoires de type SDRAM, les composants peuvent embarquer des contrôleurs mémoires, qui se chargent de réordonner les accès de manière consécutive, dans la limite de leur capacité [46, 47]. Une autre possibilité consiste à mettre en place une hiérarchie

mémoire à l'aide de caches ou de mémoires *scratchpad*.

Une autre approche consiste à essayer d'exploiter la localité spatiale, c'est-à-dire accéder consécutivement à des données qui sont à des adresses proches dans la mémoire ; mais aussi optimiser la localité temporelle, c'est-à-dire minimiser le temps entre deux accès consécutifs à une même donnée, afin de profiter de sa présence dans le cache. Ces optimisations peuvent s'effectuer directement au niveau de la description algorithmique de l'application dans le langage de haut niveau.

Une technique bien connue consiste à appliquer des transformations de pavage (*tiling*) sur les nids de boucles. Comme illustré par la figure 2.11, l'objectif est de faire en sorte que les boucles plus internes n'impliquent pas (ou peu) d'éviction de données, de manière à exploiter au mieux le cache.

```
void prodmat(int** A, int** B, int** C) {
  int i,j,k;
  for (i=0; i<256; i++)
    for (k=0; k<256; k++)
      for (j=0; j<256;j++)
        C[i][j] += A[i][k]*B[k][j];
}
```

(a) Exemple de produit matriciel.

```
void prodmat_tiled(int** A, int** B, int** C) {
  int i,j,k,jj,kk;
  for (kk=0; kk<256; kk+=8)
    for (jj=0; jj<256;jj+=8)
      for (i=0; i<256; i++)
        for (k=kk; k<kk+8; k++)
          for (j=jj; j<jj+8;j++)
            C[i][j] += A[i][k]*B[k][j];
}
```

(b) Produit matriciel de la figure (a) pour lequel la transformation de pavage est appliquée sur les deux boucles les plus internes.

FIGURE 2.11 – Exemple d'application du pavage sur le produit matriciel de la figure (a).

Une dernière optimisation possible, bien adaptée à la synthèse de matériel, consiste à jouer sur le choix de l'encodage des données. Par exemple le compteur de la boucle de l'exemple de la figure 2.10 ne requiert que 9 bits pour stocker toutes les valeurs entre 0 et 256 (inclus), au lieu des 32 bits utilisés généralement pour le type `int`. De la même manière, les valeurs de type à virgule flottante (*float* et *double*) peuvent être codées en utilisant des encodages non standards, plus petits, pour réduire la taille des mémoires. De plus, pour simplifier les opérateurs, les valeurs réelles peuvent être encodées dans des représentations en virgule fixe au lieu des représentations en virgule flottante [48].

2.3.3 Exploitation du parallélisme

Le dernier axe d'optimisation pour obtenir un circuit performant consiste à rechercher du parallélisme, et à l'implémenter en matériel. Cette recherche de parallélisme ainsi que sa mise en œuvre sont importantes pour respecter les critères de vitesse d'exécution tout en modérant l'impact sur les ressources matérielles.

Caractérisation du parallélisme

Dans une application informatique, le parallélisme est caractérisé par la possibilité d'exécuter plusieurs opérations simultanément. Par exemple dans le CDFG ordonnancé de la figure 2.4, il est possible d'exécuter deux additions et une multiplication simultanément.

Lorsque le parallélisme apparaît dans un même Dag, on parle alors de parallélisme au niveau opération. Lorsque plusieurs blocs fonctionnels indépendants sont exécutés en parallèle, on parle alors de parallélisme de tâche. Entre les deux, on peut trouver du parallélisme de contrôle, en particulier dans les structures itératives.

Extraction du parallélisme

Le parallélisme de tâche peut être extrait lors de la conception de l'application. Le parallélisme au niveau instruction est généralement extrait lors de l'ordonnancement du Dag, qui va se charger de trouver le meilleur compromis entre la profondeur du Dag et le nombre d'opérateurs. Il est possible d'améliorer cette extraction en travaillant sur des Dag plus grands, obtenus grâce à des transformations de code. Par exemple, la fusion de boucles permet de regrouper en un même Dag le corps de deux boucles, comme le montre la figure 2.12. Les deux instructions peuvent alors être exécutées en parallèle au sein du corps de la boucle fusionnée. Une autre transformation de boucle qui permet d'optimiser le parallélisme instruction consiste à dérouler partiellement les boucles, comme dans le cas de l'exemple de la figure 2.2.

```
void func3(int tab[256]) {
  int i, s = 0;
  for (i = 0; i < 256; i++)
    s++;
  for (i = 0; i < 256; i++)
    tab[i] = i;
}
```

```
void func3(int tab[256]) {
  int i, s = 0;
  for (i = 0; i < 256; i++) {
    s++;
    tab[i] = i;
  }
}
```

FIGURE 2.12 – Exemple de boucles (à gauche) et de leur fusion (à droite).

Recherche de parallélisme dans les boucles

Le parallélisme inter-itérations consiste à rechercher les itérations d'une boucle qui peuvent être exécutées en parallèle. L'approche est différente du parallélisme instruction : le compilateur ne se base pas seulement sur une analyse du corps de la boucle pour rechercher du parallélisme, mais cherche à prouver l'absence de dépendances entre plusieurs itérations, successives ou non, d'une boucle ou d'un nid de boucles.

Une dépendance entre deux opérations représente une relation de causalité : deux opérations accèdent à une même donnée et au moins une des deux opérations en modifie la valeur. Dans un tel cadre, si l'ordre d'exécution de deux opérations dépendantes est modifié, le résultat des opérations est lui aussi modifié. Le nouvel ordre d'exécution ne respecte pas la sémantique originale, et la transformation qui a conduit à ce nouvel ordre d'exécution est alors illégale.

Les outils de HLS disposent de méthodes qui leur permettent de déterminer les dépendances présentes au sein d'un nid de boucles. Ces méthodes sont cependant limitées, ce qui peut empêcher l'outil de déterminer précisément l'absence d'une dépendance, interdisant ainsi la mise en œuvre parallèle d'une boucle.

Parfois, le parallélisme inter-itérations n'est pas explicite, parce qu'il existe effectivement des dépendances entre plusieurs itérations d'un nid de boucles. Il est pourtant parfois possible de transformer ce nid de boucles et de mettre en évidence du parallélisme, comme c'est le cas dans l'exemple de la figure 2.13. Ces transformations ne sont pas toujours légales, pour les mêmes raisons que celles mentionnées ci-dessus. Des techniques plus avancées permettent de trouver automatiquement une transformation qui met en évidence du parallélisme (cf. chapitre 3).

```
void matvec(int mat[32][32],
      int vec[32], int res[32][32]) {
  int i,j;
  for (i = 0; i < 32; i++)
    for (j = 0; j < 32; j++)
      if (j == 0)
        res[i][j] = mat[i][j]*vec[j];
      else
        res[i][j] = res[i][j-1]
                + mat[i][j]*vec[j];
}
```

```
void matvec(int mat[32][32],
      int vec[32], int res[32][32]) {
  int i,j;
  for (j = 0; j < 32; j++)
    for (i = 0; i < 32; i++)
      if (j == 0)
        res[i][j] = mat[i][j]*vec[j];
      else
        res[i][j] = res[i][j-1]
                + mat[i][j]*vec[j];
}
```

FIGURE 2.13 – Exemple de boucle (à gauche), pour lequel la boucle interne (indice j) n'est pas parallèle, car deux itérations successives accèdent à la même case mémoire (à cause de l'accès `res[i][j-1]`). Après une permutation des deux boucles (*loop interchange*, à droite), les indices i et j sont permutés, la boucle interne ne porte plus de dépendances et est donc parallèle.

Mise en œuvre du parallélisme

La mise en œuvre du parallélisme est similaire au niveau tâche, boucle ou instruction. On retrouve ainsi deux possibilités, la vectorisation et le pipeline, illustrés par la figure 2.14. La vectorisation consiste à répliquer les unités de traitements, pour pouvoir exécuter en parallèle plusieurs calculs indépendants. Il est possible de répliquer aussi bien des opérateurs, que des chemins de données ou encore des composants complets. Cette réplication apporte un gain en vitesse d'exécution tant que le débit des accès mémoires n'est pas limitant. Dans le cadre des boucles, la vectorisation est souvent implémentée par un déroulage partiel. En effet, lorsqu'une boucle est parallèle et déroulée partiellement, les opérations du corps de la boucle déroulée peuvent être exécutées en parallèle, résultant en une exécution vectorisée.

L'inconvénient de la vectorisation est son coût en ressources, car elle implique une réplication des opérateurs. La seconde approche, le pipeline, consiste à découper une opération en plusieurs étapes, puis d'exécuter plusieurs instances de l'opération sur des données indépendantes de manière entrelacée.

La figure 2.14 illustre les différentes mises en œuvre du parallélisme sur un exemple de boucle.

Le corps de la boucle porte une dépendance entre les instructions `i1` et `i2`, sur la cellule i du tableau `x`, il est donc impossible d'exécuter `i1` et `i2` en parallèle sans transformer la boucle. L'exécution séquentielle (figure 2.14a) nécessite une ressource matérielle pour chacune des deux instructions, et permet l'exécution totale en 16 cycles. L'exécution vectorisée par un facteur 4 (figure 2.14b) nécessite 4 fois moins de cycles, mais aussi 4 fois plus de ressources. En appliquant le pipeline de boucles (figure 2.14c), on exécute la boucle en 9 cycles tout en conservant une seule ressource par instruction. Enfin l'exécution vectorisée par un facteur 2 et pipelinée (figure 2.14d) exécute la boucle en 5 cycles et nécessite 2 fois plus de ressources que la mise en œuvre séquentielle.

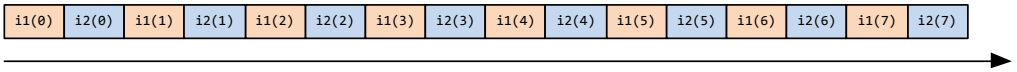
2.4 Conclusion

Malgré le gain en productivité qu'ils offrent, les outils de synthèse de haut niveau restent limités dans leur capacité à exploiter la sémantique des langages de haut niveau. En effet, certains outils tels que GAUT [49] ne supportent que les boucles constantes, qu'il est possible de mettre à plat complètement, et se concentrent sur la génération d'un chemin de données performant. Les outils commerciaux dominant le marché, tels que Catapult-C de Mentor Graphics [50], C-to-Silicon de Cadence [51], AutoESL de Xilinx [52] ou Impulse-C de Impulse Accelerated [53], et certains outils académiques, tels que LegUp [54], sont capables de synthétiser des machines à états pour contrôler des boucles dont les bornes ne sont pas connues statiquement.

Cependant, contrairement aux compilateurs, ces outils sont souvent incapables de réordonnancer les boucles. En excluant la fusion, appliquée par défaut dans Catapult-C, aucune transformation de réordonnancement des boucles n'est disponible dans ces outils. De plus, lorsque cette transformation est disponible, elle est très limitée dans son application par les faiblesses de l'analyse de dépendance mise en œuvre dans ces outils.

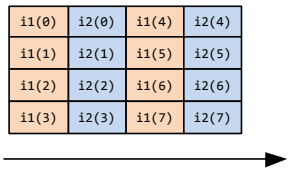
Il en va de même pour l'optimisation de la localité et la recherche de parallélisme, caractéristiques primordiales à exploiter pour obtenir des accélérateurs performants : les outils sont souvent incapables de déterminer que des itérations successives d'une boucle sont indépendantes, et interdisent sa mise en œuvre parallèle. Lorsque le concepteur détermine qu'une boucle est effectivement parallèle, il faut alors utiliser des annotations pour outrepasser les analyses de l'outil. Le tableau 2.1 récapitule les forces et faiblesses de quelques outils disponibles aujourd'hui sur le marché. Ces limitations sont la conséquence d'un effort de développement concentré sur les optimisations à bas niveau et un support très complet des technologies ciblées offerts par les outils de HLS.

```
for (i = 0; i < 8; i++) {
  x[i]   = f(in[i]); //i1(i)
  out[i] = g(x[i]);  //i2(i)
}
```



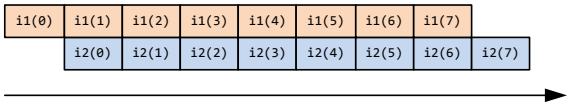
(a) Exécution séquentielle.

```
for (i = 0; i < 8; i+=4) {
  x[i]   = f(in[i]);   out[i]   = g(x[i]);
  x[i+1] = f(in[i+1]); out[i+1] = g(x[i+1]);
  x[i+2] = f(in[i+2]); out[i+2] = g(x[i+2]);
  x[i+3] = f(in[i+3]); out[i+3] = g(x[i+3]);
}
```



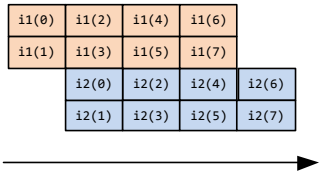
(b) Exécution vectorisée : déroulage partiel de la boucle par un facteur 4.

```
x[0] = f(in[0]);
for (i = 1; i < 8; i++) {
  x[i]     = f(in[i]);
  out[i-1] = g(x[i-1]);
}
out[7] = g(x[7]);
```



(c) Exécution pipelinée : décalage (*loop shifting*) de `i2`.

```
x[0] = f(in[0]);
x[1] = f(in[1]);
for (i = 2; i < 8; i+=2) {
  x[i]   = f(in[i]);
  x[i+1] = f(in[i+1]);
  out[i-2] = g(x[i-2]);
  out[i-1] = g(x[i-1]);
}
out[6] = g(x[6]);
out[7] = g(x[7]);
```



(d) Exécution vectorisée (facteur 2) et pipelinée.

FIGURE 2.14 – Illustration des différentes mises en œuvre du parallélisme. Chaque case représente une opération correspondant à l'exécution d'une des instructions du corps de la boucle (`i1` ou `i2`) pour une valeur de i donnée. Toutes les opérations sur une même colonne sont exécutées simultanément.

	Catapult-C	C-to-Silicon	Impulse-C	AutoESL	GAUT	LegUp
Gestion des pointeurs	++	-	-	-	-	++
Organisation de la mémoire	+++	+	-	+++	-	+
Interfaces externes	++	++	++	++	+	+
Déroulage des boucles	+++	+	+	+	+	+++
Pipeline	+++	+	+	++	-	+
Réordonnancement des boucles	+	- -	- -	+	- - -	++
Analyse de dépendance	++	+	-	+	- -	+++
Directives de compilation	++	+	+	++	-	+
Réutilisation des ressources	+++	++	-	++	++	- -

TABLE 2.1 – Comparatif qualitatif des forces et faiblesses des outils de synthèse de haut niveau du marché. Les outils ne supportent généralement qu'un sous-ensemble du langage C, parfois étendu avec des types et des annotations. En particulier, à cause du caractère statique des architectures, les mémoires doivent être dimensionnées statiquement et les pointeurs doivent être résoluble statiquement. Les transformations de boucles sont principalement limitées par les analyses de dépendances basiques et conservatives implémentées dans ces outils.

Chapitre 3

Modèle polyédrique et synthèse de haut niveau

3.1 Introduction

Les structures de contrôle itératives permettent de factoriser une quantité importante de calculs de manière compacte. Très largement utilisées dans les programmes C/C++, ces structures de boucles concentrent la majorité des calculs effectués dans les applications typiques des systèmes embarqués. Elles font par conséquent l'objet d'une attention particulière lors de la compilation.

Pour optimiser l'exécution des boucles, les compilateurs y appliquent des transformations visant à améliorer la localité (spatiale et temporelle) des accès mémoire et d'en extraire du parallélisme. Dans le contexte de la synthèse de haut niveau, l'architecture peut ainsi être adaptée au degré de parallélisme disponible dans l'application. De la même manière, la hiérarchie et la taille des mémoires peuvent être adaptées aux besoins de l'application embarquée. La problématique est alors de transformer le nid de boucles pour mettre en avant le parallélisme ou améliorer la localité des accès mémoire, de manière à obtenir un circuit spécialisé plus performant.

Ce chapitre présente comment l'utilisation d'un formalisme mathématique pour représenter les nids de boucles permet d'atteindre ces objectifs. La première section présente un flot global de compilation utilisant le modèle polyédrique, ainsi que la représentation des boucles et des transformations. Pour simplifier les explications et se concentrer sur l'aspect syntaxique des transformations, cette première section ignore volontairement la notion de légalité des transformations, qui est introduite dans la deuxième section. La troisième section présente quelques techniques pour dériver automatiquement une transformation légale améliorant le parallélisme ou la localité. La quatrième section présente les travaux qui se sont inspirés du modèle polyédrique pour proposer des techniques efficaces de synthèse de matériel spécialisé.

3.2 Flot global de transformation

Cette section présente un flot global de transformation dans le modèle polyédrique, illustré par la figure 3.1, qui s'insère comme une étape d'optimisation sur la représentation interne d'un compilateur. Le point d'entrée est une spécification sous forme de boucles d'une tâche critique. À partir de cette représentation sous forme de boucles, il faut (1) en extraire une représentation dans le modèle polyédrique, (2) la transformer, puis (3) générer un nouveau nid de boucles qui parcourt la représentation polyédrique transformée.

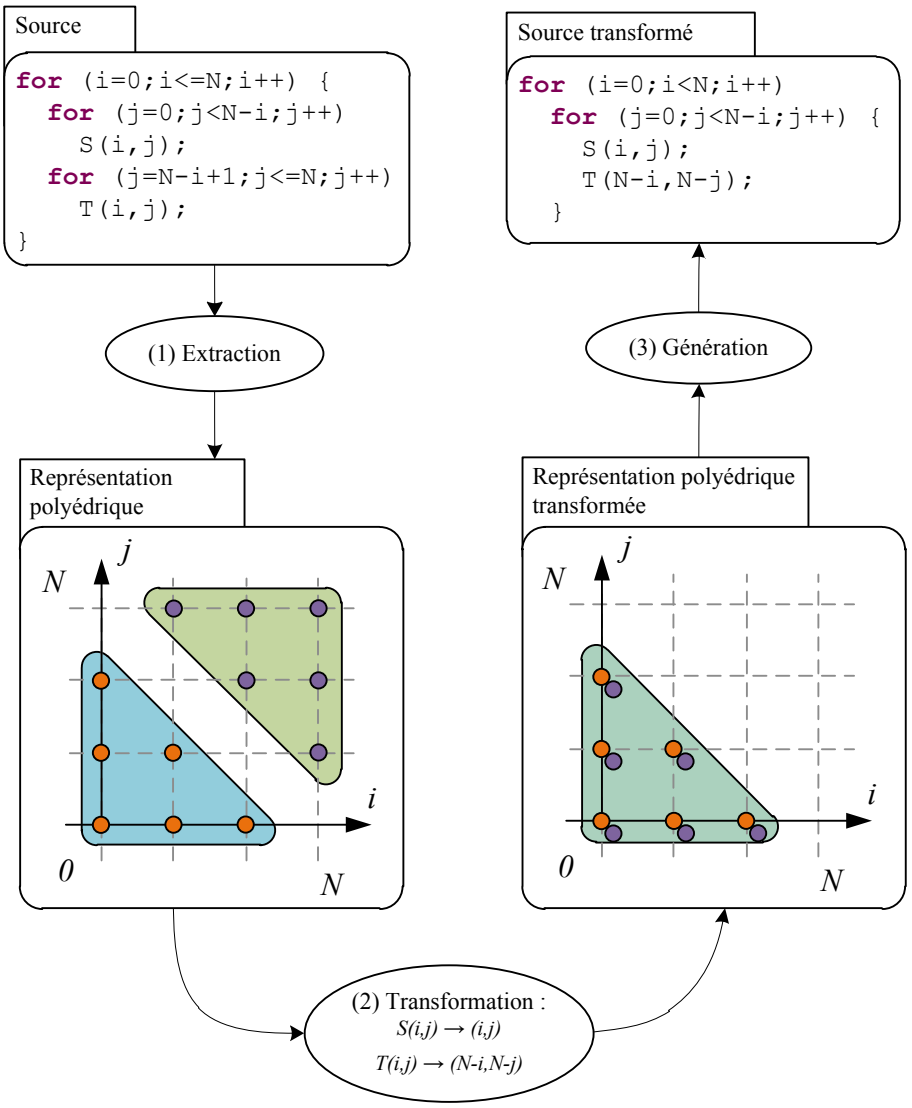


FIGURE 3.1 – Flot global de compilation dans le modèle polyédrique. La transformation présentée en exemple réalise une double inversion (sur les deux boucles englobant l'instruction T) et un double décalage de $+N$ (sur ces deux même boucles) de manière à réduire la taille du contrôle. Les détails sont présentés dans les sous-sections suivantes.

Dans ce chapitre, les programmes sont restreints aux boucles `for` dont les bornes, les gardes et les fonctions d'accès des tableaux sont des fonctions affines des itérateurs de boucles englobantes et des paramètres du nid de boucles. Cette classe de programmes est appelée Scop (*Static Control Programs* : Programmes à Contrôle Statique) [55], ou de manière équivalente ACL (*Affine Control Loops* : Boucles à Contrôle Affine). De plus, pour simplifier l'explication, cette section ne considère que des instructions abstraites, sans aucune dépendance, et se concentre sur l'aspect syntaxique des transformations dans le modèle polyédrique. La notion de dépendance est introduite dans la section suivante.

3.2.1 Représentation des programmes dans le modèle polyédrique

À chacune des structures de boucle `for` qui composent le Scop est associé un itérateur entier. Étant donnée une instruction S_i à une profondeur p, le vecteur $\vec{x} \in \mathbb{Z}^p$ composé de tous les itérateurs des boucles englobant S_i est noté *vecteur d'itération* de S_i. L'ensemble des valeurs possibles de $\vec{x}$ représente le *domaine d'itération* de S_i noté $D_{S_i}(\vec{n}) \subseteq \mathbb{Z}^p$, où $\vec{n} \in \mathbb{Z}^q$ représente les q paramètres du programme.

Étant donné que les bornes des boucles et les gardes sont des expressions affines, on peut représenter $D_{S_i}(\vec{n})$ par une union de d polyèdres $D^j_{S_i}(\vec{n}), 1 \leq j \leq d$, définis par r^j_i contraintes affines. On note alors :

$$D_{S_i}(\vec{n}) = \bigcup_{j=1}^{d} D^j_{S_i}(\vec{n}), \text{ avec } D^j_{S_i}(\vec{n}) = \{\vec{x} \mid A^j_i.\vec{x} + B^j_i.\vec{n} + \vec{c_i}^j \geq \vec{0}\} \tag{3.1}$$

où $A^j_i \in \mathbb{Z}^{r^j_i} \times \mathbb{Z}^p$ et $B^j_i \in \mathbb{Z}^{r^j_i} \times \mathbb{Z}^q$ sont des matrices constantes, et $\vec{c_i}^j \in \mathbb{Z}^{r^j_i}$ un vecteur constant. L'instance particulière de l'instruction S_i pour le vecteur d'itération $\vec{x} \in D_{S_i}(\vec{n})$ est *l'opération* $S_i(\vec{n}, \vec{x})$.

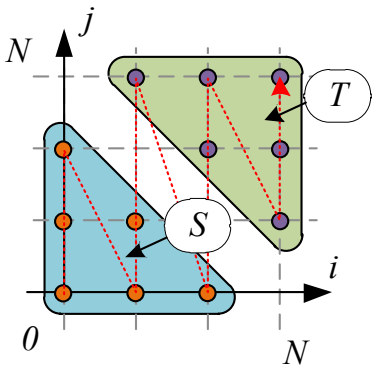
Par exemple, dans le cas du nid de boucles de la figure 3.2, les itérateurs de boucles englobant l'instruction S sont i et j, et le seul paramètre du programme est N. L'ensemble des valeurs que peuvent prendre ces itérateurs est défini par l'ensemble $D_S(N)$, dont les contraintes affines sont extraites des bornes des boucles (il en va de même pour l'instruction T et son domaine d'itération $D_T(N)$). La représentation sous forme de matrices est construite simplement, comme le montre la figure 3.2b. L'instance particulière de l'instruction T pour un vecteur d'itération (i, j) est l'opération $T(N, i, j)$.

Cet exemple montre des domaines d'itérations convexes, qui ne nécessitent pas d'union de polyèdres. De plus, pour simplifier l'explication, les domaines ont une intersection vide. Lorsque cette propriété n'est pas vérifiée, il faut introduire des nouvelles dimensions représentant l'ordre textuel des instructions (cf. section 3.3.4).

La première étape, dans un flot de compilation utilisant la représentation des boucles dans le modèle polyédrique, consiste à extraire la représentation polyédrique d'un nid de boucles décrit dans un langage impératif. Cette extraction est généralement réalisée par une analyse statique du nid de boucles. Des transformations de programme peuvent être appliquées en amont de manière à faire apparaitre des expressions affines dans les boucles. La recherche des Scop peut ensuite être

```
for (i=0;i<=N;i++) {
  for (j=0;j<N-i;j++)
    S(i,j);
  for (j=N-i+1;j<=N;j++)
    T(i,j);
}
```

$$D_S(N) = \{i, j \mid 0 \leq i < N \wedge 0 \leq j < N - i\}$$
$$D_T(N) = \{i, j \mid 0 < i \leq N \wedge N - i < j \leq N\}$$



(a) Exemple de nid de boucles ainsi que sa représentation dans le modèle polyédrique.

$$\begin{cases} 0 \leq i \\ i < N \\ 0 \leq j \\ j < N - i \end{cases} = \begin{cases} i \geq 0 \\ N - i - 1 \geq 0 \\ j \geq 0 \\ N - i - j - 1 \geq 0 \end{cases} = \begin{pmatrix} 1 & 0 \\ -1 & 0 \\ 0 & 1 \\ -1 & -1 \end{pmatrix} \begin{pmatrix} i \\ j \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} (N) + \begin{pmatrix} 0 \\ -1 \\ 0 \\ -1 \end{pmatrix} \geq 0$$

(b) Représentation du polyèdre $D_S(N)$ sous forme de matrices, suivant la notation donnée en (3.1).

FIGURE 3.2 – Exemple de nid de boucles, avec sa représentation dans le modèle polyédrique sous forme d'ensembles de points entiers délimités par des contraintes affines (a). La figure à droite représente le domaine d'itération lorsque $N = 3$. La flèche en pointillé représente l'ordre d'exécution des itérations (l'ordre lexicographique). Les domaines d'itération sont convexes, par conséquent un seul polyèdre suffit pour représenter chacun des domaines. Comme le montrent les égalités de la figure (b) pour le domaine $D_S(N)$, il est possible de retrouver la représentation sous forme de matrices en identifiant les coefficients de chaque itérateur, paramètre et constante.

réalisée grâce à des techniques de reconnaissance de motifs par exemple. Des outils permettent de réaliser cette étape automatiquement [55, 56, 57].

3.2.2 Spécification et application des transformations

Les transformations de nids de boucles ont pour objectif de changer l'ordre d'exécution des opérations. Dans un flot de compilation polyédrique ces transformations sont exprimées par des fonctions affines, une par instruction. Ces fonctions définissent les valeurs des indices du nid de boucles transformé, en fonction des indices et des paramètres du nid de boucles original :

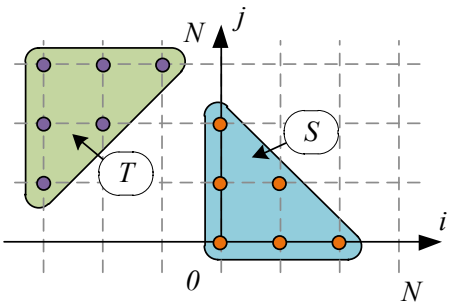
$$\begin{aligned} \theta_{S_i} : \quad & D_{S_i} \rightarrow \mathbb{Z}^m \\ & (\vec{n}, \vec{x}) \rightarrow \theta_{S_i}(\vec{n}, \vec{x}) = A_i.\vec{x} + B_i.\vec{n} + \vec{c_i} \end{aligned}$$

avec $A_i \in \mathbb{Z}^m \times \mathbb{Z}^p$, $B_i \in \mathbb{Z}^m \times \mathbb{Z}^q$ et $\vec{c_i} \in \mathbb{Z}^m$.

Dans l'exemple de la figure 3.2, inverser la boucle externe de l'instruction T revient à appliquer

la transformation θ_1 suivante aux domaines d'itération :

$$\theta_1 : \begin{cases} \theta_1^S(N,i,j) = (i,j) \\ \theta_1^T(N,i,j) = (-i,j) \end{cases}$$



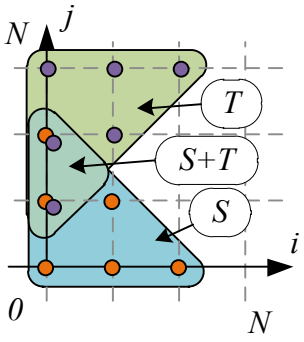

```
for (i=-N;i<0;i++)
  for (j=N+i+1;j<=N;j++)
    T(-i,j);
for (i=0;i<N;i++)
  for (j=0;j<N-i;j++)
    S(i,j);
```

FIGURE 3.3 – Application de la transformation θ_1 à l'exemple de la figure 3.2.

On remarque que la transformation du nid de boucles est composée de plusieurs fonctions affines, une par instruction. Pour ne modifier que l'ordonnancement de l'instruction T, il faut utiliser la fonction identité pour l'instruction S. L'application de θ_1 sur l'exemple de la figure 3.2 est illustrée par la figure 3.3.

La transformation θ_1 a transformé l'indice de la boucle externe pour T uniquement, et i prend maintenant des valeurs négatives. Pour retrouver un itérateur positif, il faut décaler la boucle externe englobant T de N en appliquant la transformation θ_2, illustrée par la figure 3.4 :

$$\theta_2 : \begin{cases} \theta_2^S(N,i,j) = (i,j) \\ \theta_2^T(N,i,j) = (i+N,j) \end{cases}$$




```
for (i=0;i<N;i++)
  for (j=0;j<=N;j++) {
    if (i<N-j)
      S(i,j);
    if (i<j)
      T(N-i,j);
  }
```

FIGURE 3.4 – Application de la transformation θ_2 au résultat de la transformation θ_1 de la figure 3.3.

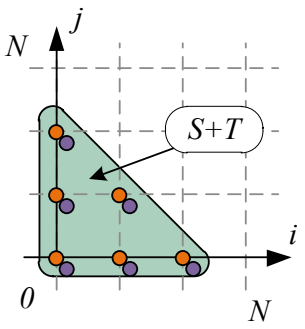
La composition des transformations θ_1 et θ_2 peut être exprimée comme une seule transfor-

mation θ_{12}, correspondant à la composition des fonctions affines les définissant :

$$\theta_{12} : \begin{cases} \theta_{12}^{S}(N,i,j) = (i,j) \\ \theta_{12}^{T}(N,i,j) = (N-i,j) \end{cases}$$

En procédant de la même manière avec la boucle interne de T, on peut construire la transformation θ_3 qui réalise une fusion des domaines d'itération des deux instructions, dont le résultat est illustré par la figure 3.5 :

$$\theta_{3} : \begin{cases} \theta_{3}^{S}(N,i,j) = (i,j) \\ \theta_{3}^{T}(N,i,j) = (N-i,N-j) \end{cases}$$




```
for (i=0;i<N;i++)
  for (j=0;j<N-i;j++) {
    S(i,j);
    T(N-i,N-j);
  }
```

FIGURE 3.5 – Application de la transformation θ_3 à l'exemple de la figure 3.2.

Le formalisme polyédrique permet ainsi de représenter des transformations complexes et de les composer de manière simple et compacte. Il est alors possible de spécifier manuellement les transformations pour améliorer la localité ou le parallélisme. Les techniques pour vérifier la légalité de ces transformations sont présentées en section 3.3, et les méthodes pour déterminer automatiquement une transformation légale en section 3.4.

3.2.3 Génération de code

Une fois les transformations appliquées, la représentation du nid de boucles est toujours sous la forme de domaines délimités par des contraintes affines. Il faut alors regénérer une représentation du nid de boucles textuelle exploitable par un compilateur.

La technique la plus utilisée consiste à regénérer une structure de nid de boucles qui parcourt le domaine d'itération transformé (cf. figure 3.6a). Elle a été mise au point par une série de travaux initiés par ANCOURT et IRIGOIN [58], puis améliorée par LE VERGE et coll. [59], QUILLERÉ et coll. [60], et BASTOUL [7]. Pour obtenir une exécution rapide, la technique utilisée génère du code avec un nombre de gardes réduit, au prix d'une taille du code plus importante.

Une autre approche, mise au point par BOULET et FEAUTRIER [5], vise à générer une machine à états finis qui parcourt le domaine d'itération transformé (cf. figure 3.6b). Comme les machines à états sont facilement implémentables en matériel, cette seconde technique semble plus adaptée à la synthèse de haut niveau.

```
for (i=0;i<N;i++)
  for (j=0;j<N-i;j++) {
    S(i,j);
    T(N-i,N-j);
  }
```

(a) Exemple de code généré sous forme de nid de boucles.

```
if (N>0) {
  i=0;j=0;
} else done = 1;
while (!done) {
  S(i,j);
  T(N-i,N-j);
  if (j<N-i-1) {
    j++;
  } else if (i<N-1) {
    i++;j=0;
  } else done = 1;
}
```

(b) Exemple de code sous forme de machine à états finis.

FIGURE 3.6 – Exemples de code généré pour l'exemple de la figure 3.2, après application de la transformation θ_3 en utilisant la technique de BASTOUL (a) et celle de BOULET et FEAUTRIER (b).

Les techniques de génération de code ainsi que l'incidence de leur choix pour la synthèse de haut niveau sont décrites plus en détails dans le chapitre 4.

3.3 Légalité des transformations

D'un point de vue syntaxique, un flot de compilation utilisant la représentation polyédrique des nids de boucles offre un formalisme compact pour représenter les domaines d'itération sous la forme d'unions de polyèdres, et regrouper un vaste ensemble de transformations sous la forme de fonctions affines. Cependant la force du modèle polyédrique réside surtout dans sa capacité à représenter de manière exacte les dépendances de données présentes au sein des nids de boucles. Il est ainsi possible de s'assurer qu'une transformation donnée est légale vis-à-vis des dépendances présentes dans le nid de boucles.

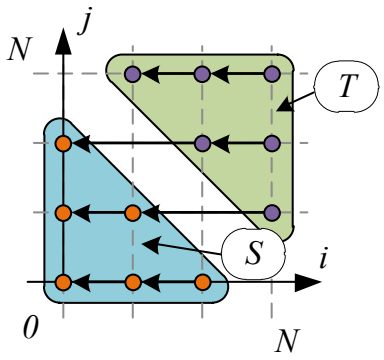
Cette section introduit dans un premier temps les dépendances, et étend le modèle de la section précédente. La sous-section 3.3.2 explique comment une transformation peut introduire une violation de dépendance, et définit les conditions de légalité d'une transformation vis-à-vis d'une dépendance. La sous-section 3.3.3 présente l'analyse de dépendances qui permet d'extraire automatiquement toutes les informations requises pour déterminer la légalité d'une transformation pour un nid de boucles.

3.3.1 Représentation des dépendances de données

La réutilisation des valeurs au sein d'un Scop implique des dépendances de données entre les opérations qui produisent et consomment ces valeurs. Cette réutilisation des valeurs est possible

si on introduit dans le modèle de programmes la notion d'accès mémoire. Par exemple, sur la figure 3.7, l'opération $S(3,1,0)$ utilise une valeur produite par l'opération $S(3,0,0)$, enregistrée à l'emplacement mémoire `t[1][0]`. On note cette dépendance $S(3,1,0) \rightarrow S(3,0,0)$.

```
for (i=0;i<=N;i++) {
  for (j=0;j<N-i;j++)
S:  t[i+1][j] = f(t[i][j]);
  for (j=N-i+1;j<=N;j++)
T:  t[i][j] = g(t[i-1][j]);
}
```



$$d_1 : S(N,i,j) \rightarrow S(N,i-1,j) : D_{d_1}(N) = \{i,j \mid i \geq 1\} \cap D_S$$
$$d_2 : T(N,i,j) \rightarrow T(N,i-1,j) : D_{d_2}(N) = \{i,j \mid j > N-i+1\} \cap D_T$$
$$d_3 : T(N,i,j) \rightarrow S(N,i-2,j) : D_{d_3}(N) = \{i,j \mid i \geq 2 \wedge j = N-i+1\} \cap D_T$$

FIGURE 3.7 – Exemple de la figure 3.2 dans lequel les instructions abstraites sont remplacées par des accès tableau. La représentation graphique montre les dépendances de données lorsque $N = 3$. Chaque flèche lie une opération qui effectue un accès en lecture à l'opération qui a produit la valeur lue. Ces dépendances sont représentées par les fonctions affines d_1, d_2 et d_3.

Il est possible de représenter d'une façon réduite l'ensemble des dépendances d'un Scop. De manière comparable aux domaines d'itérations, les dépendances sont représentées par des fonctions affines. Ces fonctions associent les opérations de lecture aux opérations qui ont produit les valeurs lues, et elles sont définies sur des domaines de validité qui représentent les sous domaines sur lesquels les dépendances s'appliquent.

Par exemple, les dépendances $S(3,1,0) \rightarrow S(3,0,0)$, $S(3,1,1) \rightarrow S(3,0,1)$ et $S(3,2,0) \rightarrow S(3,1,0)$ peuvent être factorisées en la fonction suivante, lorsque $N = 3$:

$$d_1 : S(N,i,j) \rightarrow S(N,i-1,j) : D_{d_1}(N), \text{ avec } D_{d_1}(N) = \{i,j \mid i \geq 1\} \cap D_S$$

En pratique, lorsque plusieurs opérations écrivent leurs résultats dans le même emplacement mémoire, il faut tenir compte des dépendances *mémoire* en plus des dépendances de *données*. Il est alors nécessaire de s'assurer que les valeurs, en plus d'être disponibles (dépendances RAW : *Read After Write*), ne sont pas écrasées avant d'être utilisées (WAR : *Write After Read*), et que l'état de la mémoire à la fin de l'exécution du Scop est correct (WAW : *Write After Write*). Les techniques de réallocation mémoire présentées en section 3.3.4 permettent de s'affranchir de ces dépendances WAR et WAW.

3.3.2 Transformations des dépendances et condition de légalité

Lorsqu'on applique la transformation θ_3 (définie dans la section 3.2.2) à ce nouveau nid de boucles, il faut aussi l'appliquer à ses dépendances. La version transformée par θ_3 est présentée par la figure 3.8.

```
for (i=0;i<N;i++)
  for (j=0;j<N-i;j++) {
S:  t[i+1][j] = f(t[i][j]);
T:  t[N-i][N-j] = g(t[N-i-1][N-j]);
  }
```

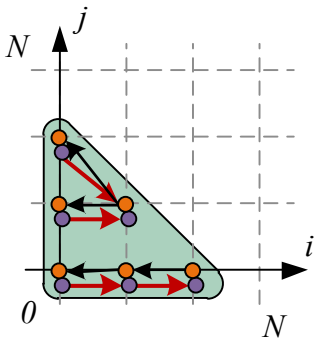


FIGURE 3.8 – Représentation du nid de boucles de la figure 3.7 après application de la transformation θ_3. Le domaine d'itération ainsi que l'ordre d'exécution sont les mêmes que sur la figure 3.5, mais la direction des flèches représentant certaines dépendances a changé. En particulier, les dépendances représentées par ces flèches sont brisées par l'application de la transformation.

Une fois transformé, le domaine est identique à celui présenté par la figure 3.5, mais la direction des flèches représentant certaines dépendances a changé par rapport à la figure 3.7. Alors qu'avant transformation les flèches liaient une lecture à une écriture passée, certaines flèches transformées par θ_3 lient une lecture à une écriture future. En d'autres termes, la valeur accédée est censée être produite dans le futur.

Par exemple, dans le nid de boucles original de la figure 3.7, l'opération $R = T(3, 2, 2)$ lit une valeur dans le tableau `t` à l'indice `t[1][2]`. À ce moment de l'exécution, la valeur lue a été produite par l'opération $W = S(3, 0, 2)$. Selon l'ordre lexicographique, $(3, 0, 2)$ précède $(3, 2, 2)$, et W (l'écriture) est bien exécutée avant R (la lecture) dans le nid de boucles original. Lorsque ces deux opérations sont transformées par θ_3, R est exécutée lorsque le vecteur d'itération vaut $(3, 1, 1)$, et W pour le vecteur d'itération $(3, 0, 2)$. Selon l'ordre lexicographique, $(3, 0, 2)$ ne précède pas $(3, 1, 1)$. Par conséquent, W (l'écriture) est exécutée après R (la lecture), et R lit donc une valeur différente de celle lue lors de l'exécution originale.

Par conséquent, appliquer la transformation θ_3 au nid de boucles de la figure 3.7 produit un programme qui ne respecte pas la causalité, et altère la sémantique du programme initial. θ_3 est donc une transformation illégale pour ce nid de boucles.

Condition de légalité

Il est possible de définir une condition selon laquelle une transformation est légale vis-à-vis des dépendances de données présentes dans un nid de boucles. Cette condition doit assurer que la transformation ne change pas le sens lexicographique des dépendances. Par exemple, pour être légale, la transformation θ_3 devrait garantir que si l'opération $T(N, i, j)$ dépend de l'opération

$S(N, i-2, j)$ dans le nid de boucles original, alors l'opération transformée de $T(N, i, j)$, soit $\theta_{3,T}(N, i, j)$, est exécutée après l'opération transformée de $S(N, i-2, j)$, soit $\theta_{3,S}(N, i-2, j)$.

À chaque domaine d'itération est associé un ordre d'exécution implicite, à savoir l'ordre lexicographique. Étant donnés deux vecteurs d'itérations (i, j) et (i', j'), (i, j) précède lexicographiquement (i', j') si $i < i'$ ou $i = i'$ et $j < j'$.

Définition : Ordre lexicographique

Soit deux vecteurs d'itération $\vec{x} \in \mathbb{Z}^m$ et $\vec{y} \in \mathbb{Z}^n$. Soit $\vec{x}_{[q]}$ la q^e composante du vecteur $\vec{x}$, et $\vec{x}_{[0..q]}$ le vecteur $(\vec{x}_{[0]}, \ldots, \vec{x}_{[q]})$. On dit alors que $\vec{x}$ précède lexicographiquement $\vec{y}$, noté $\vec{x} \prec \vec{y}$ ssi :

$$\begin{cases} \vec{x}_{[0]} < \vec{y}_{[0]} \\ \exists q \in [0..min(m,n)] \mid \vec{x}_{[0..q-1]} = \vec{y}_{[0..q-1]} \wedge \vec{x}_{[q]} < \vec{y}_{[q]} \end{cases}$$

Définition : Condition de légalité d'un ordonnancement

Soit θ une transformation. θ est légale vis-à-vis de la dépendance $d : R(\vec{n}, \vec{x}) \rightarrow S(\vec{n}, \vec{y})$ si et seulement si $\theta_R(\vec{n}, \vec{x}) \succ \theta_S(\vec{n}, \vec{y})$, pour toutes les itérations $\vec{x}$ appartenant à $D_d(\vec{n})$, le domaine de validité de d.

Dans l'exemple de la figure 3.7, le nid de boucles porte une dépendance $d_3 : T(N, i, j) \rightarrow S(N, i-2, j)$. Après transformation on a $\theta_3^T(N, i, j) = (N-i, N-j)$, et $\theta_3^S(N, i-2, j) = (i-2, j)$. Or lorsque $(N, i, j) = (3, 2, 2)$, on a $(N-i, N-j) \prec (i-2, j)$. La transformation θ_3 apparait effectivement comme illégale vis-à-vis de d_3.

3.3.3 Analyse des dépendances de données

Pour vérifier la légalité d'une transformation pour un nid de boucles, il faut vérifier la légalité de la transformation pour toutes les dépendances présentes dans le nid de boucles original. Grâce à l'approche proposée par Feautrier [61], rappelée ci-dessous, il est possible de déterminer automatiquement toutes les dépendances de *données* d'un Scop, et ainsi de s'assurer qu'une transformation respecte la causalité. D'autres approches permettent de vérifier la légalité d'une transformation vis-à-vis des dépendances *mémoire* [62].

Pour rappel, une dépendance de données directe d existe entre deux opérations $R(\vec{n}, \vec{x})$ et $S(\vec{n}, \vec{y})$ si l'opération $R(\vec{n}, \vec{x})$ lit une valeur produite par l'opération $S(\vec{n}, \vec{y})$. Une telle dépendance existe donc si les instructions R et S accèdent toutes les deux à un même tableau, et à la même adresse. Pour que la référence lise une valeur produite par la source lors de l'exécution originale, la source précède lexicographiquement la référence. Enfin, la valeur lue par la référence est la dernière produite selon l'ordre lexicographique.

Dans l'exemple de la figure 3.7, à une itération donnée (N, i, j) l'instruction T accède au tableau `t` en lecture à l'indice `t[i-1][j]`. Les deux instructions qui écrivent dans `t` sont T, à l'indice `t[i][j]` et S, à l'indice `t[i+1][j]`. L'ensemble des opérations qui ont écrit dans

`t[i-1][j]` lors de l'opération $T(N, i, j)$ sont alors les opérations $S(N, i', j')$ et $T(N, i'', j'')$ qui respectent les conditions suivantes :

- Pour S, on construit l'ensemble DC_S des candidats $S(N, i', j')$:
  - $S(N, i', j') \prec T(N, i, j)$: l'écriture précède la lecture ;
  - $(i' + 1, j') = (i - 1, j)$: les deux opérations accèdent la même cellule de `t` ;
  - $(i, j) \in D_T(N)$: l'opération de lecture existe ;
  - $(i', j') \in D_S(N)$: l'opération d'écriture existe ;
- Pour T, on construit l'ensemble DC_T des candidats $T(N, i'', j'')$:
  - $T(N, i'', j'') \prec T(N, i, j)$: l'écriture précède la lecture ;
  - $(i'', j'') = (i - 1, j)$: les deux opérations accèdent la même cellule de `t` ;
  - $(i, j) \in D_T(N)$: l'opération de lecture existe ;
  - $(i'', j'') \in D_T(N)$: l'opération d'écriture existe.

L'opération qui produit la valeur lue dans `t[i-1][j]` lors de l'opération $T(N, i, j)$ est alors la dernière opération selon l'ordre lexicographique, parmi celles remplissant les conditions ci-dessus. En d'autres termes, c'est le maximum lexicographique de $DC_S \cup DC_T$. Le résultat peut être calculé grâce à la programmation entière paramétrique (Pip [63, 64]) et prend la forme d'un Quast (*Quasi-Affine Selection Tree* : Arbre de sélection quasi-affine), représentable sous la forme de fonctions quasi-affines [1] par morceaux. Pour cet exemple, il correspond aux dépendances d_2 et d_3 de la figure 3.7.

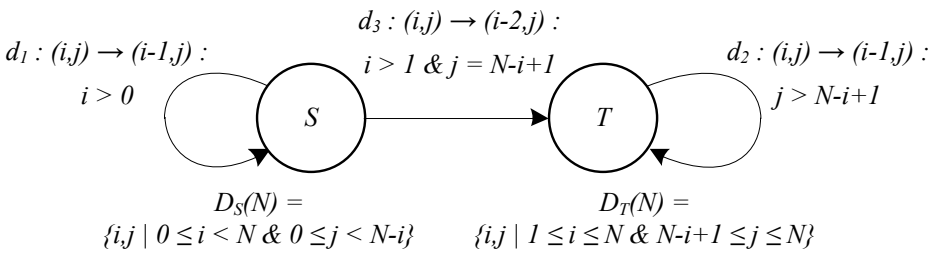


Figure 3.9 – PRDG pour le nid de boucles de la figure 3.7.

Lorsque l'ensemble des dépendances est calculé pour un nid de boucles, le graphe représentant ces dépendances est appelé PRDG (*Polyhedral Reduced Dependence Graph* : Graphe des dépendances polyédriques réduites). Dans un Prdg, chaque nœud représente une instruction et son domaine d'itération associé, et chaque arc représente une dépendance, valué par sa fonction de dépendance et le domaine sur lequel s'applique la dépendance. Le Prdg du nid de boucles de la figure 3.7 est présenté par la figure 3.9.

1. Les fonctions quasi-affines sont des fonctions affines dans lesquelles la division et le modulo par une constante sont autorisés. Il est possible de les représenter sous la forme de fonctions affines en insérant des dimensions.

3.3.4 Extensions

Les approches présentées précédemment peuvent être étendues pour couvrir une classe de programme plus vaste. Cette sous-section présente les techniques utilisées pour :

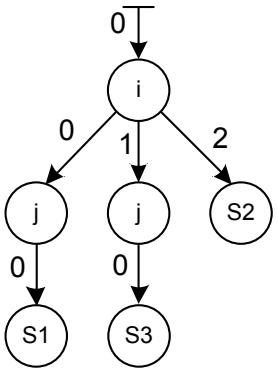
- intégrer l'ordre textuel comme partie de l'ordre lexicographique ;
- étendre l'applicabilité des approches au prix d'une perte en précision ;
- s'affranchir des dépendances *mémoire* ;
- spécifier et vérifier la légalité d'ordonnancements parallèles.

Intégration de l'ordre textuel

Les domaines d'itérations des instructions d'un Scop ne sont pas forcément mutuellement exclusifs. Différentes instructions d'un même Scop peuvent même avoir des domaines d'itérations équivalents. Dans de tels cas, l'ordre lexicographique ne suffit pas pour ordonner les opérations, condition nécessaire pour pouvoir appliquer l'analyse de dépendances.

Dans l'exemple de la figure 3.10, les deux instructions S_1 et S_2 ont le même domaine d'itération. Lors de la lecture de `t[i][0]` par l'opération $S_3(N, i)$, deux opérations $S_1(N, i, 0)$ et $S_2(N, i, 0)$ peuvent avoir écrit dans cette mémoire. Or leurs vecteurs d'itérations sont égaux et l'ordre lexicographique ne suffit pas pour déterminer laquelle a été exécutée en dernier.

```
for (i=0; i < N; i++)
    for (j=0; j < N; j++)
S1:     t[i][j] = foo();
    for (j=0; j < N; j++)
S2:     t[i][j] = bar();
S3: s += t[i][0];
```



$$S_1(i, j) = (0, i, 0, j, 0)\,; \qquad S_2(i, j) = (0, i, 1, j, 0)\,; \qquad S_3(i) = (0, i, 2, 0, 0)$$

FIGURE 3.10 – Exemple de nid de boucles, dont l'arbre de syntaxe abstraite est illustré à droite. Chaque opération est repérée par un vecteur d'itération dont chaque indice est précédé par le rang de l'instruction dans le texte de la boucle correspondant à cet indice. Les fonctions d'ordonnancement originales pour ce nid de boucles sont présentées en bas.

Pour déterminer l'ordre des opérations, il suffit de compléter le vecteur d'itération avec le rang de l'instruction dans l'ordre textuel du programme [65]. Cette technique est illustrée par la figure 3.10. Cet ordonnancement des opérations est aisément construit à parti de l'arbre de syntaxe abstraite de la boucle.

Applicabilité

Les techniques présentées précédemment ne sont applicables que pour des Scop, pour lesquels les bornes des boucles, les gardes, et les fonctions d'accès des tableaux sont des fonctions affines des itérateurs de boucles englobantes et des paramètres du nid de boucles.

Sans ces conditions, l'analyse de dépendances, qui requiert le calcul du maximum lexicographique sur un ensemble de points entiers, ne pourrait pas être réalisée automatiquement. Comme l'a présenté Bastoul dans sa thèse [66], cette classe de programme est largement représentée dans les applications réelles.

Néanmoins ces contraintes peuvent être partiellement relâchées pour améliorer l'applicabilité de ce modèle, au prix d'une perte en précision [67, 68, 69]. Par exemple lorsque les accès au tableau ne sont pas affines, il est possible de considérer que le tableau est accédé dans son intégralité, ou sur une sous-région.

Modèle d'allocation mémoire

L'analyse de dépendances présentée ci-dessus ne considère que les dépendances de données de type Raw. La vérification d'une transformation en ne tenant compte que de ce type de dépendance permet seulement de s'assurer qu'une valeur a bien été produite avant qu'elle ne soit lue. Par contre, rien n'assure que la valeur n'a pas été écrasée par une autre écriture au même emplacement mémoire entre temps.

Par exemple dans la figure 3.11, la transformation θ est légale vis-à-vis des dépendances de données (graphiquement, la direction de la flèche change, mais pas le sens lexicographique). Par contre, dans le domaine d'itération transformé, l'opération $S_1(1,1)$ s'exécute entre la production de la valeur $S_1(1,0)$ et sa lecture $S_1(2,1)$. Or, le résultat de l'opération $S_1(1,1)$ est écrit dans la même variable `s` que les autres opérations, ce qui implique une violation de dépendance de type War. Pour être légales vis-à-vis de l'allocation mémoire originale, les transformations dans le modèle polyédrique doivent prendre en compte les dépendances de type War et Waw.

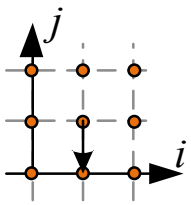
Il est cependant possible de s'abstraire de l'allocation mémoire originale en considérant que chaque variable (tableau ou scalaire) accédée dans le nid de boucles est étendue au nombre de dimensions des boucles qui l'entourent. Avec une telle abstraction, on ne manipule que des tableaux dans lesquels chaque cellule n'est accédée qu'une seule fois en écriture. On parle alors d'affectation unique des tableaux, appelée SA polyédrique (pour *Single Assignment* : Assignation Unique) dans la suite, comme c'est le cas pour l'exemple de la figure 3.12. Ce type d'allocation est implicite dans les programmes décrits sous forme de systèmes d'équations récurrentes (cf. section 3.5.1).

Lorsque le nid de boucles est sous une forme Sa polyédrique, les dépendances War et Waw peuvent alors être ignorées, car les valeurs produites ne sont jamais écrasées. Cependant dans le cas de nid de boucles de profondeur élevée, on obtient ainsi des tableaux de très grande taille, et la consommation mémoire devient prohibitive.

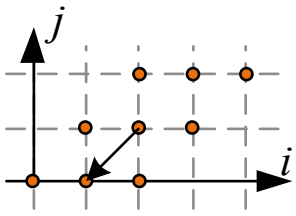
Néanmoins, la représentation à assignation unique peut être utilisée pour simplifier l'analyse et la recherche du parallélisme, et ultérieurement complétée par des techniques de contraction de tableau [70, 71] afin d'obtenir une empreinte mémoire raisonnable. L'objectif de ces techniques est

```
for (i=0; i<N; i++)
S0: s = 0;
    for (j=0; j<N; j++)
S1:     s += t[i][j];
S2: sum[i] = s;
```

(a) Exemple de nid de boucles avec plusieurs instructions.



(b) Représentation graphique du domaine d'itération de S_1.



(c) Représentation graphique du domaine d'itération de S_1 après application de $\theta(N,i,j) = (i+j, j)$.

FIGURE 3.11 – Exemple de nid de boucles calculant les sommes des colonnes du tableau `t`, en se servant d'une variable temporaire `s`. La représentation graphique du domaine d'itération original de S_1, ainsi qu'une instance de la dépendance sur l'accumulation, sont présentées en (b) quand $N = 3$. La figure (c) montre la représentation du domaine d'itération et de la dépendance transformées par la fonction $\theta(N,i,j) = (i+j, j)$. Cette transformation est légale vis-à-vis de la production des données, mais illégale vis-à-vis de la dépendance WAR portée par la variable `s`.

```
for (i=0; i<N; i++)
S0: s[i][0] = t[i][0];
    for (j=1; j<N; j++)
S1:     s[i][j] += t[i][j];
S2: sum[i] = s[i][N-1];
```

FIGURE 3.12 – Exemple de la figure 3.11a dans lequel la variable `s` est étendue au nombre de dimensions de la boucle interne. Chaque cellule de `s` est alors affectée une seule fois.

de déterminer une nouvelle allocation mémoire qui reste valide lorsqu'on applique une transformation donnée, en essayant de minimiser sa taille. D'autres techniques permettent de déterminer une allocation mémoire valide quelle que soit la transformation [72].

Ordonnancements parallèles

Les ordonnancements sont décrits sous la forme de fonctions affines des indices et de l'ordre textuel des boucles. Un ordonnancement définit un nouvel ordre d'exécution pour les opérations d'un Scop. Cependant à ce niveau rien ne permet de spécifier quelles sont les boucles parallèles dans l'ordonnancement, ni même de vérifier si une boucle spécifiée comme parallèle respecte toutes les dépendances.

Pour définir un ordonnancement parallèle, la méthode la plus simple consiste à indiquer quelles sont les dimensions de l'espace d'itération qui sont exécutées de manière parallèle. Alors que les transformations décrites précédemment étaient définies sur $D_{S_i} \to \mathbb{Z}^m$:

$$\begin{aligned} \theta_{S_i} : \quad & D_{S_i} \to \mathbb{Z}^m \\ & (\vec{n}, \vec{x}) \to \theta_{S_i}(\vec{n}, \vec{x}) = A_i.\vec{x} + B_i.\vec{n} + \vec{c_i} \end{aligned}$$

ces nouvelles transformations sont définies sur $D_{S_i} \rightarrow (\mathbb{Z} \times \mathbb{B})^m$. On associe ainsi à chaque dimension un booléen qui détermine si elle est parallèle ou non.

Par exemple, sur la transformation θ_3 présentée en section 3.2.2, pour spécifier que la deuxième dimension est parallèle, il faut la définir sur $D_{S_i} \rightarrow (\mathbb{Z} \times false) \times (\mathbb{Z} \times true)$.

Pour vérifier un ordonnancement parallèle, il faut s'assurer que la causalité est respectée pour les dimensions séquentielles, et qu'aucune dépendance n'est portée ni brisée par les dimensions parallèles.

3.4 Transformations automatiques

La représentation polyédrique permet de transformer un Scop, et de vérifier automatiquement la légalité des transformations grâce à l'analyse de dépendances de données sur les tableaux. Pour compléter l'automatisation du flot, il reste à déterminer automatiquement un ordonnancement légal. L'objectif est d'améliorer les performances lors de l'exécution du Scop transformé, en exposant des boucles parallèles, en améliorant la localité, ou en réduisant l'empreinte mémoire de la boucle.

Trouver un ordonnancement légal revient à trouver un ordonnancement qui ne viole aucune dépendance du PRDG. Une dépendance d est représentée sous la forme d'une fonction affine de l'opération qui lit une valeur (cf. figure 3.7) $d : R(\vec{n}, \vec{x}) \rightarrow S(\vec{n}, \vec{y}) : \vec{x} \in D_d(\vec{n})$. Un ordonnancement θ est légal vis-à-vis de cette dépendance de données si pour toutes les instances de la dépendance d (pour tous les $\vec{x}$ dans $D_d(\vec{n})$), $\theta_R(\vec{n}, \vec{x})$ s'exécute après $\theta_S(\vec{n}, \vec{y})$ selon l'ordre lexicographique.

3.4.1 Le cas monodimensionnel

Dans un premier temps, on s'intéresse aux ordonnancements qui associent, à chaque itération du domaine original, une date d'exécution dans $\mathbb{Z}$. On peut écrire le prototype d'une telle fonction d'ordonnancement :

$$\theta : \begin{cases} \theta_R(\vec{n}, \vec{x}) = \vec{a}_R.\vec{x} + \vec{b}_R.\vec{n} + c_R \\ \theta_S(\vec{n}, \vec{y}) = \vec{a}_S.\vec{y} + \vec{b}_S.\vec{n} + c_S \end{cases}$$

Avec dans un premier temps $\vec{a}_R$, $\vec{b}_R$, $\vec{a}_S$ et $\vec{b}_S$ des vecteurs constants, c_R et c_S deux constantes scalaires.

Comme décrit dans [73, 74] l'ordonnancement θ est donc valide vis-à-vis de d si :

$$\vec{a}_R.\vec{x} + \vec{b}_R.\vec{n} + c_R \geq \vec{a}_S.\vec{y} + \vec{b}_S.\vec{n} + c_S + 1, \text{ pour tout } \vec{x} \in D_d(\vec{n})$$

Ces inéquations ne donnent pas directement une méthode pour trouver des ordonnancements, parce qu'elles ne sont pas affines. Deux approches ont été proposées pour se ramener à un problème de programmation linéaire. Détailler ces approches est hors du cadre de cette thèse, elles sont simplement rappelées brièvement ci-dessous.

Méthode des sommets

Dans un premier temps, QUINTON [75, 76], RAJOPADHYE [77, 78] et coll. [73] ont proposé une approche basée sur la représentation des polyèdres sous forme de sommets et de rayons, appelée représentation duale. L'approche se base sur le fait que si une condition affine est valide pour l'ensemble des sommets d'un polyèdre, alors elle est valide pour l'ensemble des points de ce polyèdre. La représentation duale peut être construite automatiquement via l'algorithme de CHERNIKOVA [79], et permet d'obtenir des coordonnées rationnelles pour tous les sommets. Le polyèdre de dépendances est alors construit en remplaçant les occurrences des itérateurs par les coordonnées des sommets. Cependant un hyper cube de dimension n définit par $2n$ contraintes affines (n boucles imbriquées) est représenté par 2^n sommets dans la représentation duale.

Forme affine du lemme de FARKAS

Pour éviter des cas où le nombre de sommets serait trop grand pour une résolution du problème, FEAUTRIER [74] propose d'utiliser la forme affine du lemme de FARKAS. D'après ce lemme, on peut reformuler les contraintes du polyèdre de dépendances par des combinaisons affines des contraintes du domaine d'itération original. En développant la contrainte et la combinaison affine, en factorisant par les itérateurs, puis en procédant à une identification, on obtient un système de contraintes affines qui ne dépendent que des coefficients de l'ordonnancement et des multiplicateurs de FARKAS. Après construction d'un tel système pour chaque dépendance, on obtient le polyèdre de dépendances, définissant l'ensemble des ordonnancements légaux.

Sélection d'un ordonnancement

Quelle que soit l'approche utilisée (la méthode des sommets ou la forme affine du lemme de FARKAS), le polyèdre de dépendances définit un ensemble potentiellement infini d'ordonnancements légaux. Trouver une solution (un point entier du polyèdre) définissant un ordonnancement légal peut être résolu grâce à l'algorithme du simplex. La recherche d'une bonne solution passe par la définition de critère de qualité pour les ordonnancements. Par exemple, FEAUTRIER [74] propose d'exprimer la latence totale du nid de boucles, puis de minimiser cette latence pour favoriser le parallélisme. Dans ce même article, il propose aussi de chercher à minimiser la distance des dépendances pour favoriser la localité.

3.4.2 Le cas multidimensionnel

Certains nids de boucles n'admettent pas d'ordonnancement monodimensionnel sur $\mathbb{Z}$. C'est le cas de l'exemple de nid de boucles en figure 3.13a.

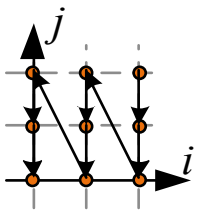
Mais on peut remarquer que ce nid de boucle admet néanmoins au moins un ordonnancement monodimensionnel :

$$\theta(N, i, j) = i \times N + j$$

cet ordonnancement n'est pas affine ($i \times N$), et ne peut pas être le résultat des approches précédentes. On peut remarquer qu'il existe pourtant un ordonnancement affine bi-dimensionnel

```
for(i=0;i<N;i++){
  for(j=0;j<N;j++){
    s += t[i][j];
```

(a) Accumulation des valeurs d'un tableau sur la variable `s`.



(b) Représentation graphique du domaine d'itération est des dépendances de (a) portées par la variable `s`.

FIGURE 3.13 – Exemple de nid de boucles (a) pour lequel aucun ordonnancement affine monodimensionnel n'est légal. La représentation graphique de son domaine d'itération ainsi que ses dépendances est présentée en (b) quand $N = 3$.

légal évident :

$$\theta(N, i, j) = (i, j)$$

On parle ainsi de temps logique multidimensionnel. θ est alors défini comme suit :

$$\theta : \begin{cases} \theta_R(\vec{n}, \vec{x}) = A_R.\vec{x} + B_R.\vec{n} + \vec{c}_R \\ \theta_S(\vec{n}, \vec{y}) = A_S.\vec{y} + B_S.\vec{n} + \vec{c}_S \end{cases}$$

où A_R, B_R, A_S et B_S sont des matrices constantes, et $\vec{c}_R$ et $\vec{c}_S$ deux vecteurs constants.

FEAUTRIER [65] définit les contraintes sur un ordonnancement multidimensionnel légal vis-à-vis d'une dépendance $d : R(\vec{n}, \vec{x}) \rightarrow S(\vec{n}, \vec{y}) : \vec{x} \in D_d(\vec{n})$ comme suit :

$$A_R.\vec{x} + B_R.\vec{n} + \vec{c}_R \succ A_S.\vec{y} + B_S.\vec{n} + \vec{c}_S, \text{ pour tout } \vec{x} \in D_d(\vec{n})$$

Le système à résoudre est une disjonction parce que l'ordre est lexicographique. Pour le résoudre, on peut itérer sur les dimensions de l'ordonnancement recherché, et appliquer l'approche monodimensionnelle (méthode des sommets ou FARKAS) à chaque étape, mais sans chercher à satisfaire toutes les dépendances. Les dépendances non satisfaites à un niveau d'itération donné le seront à une itération suivante, dans une dimension plus interne de l'ordonnancement, s'il existe une solution.

3.4.3 Transformations optimisantes

Pour extraire le maximum de parallélisme, FEAUTRIER propose une heuristique gloutonne [65] qui détermine un ordonnancement multidimensionnel qui satisfait le maximum de dépendances par dimensions. La technique consiste à calculer les composantes fortement connexes du PRDG, et à ordonnancer les composantes entres elles dans un premier temps, puis de recommencer avec les composantes elles mêmes. Néanmoins extraire le maximum de parallélisme possible ne conduit pas toujours à une exécution plus rapide. En effet un ordonnancement parallèle peut impliquer une mauvaise localité des accès mémoires.

Dans l'optique de trouver des ordonnancements qui améliorent la localité, l'heuristique implémentée dans Pluto [2] détermine des ordonnancements qui s'assurent que la référence (la lecture) d'une dépendance n'est plus supérieure lexicographiquement à la source, mais que le délai sur chaque dimension séparant la référence de la source soit supérieur ou égal à 0. Ces conditions assurent alors que des transformations de pavage peuvent être appliquées sur le nid de boucles transformé, transformations qui améliorent la localité (cf. section 2.3.2) et permettent l'exploitation de parallélisme à gros grain conduisant à des gains en performance très significatifs.

3.5 Application du modèle polyédrique à la HLS

Parmi les travaux réalisés autour du modèle polyédrique, peu se sont intéressés aux problématiques de l'application d'un tel formalisme dans le contexte de la HLS (*High-Level Synthesis* : Synthèse de Haut Niveau) d'accélérateurs matériels. Les premiers travaux concernent la synthèse de systèmes d'équations récurrentes vers des architectures systoliques. En dehors de la génération de code, détaillée dans le chapitre suivant, les travaux récents se concentrent sur des optimisations au niveau des accès mémoires, en particulier lors des accès à des données stockées sur des mémoires externes de type Ddr-Sdram (*Double Data Rate Synchronous Dynamic RAM*).

3.5.1 MMalpha

Les premiers travaux liant le modèle polyédrique à la synthèse de haut niveau ont été conduits dans l'objectif de générer des architectures systoliques à partir d'une description d'un programme sous forme de système d'équations affines récurrentes [80], parfois référencé sous l'appellation *polyhedral equational model*.

Les Sare (*System of Affine Recurrence Equation* : Système d'Équations Affines Récurrentes) décrivent les programmes sous forme d'équations mathématiques. Ces équations permettent d'affecter des valeurs à des variables (tableaux ou scalaires), en fonction d'expressions définies sur des domaines d'itérations polyédriques. Les cellules des variables ne sont affectées qu'une seule fois, sous la forme SA polyédrique et les dépendances sont décrites explicitement dans les expressions des équations. Les dépendances décrites de manière explicite permettent d'exprimer des Sare qui n'admettent pas d'ordonnancement [74], à l'inverse des Scop qui sont par définition ordonnançables.

Les Sare peuvent être décrits dans le langage fonctionnel Alpha [81, 82], développé pour MMalpha, ou Alphabets, récemment développé pour AlphaZ [83]. À la différence de MMalpha, qui cible les architectures systoliques sur FPGA (*Field Programmable Gate Array* : Réseau de Portes Programmables), AlphaZ s'intéresse à la génération de code pour les processeurs multi- et many-cœurs. À titre d'exemple, la figure 3.14 représente le nid de boucles de la figure 3.7 dans le langage Alphabets.

À partir d'une représentation sous forme de Sare, MMalpha fournit un environnement de transformation, ainsi que des mécanismes permettant la synthèse d'une architecture systolique [84] exécutant le Sare. Une approche similaire est utilisée dans l'outil PARO [85].

```
int f (int) ;
int g (int) ;
affine exemple {N| N>=0 }
      input
            int t_in {i,j| i>=0 && i<=N && j>=0 && j<=N && N>0 } ;
      output
            int t_out {i,j| i>=0 && i<=N && j>=0 && j<=N && N>0 } ;
      local
            int S {i,j| j>=0 && i+j<N && i>=0 && i<=N && N>=0 } ;
            int T {i,j| i+j>N && j<=N && i>=0 && i<=N && N>=0 } ;
      let
            S[i,j] = (case
                  {i,j| i==0 } : f(t_in[i,j]) ;
                  {i,j| j>=0 && i+j<N && i>0 } : f(S[i-1,j]) ;
            esac);
            T[i,j] = (case
                  {i,j| j==N && i==1 } : g(t_in[i-1,j]) ;
                  {i,j| j==N-i+1 && i<=N && i>1 } : g(S[i-2,j]) ;
                  {i,j| j>N-i+1 && j<=N && i<=N } : g(T[i-1,j]) ;
            esac);
            t_out[i,j] = (case
                  {i,j| i==0 } : t_in[i,j] ;
                  {i,j| i>0 && j<=N-i } : S[i-1,j] ;
                  {i,j| j>N-i } : T[i,j] ;
            esac);
.
```

FIGURE 3.14 – Nid de boucles de la figure 3.7 décrit sous forme de Sare dans le langage Alphabets.

MMalpha peut déterminer un ordonnancement multi-dimensionnel. Cependant, s'il est possible de générer un programme C, il ne dispose pas du générateur de matériel correspondant. GUILLOU, RISSET et QUINTON ont proposé par la suite une technique permettant la génération de contrôleurs pour les ordonnancements multidimensionnels [86].

En utilisant un ordonnancement monodimensionnel, toutes les opérations qui peuvent s'exécuter en parallèle se voient associer une même date d'exécution. Il faut ensuite déterminer une allocation des opérations sur les processeurs qui évite les interconnexions entre processeurs éloignés, de manière à pouvoir générer un réseau systolique où les processeurs ne communiquent qu'avec leurs voisins directs. Une telle allocation est possible en uniformisant les Sare [76]. La complétion unimodulaire de la matrice d'ordonnancement permet finalement d'associer à chaque opération un processeur qui va l'exécuter [87]. Le résultat est alors un programme Alpha transformé, représentant une architecture virtuelle, pour laquelle il reste à générer du code (cf. section 4.3.2). La figure 3.15 présente le flot sur un exemple de produit matriciel.

3.5.2 Réseau de processus polyédriques

En s'inspirant de la représentation des applications sous la forme de KPN (KAHN *Process Network* : Réseau de Processus de KAHN), et dans le contexte du projet Compaan [88, 89, 90], RIJPKEMA et coll. ont proposé une technique pour dériver des PPN (*Polyhedral Process Net-*

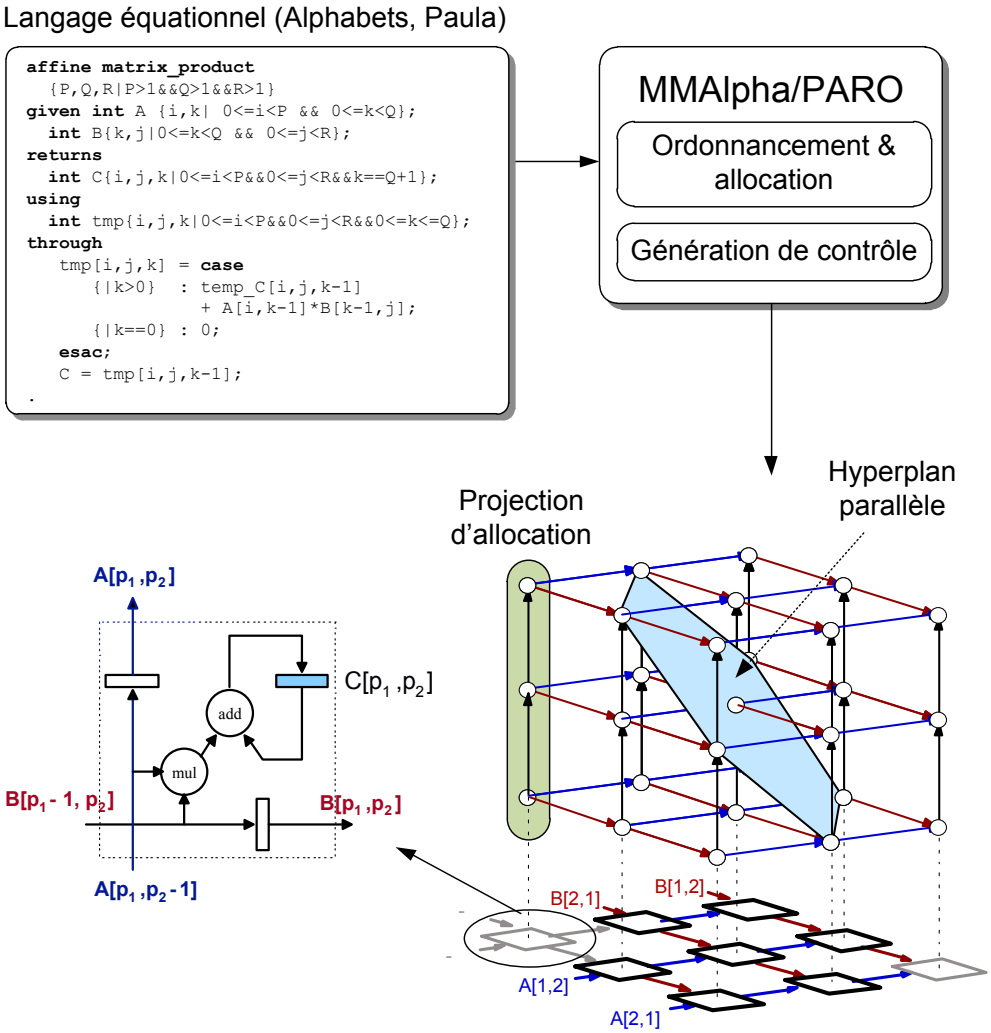


FIGURE 3.15 – Exemple de multiplication de matrices synthétisé par MMalpha sous forme de réseau systolique. À partir de la description sous forme de Sare, MMalpha recherche un ordonnancement parallèle (l'hyperplan parallèle). Dans un second temps, il faut allouer des processeurs aux opérations, en projetant la dimension du temps, et générer le contrôle. Enfin, l'architecture systolique est générée, consistant en un réseau régulier de processeurs.

work : Réseau de Processus Polyédriques) à partir de Scop (cf. figure 3.16). Les PPN représentent un sous-ensemble des KPN dans lesquels il est possible de borner la taille des canaux de communication.

L'approche construit un réseau de processus dans lequel chaque nœud correspond à au moins une instruction du Scop (parfois à plusieurs pour limiter les coûts en surface), et chaque canal représente une dépendance entre instructions, déterminée grâce à l'analyse de dépendances (cf. section 3.3). Les nœuds sont chargés d'exécuter les opérations, et de parcourir les domaines d'itérations des instructions. Les canaux peuvent être matérialisés sous différents types. On retrouve

ainsi des FIFO pour les communications inter-nœuds, et des registres pour les communications internes.

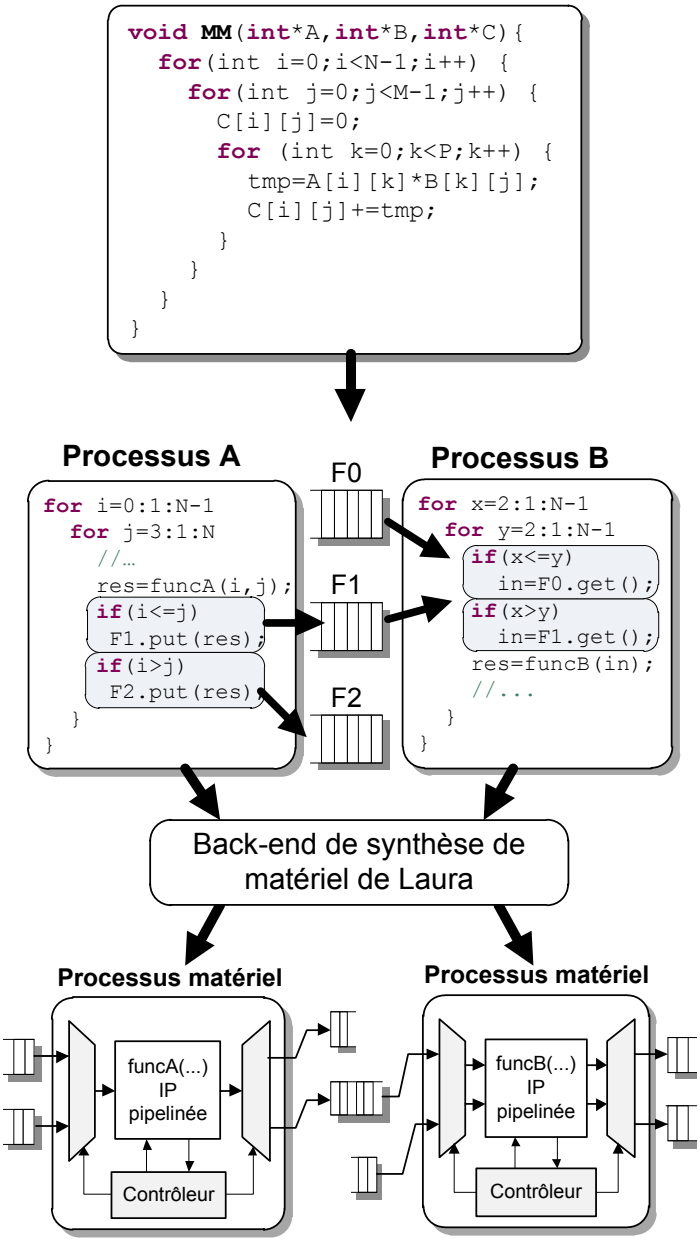


FIGURE 3.16 – Exemple de réseau de processus polyédriques synthétisé par Laura [91].

L'approche ne nécessite pas d'ordonnancement statique, et fonctionne sur le principe des lectures bloquantes. Chaque processus est matérialisé par un accélérateur dédié, qui s'exécute en parallèle des autres, tant que des données sont disponibles. Cependant, certains cas nécessitent un ordonnancement, en particulier lorsque plusieurs instructions sont exécutées sur un même nœud. De plus borner la taille des canaux nécessite le calcul d'un ordonnancement, qui ne sera pas

forcément respecté durant l'exécution. Cette borne est calculée en déterminant une expression du nombre d'élements en transit dans le canal [92, 93], puis en recherchant une borne de cette expression [94]. Ces canaux de communication sous forme de FIFO sont particulièrement adaptés à une implémentation matérielle.

3.5.3 Optimisation des accès mémoire

Lors de la conception d'accélérateurs matériels grâce à la HLS, la gestion des accès mémoires autorise bien plus de liberté que la hiérarchie rigide des architectures programmables généralistes (cf. section 2.3.2).

Dans le cas des architectures systoliques, les travaux liés à MMalpha ont montré qu'il est possible de trouver un ordonnancement qui permet de limiter la communication d'un processeur à ses voisins directs, et de stocker des données localement. Cela permet de réduire grandement les communications avec la mémoire externe, et de favoriser la réutilisation des données au sein de l'accélérateur.

De manière plus générale, PLESCO propose dans sa thèse [95] une approche en deux étapes, qui permet de minimiser les communications avec la mémoire externe, mais aussi de minimiser l'empreinte mémoire sur l'accélérateur. Basée sur la transformation de pavage (*tiling*), la technique consiste, dans un premier temps, à identifier les données utilisées par chaque tuile. Une fois identifiées, les données sont accédées de manière consécutive, afin de profiter au maximum des caractéristiques des mémoires DDR-SDRAM, puis elles sont chargées dans une mémoire locale de type *scratchpad* (Buffer1 et 2 sur la figure 3.17).

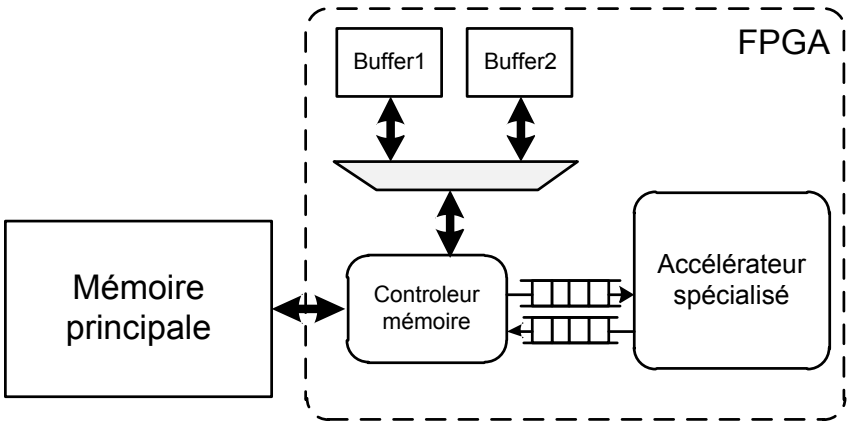


FIGURE 3.17 – Exemple d'architecture comprenant un contrôleur mémoire qui charge les données accédées par une tuile dans des mémoires locales (*scratchpad*).

Au moment de la synthèse, l'approche détermine précisément les lectures et les écritures qui doivent être effectuées pour une tuile donnée. Les accès en mémoire sont ensuite réalisés par un contrôleur mémoire spécifique à l'application. Pour maximiser la réutilisation des données entre les différentes tuiles, l'approche recherche un ordonnancement similaire à ce que propose Pluto [2], en maximisant le nombre de dépendances satisfaites par les dimensions les plus internes. Pour finir, l'empreinte des mémoires locales est minimisée grâce aux techniques de contraction de

tableaux, appliquées aux mémoires locales. En utilisant plusieurs mémoires locales, il est possible de pipeliner la lecture et l'écriture des données avec l'exécution des tuiles. De cette manière il est possible de masquer la latence des accès à la mémoire DDR-SDRAM (cf. trace d'exécution en figure 3.18).

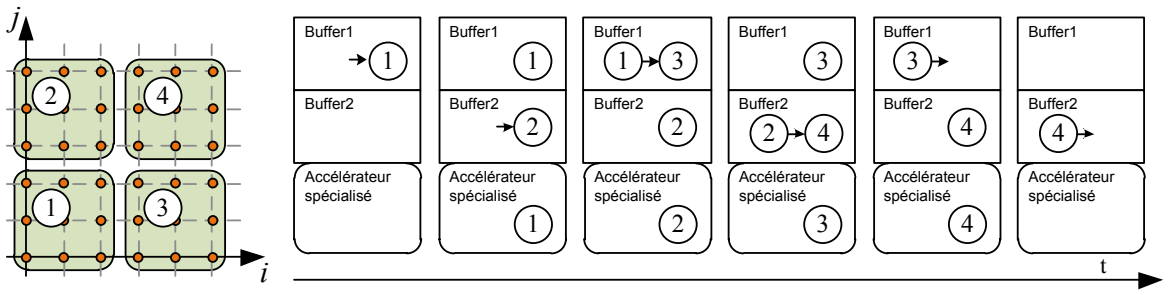


FIGURE 3.18 – L'exécution d'une tuile est entrelacée avec l'écriture des données de la tuile précédente et le chargement des données de la tuile suivante. Les accès à la mémoire globale sont pipelinés avec l'exécution des tuiles.

3.6 Conclusion

Développé depuis plus de trois décennies, le modèle polyédrique permet de transformer de manière efficace les Scop pour produire des implémentations parallèles. Les recherches récentes sur ce modèle ont permis la réalisation d'outils logiciels qui en facilitent l'analyse [7, 63, 64, 96, 97, 98].

Ces techniques ont été essentiellement développées et utilisées dans le cadre de la compilation logicielle et pour le HPC (*High-Performance Computing* : Calcul Haute Performance). Néanmoins, des travaux récents mettent en avant ces techniques dans le contexte de la HLS [14, 95]. De plus, les techniques de parallélisation et de pavage s'appliquent aussi bien dans un contexte de HLS. Cependant, il reste beaucoup de place pour des améliorations, en particulier au niveau de la génération de code.

Chapitre 4

Génération de contrôle pour le modèle polyédrique

4.1 Introduction

Une fois que le parallélisme d'un programme ou sa localité ont été améliorés grâce aux transformations polyédriques, l'ensemble des opérations reste représenté à l'intérieur du modèle polyédrique, comme opération entre « polyèdres » de données. Cette représentation ne permet pas une exécution immédiate, que ce soit logicielle ou matérielle. Le problème est alors de repasser dans une représentation algorithmique qui permet l'exécution des boucles, et contrôle le chemin de données.

Ce problème de génération de contrôle est essentiel pour une mise en œuvre performante des transformations dans le modèle polyédrique. Celles-ci peuvent conduire à des domaines d'itération définis par des contraintes plus complexes et plus nombreuses que dans le programme original. Il faut alors s'assurer que les gains en parallélisme ou localité obtenus par la transformation ne sont pas anéantis par un surcoût en contrôle devenu prépondérant lors de l'exécution.

Cette section traite le problème de la génération de code pour le parcours de polyèdres, dans le contexte de la génération de matériel. La problématique ainsi que les deux approches existantes pour parcourir des polyèdres sont présentées dans la première section. Des techniques de synthèse de matériel ont été développées à partir des générateurs de code. Elles sont comparées en section 4.3. La section 4.4 présente les optimisations apportées pour améliorer la qualité des circuits générés.

4.2 Parcours de polyèdres

4.2.1 Problématique et approches basiques

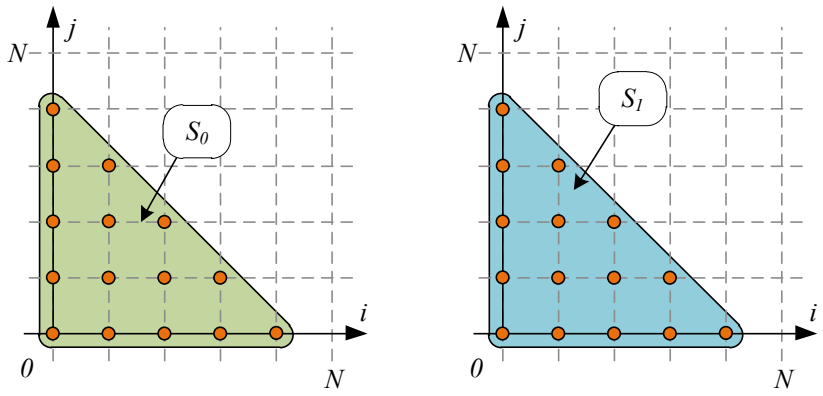
La problématique est relativement simple à énoncer. Il faut parcourir, suivant l'ordre lexicographique, un domaine d'itération défini par des contraintes affines, et déterminer quelles sont les instructions à exécuter pour chaque vecteur d'itération. Pour que le programme parcourant

```
for (i=0; i<N; i++)
  for (j=0; j<N-i; j++)
    S0(i,j);
for (i=0; i<N; i++)
  for (j=0; j<N-i; j++)
    S1(i,j);
```

(a) Exemple de nid de boucles original.

$$\mathcal{D}_{S_0}(N) = \{0, i, j \mid 0 \leq i < N \ \wedge \ 0 \leq j < N - i\}$$
$$\mathcal{D}_{S_1}(N) = \{1, i, j \mid 0 \leq i < N \ \wedge \ 0 \leq j < N - i\}$$

(b) Définition polyédrique des domaines d'itération de (a).



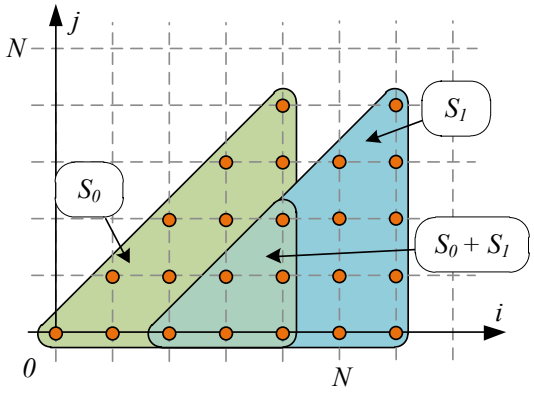
(c) Représentation graphique des domaines d'itération de (a). Les domaines d'itération sont séparés par l'ordre textuel (première dimension scalaire, cf. section 3.3.4).

$$\theta_{S_0}(0, i, j) = (i + j, j)$$
$$\theta_{S_1}(1, i, j) = (i + j + 2, j)$$

(d) Fonctions d'ordonnancement.

$$\theta_{S_0}(\mathcal{D}_{S_0}) = \{i, j \mid 0 \leq i < N \ \wedge \ 0 \leq j < i + 1\}$$
$$\theta_{S_1}(\mathcal{D}_{S_1}) = \{i, j \mid 2 \leq i < N + 2 \ \wedge \ 0 \leq j < i - 1\}$$

(e) Définition polyédrique des domaines d'itérations de (a) après application des transformations en (d).



(f) Représentation graphique des domaines d'itération de (a) transformés par θ_{S_0} et θ_{S_1}.

FIGURE 4.1 – Exemple de nid de boucles représenté dans le modèle polyédrique (a, b et c). Après avoir appliqué les transformations θ_{S_0} et θ_{S_1} (d), il faut générer du code parcourant le domaine transformé (e et f).

les polyèdres soit compilable ou synthétisable, il faut le décrire dans un langage impératif, voire directement dans un langage de description de matériel.

Ce problème est complètement indépendant des conditions de légalité de la transformation. Lors de l'étape de génération de contrôle, l'hypothèse est faite que la transformation est légale. Par ailleurs, il est tout à fait possible de générer du code pour une transformation donnée illégale (dans ce cas l'implémentation qui en résulte reste illégale).

Pour générer du contrôle correct à partir d'une représentation polyédrique, il faut que le domaine d'itération soit parcouru intégralement, dans l'ordre lexicographique, tout en préservant les valeurs des fonctions d'index. Dans la suite de cette section, différentes méthodes de génération de code sont présentées et elles sont illustrées sur l'exemple de la figure 4.1.

La figure 4.1 présente un Scop (*Static Control Programs* : Programmes à Contrôle Statique) composé d'une séquence de deux doubles boucles imbriquées (sous figures 4.1.(a), (b) et (c)). Pour ce nid de boucles est donnée une transformation polyédrique, composée de deux fonctions affines (une par instruction, en figure 4.1.(d)). Après application de la transformation, les domaines d'itération sont partiellement fusionnés.

Une approche simple consiste à parcourir le rectangle englobant (*bounding box*) de l'union des différents domaines d'itération transformés, c'est-à-dire le rectangle le plus petit qui contient cette union (voir le domaine entouré en pointillés sur la figure 4.2) : de simples boucles `for` itérant du minimum au maximum pour chaque indice suffisent alors à parcourir ce rectangle englobant. De cette manière tous les points de ces deux domaines d'itération transformés sont parcourus. Pour se restreindre aux points des domaines où un calcul est réalisé, il suffit d'ajouter des gardes, correspondant aux contraintes qui définissent les domaines d'itérations transformés.

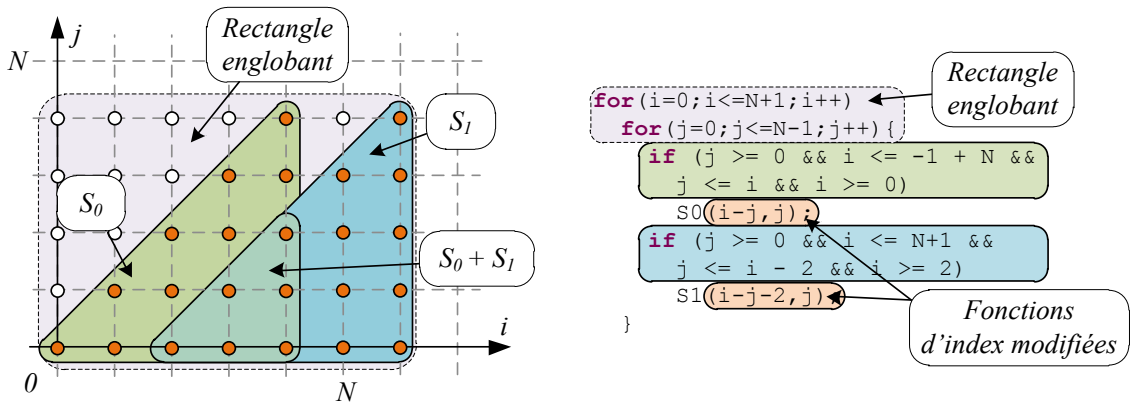


FIGURE 4.2 – Génération de code en suivant l'approche basique pour l'exemple de la figure 4.1. Les boucles `for` (entourées en pointillés) parcourent le rectangle englobant des domaines d'itération transformés, et les instructions sont gardées par leurs domaines d'itérations (entourés en traits pleins). Les fonctions d'index correspondent à l'inverse de la transformation, de sorte que les calculs soient effectués sur les valeurs originales.

Pour calculer le rectangle englobant, on peut procéder une dimension après l'autre, de la plus externe vers la plus interne, et projeter le domaine d'itération sur cette dimension. On obtient

alors les valeurs minimum et maximum pour chacune des dimensions et il suffit de générer des boucles `for` qui commencent au minimum et itèrent jusqu'au maximum. Ensuite il suffit d'insérer les gardes, sans oublier d'appliquer la fonction inverse dans les fonctions d'index (cette inversion est commune à toutes les méthodes de génération de code).

L'approche présentée précédemment est simple à mettre en œuvre, mais résulte en un nombre excessif de gardes, dont la plupart sont redondantes. De plus, le polyèdre parcouru est plus grand que le domaine d'itération, et plusieurs itérations ne correspondent à aucune opération (les points blancs de la figure 4.2). L'approche mise au point par Irigoin [58] réduit le nombre d'itérations qui n'exécutent aucune opération en parcourant l'enveloppe convexe au lieu du rectangle englobant (cf. figure 4.3). De plus, le nombre de gardes à évaluer est réduit en utilisant l'élimination de Fourier-Motzkin.

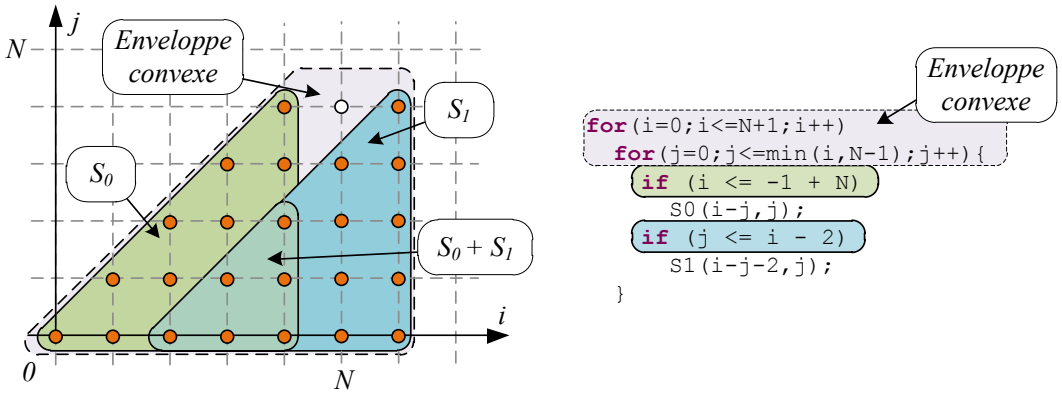


Figure 4.3 – Génération de code en suivant l'approche basique pour l'exemple de la figure 4.1, mais en ne parcourant que l'enveloppe convexe au lieu du rectangle englobant.

4.2.2 Génération de nids de boucles efficaces

L'approche mise au point par Irigoin est limitée à un seul polyèdre et génère beaucoup de contrôle redondant. De plus elle est restreinte aux transformations uniformes. Plusieurs travaux ont étendu cette approche. En particulier Le Verge et coll. [59] se basent sur la représentation duale pour éliminer les contraintes redondantes et supporter les transformations non-uniformes. Kelly et coll. [99] sont les premiers à proposer une approche, implémentée dans le calculateur Omega [97], pour générer du code pour plusieurs instructions lorsque le domaine à parcourir n'est pas convexe. Afin de minimiser le nombre de gardes présentes dans le code généré, et pour éliminer les itérations qui ne correspondent à aucune opération, Quilleré et coll. [60] ont proposé un nouvel algorithme de génération de code pour le parcours de polyèdres.

Pour parcourir les polyèdres de manière efficace, l'algorithme construit le nid de boucles dimension par dimension, en partant de la plus externe. À chaque dimension, les domaines sont projetés sur cette dimension, puis séparés en une liste de polyèdres disjoints. Chacune de ces disjonctions donne lieu à une boucle `for` qui sert alors de contexte lors de la génération de la

dimension suivante. En d'autres termes, la boucle à une profondeur n est générée étant données les valeurs des indices des boucles de profondeur $1..n-1$, chacune de ces boucles étant une partition du domaine d'itération original.

Pour obtenir des contraintes « étant données » les contraintes sur les boucles externes, l'algorithme se base sur l'opération polyédrique de simplification dans un contexte, l'opération *gist* [100]. La sémantique de cette opération est la suivante :

$$G = gist(D, C) \Leftrightarrow G \cap C = D$$

Étant donné deux polyèdres (un domaine D et un contexte C), l'opération *gist* retourne un domaine G tel que l'intersection de G avec C donne de nouveau D. L'objectif est de réduire le nombre de contraintes pour éviter la redondance des contraintes à chaque niveau de boucle et lors de l'insertion des contraintes au niveau des gardes. Cependant en pratique cette réduction n'est pas garantie.

Dans sa thèse, Bastoul [7] étend l'algorithme de Quilleré de manière à générer un code plus compact et performant, et fournit CloogG, une implémentation efficace de ce nouvel algorithme. En plus du support des domaines avec un pas supérieur à 1, cette extension génère du code moins complexe grâce à l'utilisation de transformations telles que le *loop peeling* ou le *loop splitting*[1]. Pour l'exemple de la figure 4.1, CloogG génère le code présenté en figure 4.4.

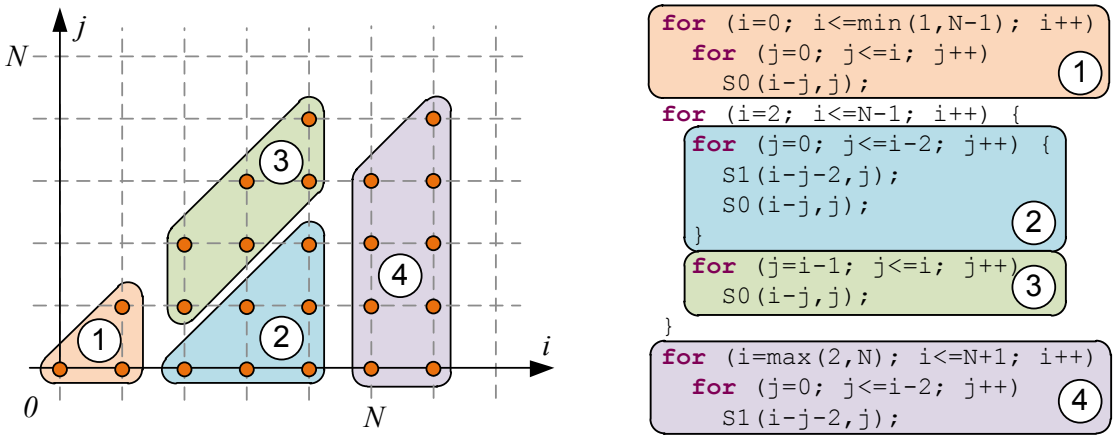


Figure 4.4 – Génération de code en suivant l'approche de CloogG pour l'exemple de la figure 4.1.

Cette technique de génération de code est très efficace dans le cadre de la compilation logicielle. En effet, lors de l'exécution d'un programme sur un processeur programmable, réduire le nombre de gardes permet de libérer le processeur pour laisser place aux opérations.

1. Les transformations de *loop peeling* et de *loop splitting* consistent à déplacer certaines itérations ou certaines partitions en dehors de la boucle, de manière à réduire le contrôle.

4.2.3 Génération d'une machine à états

Boulet et Feautrier [5] proposent une autre approche pour générer du code. Cette approche vise à générer un automate qui parcourt le polyèdre. L'approche se base sur la construction d'une fonction $next(\vec{x})$ qui, étant donné un vecteur d'itération courant $\vec{x}$, détermine son successeur immédiat suivant l'ordre lexicographique.

La construction de la fonction $next(\vec{x})$ est décrite dans l'algorithme 1. Les relations sont présentées suivant un formalisme à la Omega [97], aussi utilisé dans la bibliothèque Isl [64]. Le principe est de générer cette fonction de la dimension la plus interne vers la plus externe. Ainsi, étant donné un vecteur d'itération, on commence par déterminer l'ensemble de ses successeurs sur la dimension la plus interne. Le successeur immédiat est le plus petit, c'est-à-dire le minimum lexicographique paramétrique, de ces successeurs. Lorsqu'il n'y a plus de successeurs sur la dimension la plus interne (lorsque le vecteur d'itération donné est sur la borne supérieure), il faut alors chercher un successeur sur la dimension suivante. Pour réduire le nombre de contraintes impliquées dans la dimension suivante, on s'abstrait de la dimension courante en projetant le domaine d'itération sur les dimensions restantes ($projectOut(\mathcal{R}, 1)$ « élimine » la dernière dimension de $\mathcal{R}$). Lorsque toutes les dimensions ont été parcourues, on obtient un automate qui parcourt le domaine d'itération.

Algorithme 1 Construit la fonction $next_{\mathcal{D}}$ pour le parcours de polyèdres

procedure nextBouletCodeGen($\mathcal{D}$)
  $n \leftarrow dim(\mathcal{D})$
  $\mathcal{R} \leftarrow \mathcal{D}$
  for $p = n \rightarrow 1$ **do**
    $lexGT_p \leftarrow \{\vec{x} \rightarrow \vec{y} \mid \vec{x}_{[0..p-1]} = \vec{y}_{[0..p-1]} \wedge \vec{x}_{[p]} > \vec{y}_{[p]}\}$
    $succ_p \leftarrow \{\vec{x} \rightarrow \vec{y} \mid \vec{x} \in \mathcal{R} \wedge \vec{y} \in \mathcal{D}\} \cap lexGT_p$
    $next[p] = lexmin(succ_p)$
    $\mathcal{R} \leftarrow projectOut(\mathcal{R}, 1)$
  end for
  return *next*
end procedure

Dans l'exemple de la figure 4.1, la fonction *next* est calculée sur l'union des domaines transformés. Le successeur sur la dimension la plus interne est $next(i, j) = (i, j + 1)$ lorsque $j < i \wedge j < N - 1$, et le successeur sur la dimension i est $next(i, j) = (i + 1, 0)$ lorsque $i < N$. L'automate est initialisé avec le minimum lexicographique du domaine d'itération (en l'occurrence $(i = 0, j = 0)$). Les instructions sont finalement insérées dans le code avec les gardes correspondant à leur domaine, dont la majorité des contraintes sont éliminées grâce à l'opération *gist*. Ainsi, sachant que (i, j) est dans l'union des domaines d'itérations, S_0 est exécuté uniquement si $i < N$. Le code généré est présenté en figure 4.5.

Cette approche a été initialement développée pour la compilation logicielle, dans le contexte de la génération de code assembleur [5]. Les résultats expérimentaux ont montré que le code généré tend à être plus grand que celui généré par les techniques de la section précédente. Le

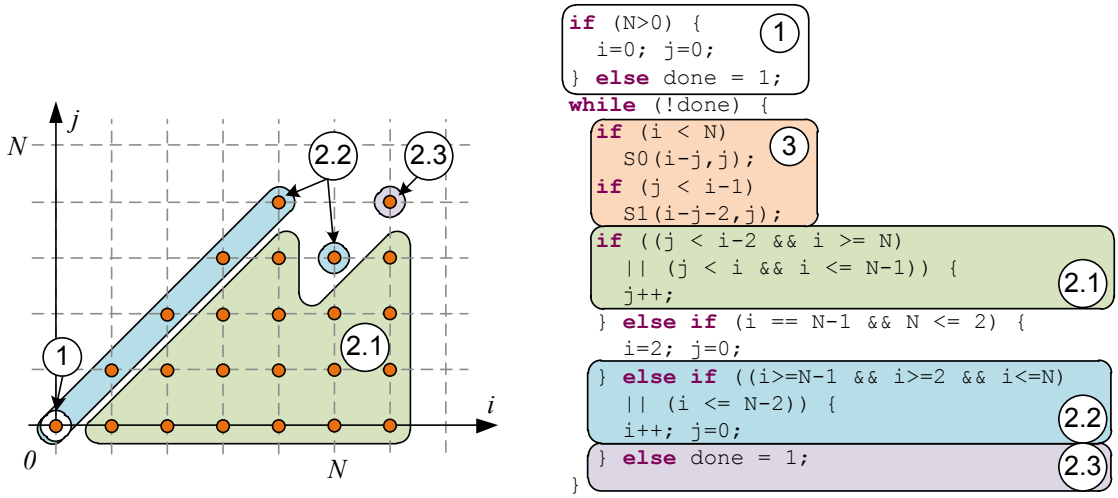


FIGURE 4.5 – Génération de code en suivant l'approche de BOULET et FEAUTRIER sur l'exemple de la figure 4.1. L'initialisation de l'automate est générée en entête (1), les transitions à la fin (2), et les commandes sont gardées par les sous-domaines d'itération respectifs (3).

temps d'exécution est légèrement plus faible dans les exemples sélectionnés grâce à la génération de code en assembleur directement, mais la complexité du code généré rend impossible un grand nombre d'optimisations dans le cadre de transformations source-à-source. L'approche semble néanmoins intéressante si l'on considère la synthèse de haut niveau comme post-traitement (*back-end*) d'un flot source-à-source : en effet la mise à plat du nid de boucles facilite la mise en œuvre du pipeline de nids de boucles, comme le chapitre suivant le montre.

4.3 Génération de contrôleur matériel

Les techniques présentées précédemment offrent des moyens pour parcourir des domaines d'itération suivant l'ordre lexicographique. En utilisant ces méthodes, il est ensuite possible de générer des contrôleurs matériels. Cette section présente quelques approches qui permettent d'y parvenir.

4.3.1 ClooGVHDL

Pour générer un contrôleur matériel qui parcourt des polyèdres, l'approche la plus simple consiste à utiliser un générateur de code dans un langage de haut niveau, et d'y ajouter un générateur de code VHDL (*Very-high-speed integrated circuits Hardware Description Language*). C'est l'approche suivie par DEVOS et coll. dans les outils JCCI et CLOOGVHDL [101, 102]. Après avoir trouvé des transformations améliorant la localité, JCCI génère un chemin de données, contrôlé par une machine à états.

Cette machine à états est générée par CLOOGVHDL, un back-end VHDL utilisé en aval de CLOOG (cf. section 4.2.2). Grâce à une option de CLOOG qui permet de réduire la taille du code

en introduisant des gardes (l'option `-f -1`), l'approche utilisée permet de générer un code VHDL parcourant les polyèdres à la manière de CodeGen, le générateur de Omega. L'automate parcourt alors l'enveloppe convexe des domaines d'itération (en visitant des itérations qui ne correspondent à aucune opération, cf. figure 4.3), et minimise le nombre de gardes autour des instructions à exécuter grâce à l'opération *gist*. Cette approche n'est pas efficace dans le contexte de la compilation logicielle : exécuter toutes les gardes demande beaucoup de temps processeur. Au contraire, ces gardes sont toutes évaluées en parallèle sur un accélérateur matériel, et l'impact ne concerne que le coût en ressources matérielles.

En utilisant cette technique, le parallélisme est exploité en dupliquant (via des transformations de déroulage de boucles) les instructions, lorsque l'analyse de dépendances le permet. Cela revient à vectoriser partiellement l'exécution de la boucle.

4.3.2 MMalpha

Les techniques de transformation de MMalpha visent à générer une architecture virtuelle, dans laquelle le nombre de processeurs est borné, et les communications sont limitées aux voisins proches (cf. section 3.5.1). Lors de la synthèse en utilisant MMalpha, le résultat est une architecture systolique pour laquelle il faut générer, en plus des chemins de données, un compteur global ainsi que la propagation des signaux de contrôle au sein du réseau.

Les premières versions de MMalpha ne considèrent que les ordonnancements monodimensionnels, pour lesquels le contrôleur parcourt un domaine sur une seule dimension (le temps). Au lieu de diffuser la valeur de l'instant courant à tous les processeurs du réseau, MMalpha analyse les contraintes d'activité des processeurs et les remplace par un pipeline de signaux de contrôle [103] (cf. figure 4.6).

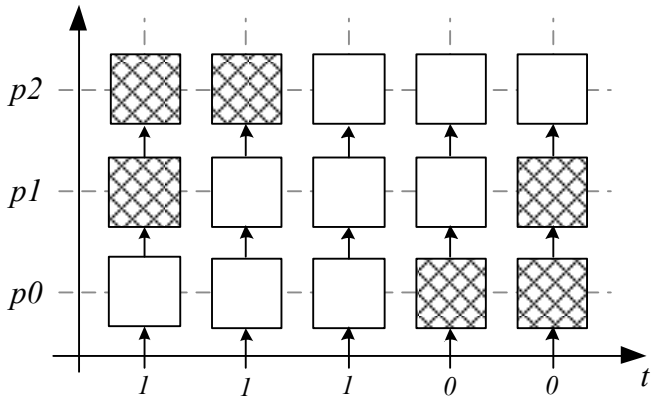


FIGURE 4.6 – Exemple de réseau systolique (3 processeurs $p0$ à $p2$). Au lieu d'envoyer le signal de contrôle à tous les processeurs, seul $p0$ reçoit un signal de contrôle qui est propagé à $p1$ et $p2$ à chaque cycle (dimension t), à la manière d'un pipeline.

Pour supporter les ordonnancements multidimensionnels, le contrôleur doit parcourir un domaine de temps logique multidimensionnel. Dans leurs travaux, GUILLOU et coll. [86] proposent d'utiliser la méthode de BOULET et FEAUTRIER pour générer un compteur énumérant les dif-

férentes étapes. L'approche n'a pas été implémentée dans le contexte de MMalpha. Elle l'a été plus récemment par YUKI dans les travaux de sa thèse [104] dans le contexte de la génération de code MPI (*Message Passing Interface*) pour le calcul haute performance, mais en utilisant la technique de génération implémentée dans CLOOG.

4.3.3 Réseau de processus polyédriques

Lors de la génération d'un réseau de processus polyédriques (cf. section 3.5.2), il faut tenir compte des domaines d'itération de chaque nœud, ainsi que des canaux de communication, en particulier ceux pour lesquels il faut réordonner les valeurs (*reordering channels*).

Pour parcourir les domaines d'itération des instructions, l'approche utilise la technique implémentée dans CLOOG. Le nid de boucles généré parcourt ainsi le domaine d'itération du processus dans l'ordre lexicographique. Afin de s'assurer que les lectures et les écritures dans les canaux sont bien effectuées avant et après l'opération, l'ordonnancement est modifié, en insérant une dimension scalaire (cf. section 3.3.4) au niveau le plus interne.

Pour réordonner les valeurs des canaux de communication, il faut soit générer un canal qui réordonne automatiquement les valeurs, soit utiliser une file (FIFO) associée à une mémoire locale qui se charge de réordonner les valeurs. Dans les deux cas, il faut déterminer une taille minimum pour le canal de manière à éviter les inter-blocages, qu'il est possible d'estimer grâce à l'expansion de BERNSTEIN [93, 94].

4.3.4 Transformations source-à-source et outils de HLS

Pour générer un contrôleur matériel qui parcourt le domaine d'itération, une autre approche consiste à générer du code C dans un premier temps, puis à utiliser les outils de HLS (*High-Level Synthesis* : Synthèse de Haut Niveau) sur ce code pour générer une description matérielle. C'est l'approche qu'utilise PLESCO [95] avec la suite d'outils Quartus fournis par Altera, et c'est celle utilisée pour les contributions présentées dans le chapitre 5. Cette approche permet d'exploiter toutes les optimisations bas niveau implémentées dans les outils de HLS, tout en profitant des transformations de haut niveau appliquées sur le code C par le compilateur source-à-source.

En utilisant l'approche implémentée dans CLOOG, l'outil de synthèse dispose de nids de boucles. En appliquant automatiquement des transformations de déroulage, de manière à vectoriser l'exécution de plusieurs itérations de la boucle, mais aussi de pipeliner l'exécution d'une boucles voire d'un nid de boucles, les outils de HLS sont ainsi capables d'exploiter le parallélisme. C'est le cas par exemple de Catapult-C et de AutoESL. Ces outils mettent en œuvre des analyses de dépendances afin d'autoriser ou non l'application de telles transformations. Ces analyses sont néanmoins très limitées dans leur application, et parfois incapables d'exploiter le parallélisme présent.

Lorsque le domaine d'itération est parcouru par une machine à états générée par la méthode de BOULET et FEAUTRIER, le code présente un seul niveau de boucle (cf. figure 4.5), dans lequel les itérateurs sont incrémentés suivant des gardes parfois complexes, et les instructions sont gardées. La structure de boucle, bien que représentant une machine à états qui s'implémente facilement en matériel, est aussi beaucoup plus difficile à analyser par l'outil car il n'y a plus de

structures de boucles « `for` ». Par conséquent l'analyse de dépendances des outils de synthèse est alors incapable de déterminer si les itérations dépendent les unes des autres. De plus, dérouler la boucle revient à dupliquer les commandes et les transitions de la machine à états, et la taille du code, et par conséquent celle du contrôleur matériel, augmente considérablement.

En utilisant la technique de Boulet et Feautrier, le parallélisme peut être mis en œuvre de deux façons. Avec un seul niveau de boucle, il est possible de pipeliner l'ensemble du nid de boucles, si les dépendances l'autorisent (cf. chapitre 5). L'autre approche consiste à dérouler partiellement la dimension la plus interne. De cette manière le corps de la boucle mise à plat contient plusieurs occurrences de la même instruction, qui seront vectorisées par l'outil de Hls.

4.4 Amélioration de la génération de contrôle

Dans cette thèse, nous avons choisi d'utiliser des transformations source-à-source appliquées en amont des outils de Hls (en particulier Catapult-C). Les techniques de génération de code ont été améliorées pour obtenir un contrôle plus performant après synthèse. La sous-section 4.4.1 explique la façon selon laquelle les deux techniques de parcours de polyèdres sont utilisées conjointement de manière à obtenir un tel code.

La technique de parcours de polyèdres de Boulet et Feautrier a été implémentée dans GeCoS grâce à la bibliothèque de manipulation de polyèdres Isl [64]. La sous-section 4.4.2 présente les optimisations apportées à l'approche initiale pour réduire la taille du code généré. La sous-section 4.4.3 rappelle les techniques mises en œuvre pour réduire la complexité. Enfin, la sous-section 4.4.4 montre qu'il est possible de déterminer avec exactitude le nombre de bits nécessaires pour encoder les valeurs des itérateurs et des expressions des gardes.

4.4.1 Structure des boucles

En plus de générer du code sous la forme d'une machine à états, la technique de Boulet et Feautrier permet de parcourir les polyèdres de manière exacte. Dans l'exemple de la figure 4.7, la version générée par Cloog (sous-figure (a)) utilise 1/3 des itérations pour tester la valeur de `k` sans exécuter `S`, lorsque `k` vaut 3. Avec la technique de Boulet et Feautrier (sous-figure (b)) la machine à états parcourt l'ensemble des valeurs de `i`, `j` et `k` pour lesquelles `S` est effectivement exécuté, et `k` n'atteint jamais la valeur 3.

Lors de l'implémentation matérielle de ce nids de boucles, et en supposant que chaque itération de la boucle est exécutée en 1 cycle, l'approche de Boulet et Feautrier permet ainsi de réduire le nombre de cycles de 30%. Cependant pour les nids de boucles complexes, avec plusieurs instructions, et une profondeur importante, le nombre et la complexité des transitions possibles impliquent un code plus conséquent, et un coût en matériel parfois prohibitif.

Pour réduire l'impact de la technique de Boulet et Feautrier sur les coûts en ressources matérielles, nous avons choisi de l'utiliser conjointement au générateur de code Cloog. En effet Cloog offre la possibilité de générer des boucles `for` pour les n dimensions externes. Dans la représentation intermédiaire générée par Cloog, les domaines des instructions pour les dimensions internes sont alors remplacés par le domaine restant à parcourir. Il suffit alors d'appliquer la

```
for (i=0; i<N; i++)
  for (j=0; j<M; j++)
    for (k=0; k<2; k++)
      S(i,j,k);
```

(a) Nid de boucles généré par Clooc.

```
i=0;j=0;k=0;
while (i<N) {
  S(i,j,k);
  if (k<1){
    k++;
  } else if (j<M-1) {
    k=0;j++;
  } else
    k=0;j=0;i++;
}
```

(b) Machine à états générée par la technique de Boulet et Feautrier.

```
for (i=0; i<N; i++) {
  j=0;k=0;
  while (j<M) {
    S(i,j,k);
    if (k<1)
      k++;
    else
      k=0;j++;
  }
}
```

(c) Approche hybride.

Figure 4.7 – Comparaison des différentes techniques de génération de code.

technique de Boulet et Feautrier. De cette manière il est possible de générer des boucles `for` pour les dimensions externes, et des machines à états pour les dimensions internes. Cette approche hybride permet de réduire le nombre de dimensions, et ainsi réduire le nombre et la complexité des transitions de la machine à états.

4.4.2 Représentation des domaines et relations polyédriques

La technique de génération de code de Boulet et Feautrier a été implémentée en utilisant la bibliothèque de manipulation de polyèdres Isl. L'approche consiste à générer une fonction qui associe, à chaque point du domaine d'itération, son successeur immédiat suivant l'ordre lexicographique, comme le construit l'algorithme 1. Cependant le résultat fourni par Isl en sortie de cet algorithme est une liste de relations (une par dimension) qui a deux inconvénients majeurs.

- Il existe une infinité de représentations sous forme de contraintes pour une relation polyédrique. Isl ne produit en résultat qu'une seule de ces représentations, qui n'est pas forcément la plus adaptée pour la génération de code.
- Isl produit des relations partitionnées, qui pourraient être représentées de manière plus compacte, afin de réduire le nombre de transitions de la machine à états. Un exemple d'un tel cas est présenté en figure 4.8. Cet exemple permet de se faire une idée des conséquences sur la génération de code. En effet, chaque partition de la fonction est matérialisée par une transition. Multiplier le nombre de partitions multiplie le nombre de transitions, et augmente le coût en ressources matérielles.

```
for (i=0; i<N; i++)
  S(i);
```

(a) Exemple de boucle.

$$next(i,j) = \{0 \to 1\} \cup \{i \to i+1 : 1 \leq i < N-1\}$$

(b) Fonction *next* produite par ISL.

$$next(i,j) = \{i \to i+1 : i < N-1\}$$

(c) Fonction optimisée pour la génération de code.

FIGURE 4.8 – La fonction *next* produite par ISL est parfois séparée en plusieurs morceaux (b). Pour la génération de code, il faut minimiser ce nombre de morceaux, de manière à minimiser le nombre de transitions de la machine à états (c).

Nous avons proposé deux techniques pour raffiner les contraintes définissant la fonction *next* avant d'appeler le générateur de code. Ce raffinement se fait deux étapes.

- Dans un premier temps, on regroupe les sous-relations pour lesquelles l'expression du successeur est la même sur le domaine d'application, de manière à minimiser le nombre de transitions de la machine à états. L'heuristique est simple, et consiste à prendre toutes les relations deux par deux, et d'appliquer un test d'équivalence polyédrique pour déterminer si leur fusion est légale.
- Dans un second temps, il faut simplifier les domaines d'application de chaque transition, de manière à réduire le nombre de contraintes qui définissent le domaine. ISL propose plusieurs fonctions pour réduire le nombre de contraintes [2].

4.4.3 Réduction de force

Dans le code généré par CLOOG ou via la technique de BOULET et FEAUTRIER, le contrôle est dominé par des bornes, gardes et fonctions d'accès aux tableaux exprimées par des fonctions affines (et quasi-affines) des indices et des paramètres. Ces expressions comportent des opérateurs coûteux à mettre en œuvre en matériel, comme la division ou le modulo. Des techniques existent pour éliminer ces opérateurs lorsqu'ils sont utilisés dans une boucle : l'objectif est de réduire le contrôle à de simples additions, comparaisons et décalages. On parle alors de réduction de force (*strength reduction*).

La transformation des multiplications est relativement simple à appliquer. Il suffit de remplacer les multiplications des indices de boucle par des registres incrémentés à chaque itération, comme présenté en section 2.3.1. Pour les divisions et les modulos, ZISSULESCU et coll. [105] proposent une approche basée sur cinq optimisations pour remplacer les divisions par des modulos, et ensuite simplifier les modulos au maximum. L'approche introduit beaucoup de conditionnelles et de variables.

L'avantage vient du fait que les expressions n'utilisent plus de divisions, et peu de modulos. Les conditionnelles, bien que nombreuses, peuvent toutes être exécutées simultanément en ma-

2. Notamment `isl_set_coalesce`, `isl_set_remove_redundancies`, `isl_set_detect_equalities`.

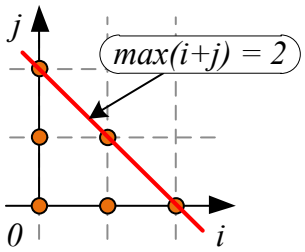
tériel, et rapidement du fait des opérateurs impliqués (additions et décalages principalement). En revanche les variables introduites sont matérialisées par des nouveaux registres. Leur mise en œuvre affecte peu le coût en ressources matérielles lors d'une implémentation sur FPGA (*Field Programmable Gate Array* : Réseau de Portes Programmables), car les éléments de base des FPGA comportent généralement un registre, mais ce coût est relativement élevé sur une technologie Asic (*Application-Specific Integrated Circuit* : Circuit Intégré Spécifique à une Application).

4.4.4 Analyse de taille de type

Après application de la réduction de force, il est possible de limiter la taille des registres en analysant l'intervalle des valeurs qu'ils peuvent prendre. Les outils de HLS peuvent estimer l'intervalle des valeurs pour les indices des boucles et les expressions des bornes, gardes et indices des tableaux [106]. Cependant à cause de la représentation sous forme de machine à états, et après les transformations de réduction de force, il leur est impossible d'évaluer précisément cet intervalle (cf. figure 4.9a).

```
for (i=0; i<3; i++)
  for (j=0; j<3-i; j++)
    S(i+j);
```

(a) Exemple de nid de boucles.



(b) Représentation graphique du domaine d'itération.

```
i=0;j=0;
while (i<3) {
  S(i+j);
  if (j<2-i)
    j++;
  else
    j=0;i++;
}
```

(c) Machine à états parcourant le domaine d'itération.

FIGURE 4.9 – Exemple de nid de boucles pour lequel il faut déterminer le nombre de bits nécessaires pour coder l'expression $i+j$. Une analyse simpliste se base sur l'intervalle de i et j (tous deux définis sur [0..2]) et en déduit que $i+j$ est défini sur [0..4], alors qu'une analyse basée sur le modèle polyédrique permet de déterminer la valeur maximale de $i+j$ sur le domaine d'itération, c'est-à-dire 2, et ainsi d'économiser un bit. De plus, l'analyse simpliste est beaucoup plus difficile à mettre en œuvre sur le code de la figure (c), généré par la méthode de BOULET et FEAUTRIER.

En conservant les informations du modèle polyédrique pendant l'étape de réduction de force, on peut associer à chaque expression, et à chaque registre qui stocke sa valeur, un domaine polyédrique sur lequel l'expression est évaluée. Les domaines paramétrés doivent alors être bornés, grâce au contexte fourni par le compilateur, ou à défaut par les types des variables définissant les paramètres. De cette manière, les domaines ont des bornes concrètes, et il est possible de déterminer un minimum et un maximum pour les expressions, et le nombre exact de bits nécessaires pour coder les valeurs de cet intervalle. Lors de la génération de code il est ensuite possible d'utiliser des types au bit près, tel que les `ac_int`, pour spécifier la largeur [107].

4.5 Conclusion

La technique de parcours de polyèdres utilisée dans CloоG a été améliorée au fil du temps pour satisfaire les problématiques liées à la compilation logicielle pour processeurs généralistes, en éliminant un maximum de gardes. Pour la synthèse de haut niveau, la taille de code conduit à des circuits coûteux en ressources matérielles. L'approche de Boulet et Feautrier permet la génération de machines à états, qui sont plus faciles à mettre en œuvre en matériel. Cette approche apporte aussi l'avantage de mettre à plat le nid de boucles, ce qui permet d'appliquer le pipeline sur plusieurs niveaux, lorsque la transformation est légale (cf. chapitre suivant).

En suivant les deux approches, les expressions présentes sont affines ou quasi-affines, et bénéficient des techniques de réduction de force, et il est possible de déterminer exactement l'intervalle des valeurs. De cette manière le coût en ressources matérielles diminue, et le contrôle s'évalue plus rapidement.

En termes d'optimisations, il est possible de prendre en compte le contexte de l'itération courante pour simplifier les contraintes (via l'opération *gist*) de la fonction *next*. Le code généré serait alors plus grand, mais porterait sur des contraintes plus simples, voire sur de simples booléens. L'approche est encore à l'étape de prototype.

Le générateur de code suivant l'approche de Boulet et Feautrier a été implémenté dans l'infrastructure de compilation GeCoS. Une évaluation de ses performances est présentée dans le chapitre 6.

Chapitre 5

Amélioration de l'applicabilité du pipeline de nid de boucles

5.1 Introduction

Lors de la conception d'un accélérateur matériel spécialisé, il est primordial d'exploiter pleinement le parallélisme afin d'obtenir un circuit performant. Pour atteindre un tel objectif, le pipeline de boucles (cf. section 2.3.3) est une transformation clé en HLS (*High-Level Synthesis* : Synthèse de Haut Niveau). Cependant lorsque le nombre d'itérations de la boucle pipelinée est faible comparée à la latence, les phases de remplissage et de vidage du pipeline dominent le temps d'exécution, et le taux d'utilisation matérielle chute.

Pour réduire l'impact de la latence du pipeline lorsque le nombre d'itérations de la boucle pipelinée est faible, il est possible d'entrelacer l'exécution de plusieurs instances de la boucle interne, de manière à obtenir un pipeline de nid de boucles.

L'objectif du travail présenté ici est d'améliorer l'applicabilité et l'efficacité du *pipeline de nid de boucles* dans les outils de Hls. Ce chapitre présente les contributions suivantes :

- Une analyse de la légalité du pipeline de nid de boucles en deux temps, qui consiste à approximer la légalité de manière prudente dans un premier temps, puis d'en faire une analyse plus précise dans un second temps, si nécessaire. Étant donnée la latence du pipeline, cette analyse permet de déterminer si appliquer la transformation de pipeline à un nid de boucles en altère la sémantique.
- Une technique de correction, appliquée lorsque le test de légalité échoue. Il est ainsi possible de corriger le pipeline en insérant des états d'attente, appelés *bulles*, de manière à respecter les dépendances de données.
- Pour appliquer la transformation de pipeline, le nid de boucles est mis à plat grâce aux techniques présentées dans le chapitre 4.

Ces contributions se basent sur la représentation des boucles dans le modèle polyédrique, présentée en chapitre 3. Grâce à cette représentation, notre méthode est applicable à une classe de programmes plus vaste, à savoir les Scop (*Static Control Programs* : Programmes à Contrôle

Statique) (cf. section 3.3.4) que les travaux précédents [108, 109, 110, 111].

Cette approche a été mise en œuvre dans un compilateur source-à-source en utilisant des outils d'analyse des transformations mis au point par la communauté scientifique. Son applicabilité a été validée sur un ensemble de noyaux de calcul représentatif de la HLS.

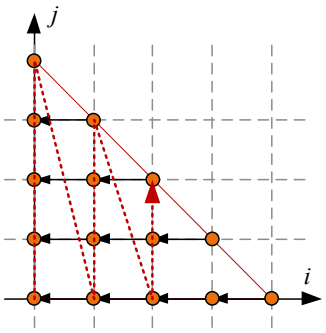
La sous-section 5.2 rappelle l'intérêt du pipeline de boucles en HLS, ainsi que les difficultés techniques que présente son application. L'état de l'art en la matière est présenté en sous-section 5.3. L'analyse de légalité en deux temps est ensuite présentée dans la sous-section 5.4. Pour finir la sous-section 5.5 montre comment il est possible de corriger un pipeline *a priori* illégal en insérant des bulles.

5.2 Pipeline de nid de boucles en HLS

L'objectif de cette section est de présenter et de motiver le problème traité, c'est-à-dire le *pipeline de nid de boucles*. Pour faciliter la compréhension, l'exemple présenté en figure 5.1a sera utilisé tout au long de cette section. Cet exemple est un extrait simplifié de l'algorithme de décomposition QR, et consiste en une double boucle imbriquée parcourant un domaine d'itération triangulaire.

```
/* code source original */
for(int i=0;i<N;i++) {
  for(int j=0;j<N-i;j++){
S0:    Y[j] = func(Y[j]);
  }
}
```

(a) Extrait simplifié de la décomposition QR.



(b) Représentation graphique de (a).

$$\mathcal{D}_{S_0} = \{i, j \mid 0 \leq i < N \wedge 0 \leq j < N - i\}$$

(c) Définition polyédrique du domaine d'itération de (a).

$$d : (i, j) \rightarrow (i, j - 1) : (i, j) \in D_d$$
$$\text{avec } D_d = \{i, j \mid i > 0\} \cap \mathcal{D}_{S_0}$$

(d) Définition polyédrique de la dépendance de données de (a).

FIGURE 5.1 – Extrait de la décomposition QR (a), et représentation graphique de son domaine d'itération et de ses dépendances de données sur le tableau `Y` (flèches noires) quand $N = 5$ (b). La flèche rouge en pointillés indique l'ordre d'exécution des itérations du nid de boucles. Le domaine d'itération ainsi que la seule dépendance sont décrits dans le modèle polyédrique en (c) et (d).

5.2.1 Pipeline de boucles

Le *pipeline de boucle* consiste à exécuter le corps d'une boucle en utilisant plusieurs composants matériels successifs appelés *étages de pipeline* (*pipeline stages*). L'efficacité de cette transformation vient du fait que plusieurs itérations de la boucle peuvent être exécutées simultanément dans les différents étages. Pour appliquer une telle transformation, il faut s'assurer que les itérations successives sont indépendantes. Le pipeline de boucle est caractérisé principalement par deux paramètres :

- *L'intervalle d'initiation* (noté Φ par la suite) est le nombre de cycles d'horloge séparant l'exécution de deux itérations successives.
- La *latence du pipeline* (notée Δ) représente le nombre de cycles d'horloge requis pour exécuter complètement une itération de la boucle. Lorsque $\Phi = 1$ cette latence correspond au nombre d'étages du pipeline.

Dans l'exemple de la figure 5.2, la boucle interne (sur l'itérateur j) ne porte aucune dépendance. Par conséquent, il est possible de pipeliner cette boucle en entrelaçant l'exécution d'itérations successives. La figure 5.2 représente l'exécution pipelinée de l'exemple de la figure 5.1, avec un intervalle d'initiation $\Phi = 1$ et une latence $\Delta = 4$.

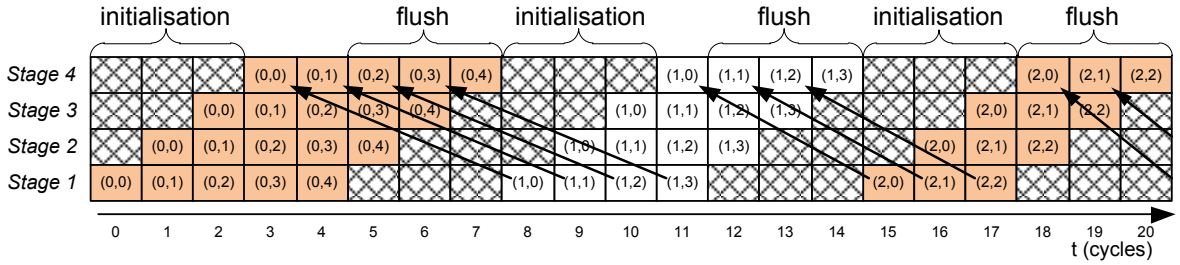


FIGURE 5.2 – Représentation de l'exécution pipelinée de l'extrait simplifié de la décomposition QR présenté en figure 5.1a, lorsque $N = 5$, $\Phi = 1$, et la latence du pipeline $\Delta = 4$. Les flèches représentent les dépendances entre les opérations.

En pratique, la valeur de l'intervalle d'initiation Φ est contrainte par deux facteurs :

- la présence de dépendances portées par la boucle, qui empêchent les exécutions d'itérations successives d'être entrelacées ;
- les ressources matérielles disponibles, puisqu'il faut qu'à chaque opération soit allouée une unité fonctionnelle (en étage) pour que l'exécution pipelinée soit possible.

Dans cet exemple, entre deux itérations successives de la boucle externe i, il y a une phase de vidage du pipeline (*flush*), qui est nécessaire pour s'assurer qu'aucune dépendance n'est violée. Les techniques présentées dans ce chapitre visent à éviter ces vidages, avec pour objectif une exécution plus efficace des implémentations pipelinées.

Parce qu'il permet d'obtenir des débits maximisés et améliore l'utilisation du matériel, le pipeline de boucle est une transformation clé pour la HLS. Par ailleurs, les concepteurs cherchent

la plupart du temps à obtenir le meilleur de leurs implémentations, en initiant une nouvelle itération à chaque cycle, c'est-à-dire en choisissant $\Phi = 1$.

Cependant les performances qu'il est possible d'obtenir en appliquant le pipeline sont souvent limitées par les analyses de dépendances de données imprécises des outils de HLS, ce qui les rend incapables de détecter s'il est possible d'appliquer le pipeline de boucle, en particulier lorsque la boucle implique des accès mémoires complexes. Afin de contourner ces limitations, les outils de HLS permettent d'outrepasser l'analyse de dépendances grâce à des directives de compilation (en général sous la forme de `#pragma`). Spécifiées par l'utilisateur, ces directives forcent l'outil à ignorer certains accès mémoires lors de l'analyse de dépendances. Lorsqu'elles sont utilisées à tort, par exemple pour forcer l'outil à ignorer des dépendances qui s'avèrent être effectives, ces directives conduisent à un pipeline illégal. C'est donc le concepteur qui a la charge de décider si la transformation est légale ou non.

5.2.2 Le surcoût de la latence du pipeline

Lorsque le nombre d'itérations est important, l'impact de la latence du pipeline sur les performances est négligeable, et l'utilisation du matériel est très proche de 100%. Mais dès que le nombre d'itérations devient comparable à la latence du pipeline Δ, on observe une baisse significative des performances. Cette baisse est due aux phases de vidage qui dominent l'exécution. C'est le cas de l'exemple de la figure 5.2. Lors que $N = 5$ et $\Delta = 4$, le taux d'utilisation du matériel est de 50% seulement.

Pourtant, sur ce même exemple, un ordonnancement dans lequel les itérations successives de la boucle externe i seraient entrelacées permettrait d'atteindre un taux d'utilisation proche de 100%.

5.2.3 Pipeline de nid de boucles : principe

Initialement proposé par DOSHI et coll. [108], le *pipeline de nid de boucles* vise à améliorer l'exécution d'une boucle pipelinée. Comme cela est fait dans d'autres travaux et outils [112, 52, 50], l'implémentation présentée est appliquée en deux étapes :

- dans un premier temps le nid de boucles est réécrit de manière à obtenir une seule boucle. Cette transformation est appelée *loop coalescing*[1] (mise à plat de boucles), comme présenté précédemment dans la chapitre 4 ;
- dans un deuxième temps, le pipeline est appliqué à la boucle mise à plat.

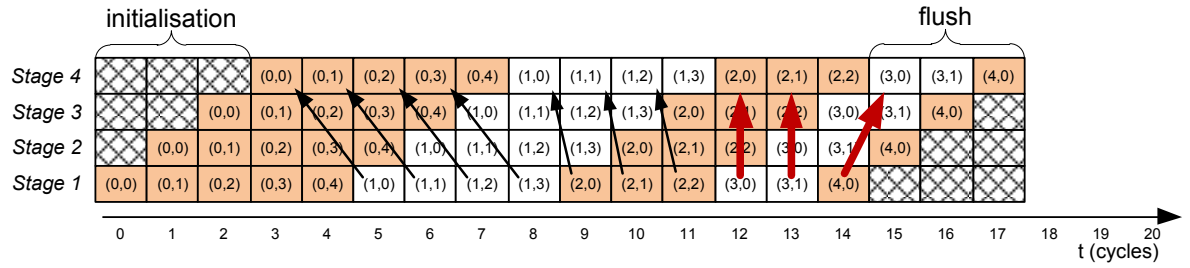
Jusqu'à présent, le problème du pipeline de nid de boucles a été traité uniquement dans le cas de nids de boucles parfaitement imbriqués, avec des bornes constantes et des dépendances de données uniformes – un sous ensemble très restrictif des nids de boucles – ou avec une analyse de dépendances relativement imprécise. Ces limitations peuvent paraître trop prudentes, mais il s'avère qu'implémenter le pipeline de nid de boucles (et en particulier assurer que les dépendances de données sont respectées) est loin d'être évident et requiert une attention particulière, comme nous allons le montrer dans l'exemple ci-dessous.

1. Aussi appelée *loop flattening*.

Par exemple, la figure 5.3a montre une version mise à plat du nid de boucles de la figure 5.1. Dans cet exemple, parce que les accès tableau de cette version mise à plat sont plus difficiles à analyser (ils ne dépendent plus d'indices de boucle comme dans la figure 5.1), il est tentant de contourner l'analyse de dépendances grâce à une directive (`#pragma ignore_mem_depcy Y`) pour forcer l'outil de HLS à appliquer le pipeline de boucle. Sans une telle directive, l'analyse de dépendance conservative interdit l'application du pipeline.

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
S0: Y[j] = func(Y[j]);
    if(j < N - i - 1)
       j++;
    else
       j=0,i++;
}
```

(a) Nid de boucles de la figure 5.1a mis à plat.



(b) Représentation de l'exécution pipelinée de (a).

FIGURE 5.3 – Illustration d'un pipeline de nid de boucles illégal. L'exemple est une version mise à plat de l'extrait de la décomposition QR de la figure 5.1, pour $N = 5$, $\Phi = 1$ et $\Delta = 4$. Les flèches épaisses en rouge montrent les dépendances violées.

Alors que l'ordonnancement semble correct, certaines dépendances RAW (*Read After Write* : lecture après écriture) sont violées dès que $i \geq 3$, comme le montre la figure 5.3b. En effet pour ces valeurs de i les dépendances entre deux itérations successives de la boucle externe i empêchent les pipelines de la boucle interne j de se chevaucher. Par exemple, l'accès en lecture sur `Y[0]` à $(i = 3, j = 0)$ est exécuté à $t = 12$, avant que `Y[0]` ne soit mis à jour par l'opération d'écriture à $(i = 2, j = 0)$, qui est aussi exécuté à $t = 12$ sur le dernier étage du pipeline.

Parmi les outils de HLS commerciaux et académiques que nous avons testé, seuls deux permettent d'appliquer automatiquement le pipeline de nids de boucles : Catapult-C de Mentor

Graphics [50], et AutoEsl de Xilinx [2] [52]. Cependant leurs implémentations comportent quelques défauts, et les outils peuvent générer des ordonnancements pipelinés illégaux lorsque les boucles n'ont pas de bornes constantes. Même en l'absence de directives de compilation pour ignorer les dépendances, ils semblent échouer pour les mêmes raisons que celles illustrées en figure 5.3, c'est-à-dire que leur analyse suppose que les dépendances portées sur `Y` par la boucle externe ne sont jamais violées [3].

5.3 Approches existantes

Le pipeline de boucles et de nids de boucles a été largement étudié, dans le cadre de la Hls et de la compilation pour processeurs programmables. Cette section présente l'originalité de notre approche face à l'état de l'art du domaine. La sous-section 5.3.1 décrit différentes approches pour appliquer le pipeline de boucles dans le contexte de la Hls. La sous-section 5.3.2 présente les travaux réalisés dans l'optique d'améliorer l'applicabilité et l'efficacité du pipeline de nids de boucles. Enfin la sous-section 5.3.3 confronte les techniques de corrections existantes à notre proposition d'insertion de bulles.

5.3.1 Pipeline de boucle en synthèse de matériel

L'extraction du parallélisme à grain fin, prérequis pour appliquer le pipeline de boucles, a été traitée dans des travaux antérieurs dans le contexte de la synthèse d'architectures systoliques. Parmi ces travaux, Derrien et coll. [110] proposent de partitionner le domaine d'itération pour chercher à combiner le parallélisme au niveau opération (pipeline) et au niveau de la boucle. Un problème similaire est traité par Teich et coll. [111], qui proposent de combiner le *modulo scheduling* avec des techniques de parallélisation de boucles. La principale limitation de ces contributions est qu'elles ne permettent que les ordonnancements monodimensionnels, ce qui limite fortement leur applicabilité (cf. section 3.4.2).

Alias et coll. [113] traitent le problème de la génération de pipeline de nid de boucles efficaces pour la génération d'accélérateurs matériels utilisant des chemins de données à virgule flottante spécialisés. Leur approche (elle aussi basée sur le modèle polyédrique) consiste à trouver un hyperplan parallèle pour le nid de boucles, et à dériver un pavage de manière à ce qu'appliquer un pipeline d'une latence Δ soit légal. Ce travail ne concerne que les nids de boucles parfaitement imbriqués, et nécessite que les tuiles incomplètes soient parcourues comme des tuiles complètes, ce qui implique une perte d'efficacité. De plus, les auteurs se restreignent aux dépendances uniformes, de manière à garantir que la distance de réutilisation est toujours constante pour une tuile donnée. En comparaison, les approches présentées dans notre travail prennent en compte les nids de boucles non parfaitement imbriqués avec des dépendances affines, et permettent de déterminer une correction plus précise que de remplir les tuiles incomplètes (en termes de nombre d'états d'attente).

2. AutoEsl nécessite une mise à plat explicite du nid de boucle, la transformation de pipeline n'étant applicable que sur un seul niveau de boucle.

3. Les expérimentations avec la dernière version de Catapult-C semblent montrer que ce problème a été corrigé.

La suite d'outils Compaan/Laura [92] aborde le problème différemment, en ne cherchant pas un ordonnancement global pour le nid de boucles. Au contraire, chaque instruction du programme se voit associer un processus. Les dépendances entre les opérations sont matérialisées par des canaux de communication de type FIFO, suivant la sémantique des PPN (*Polyhedral Process Network* : Réseau de Processus Polyédriques) [90] (cf. section 3.5). La causalité des ordonnancements est assurée par la disponibilité des données sur la sortie de ces canaux, par conséquent il n'est pas nécessaire de prendre en compte la latence d'exécution des instructions dans l'ordonnancement des processus [114]. Cependant cette approche conduit à un surcoût matériel important, car chaque instruction nécessite son propre contrôleur matériel, ainsi que des mémoires complexes pour réordonner les données des canaux de communication.

5.3.2 Pipeline de nid de boucles

Le pipeline logiciel est une optimisation clé pour exploiter le parallélisme instruction disponible dans la plupart des noyaux de calcul intensif. Depuis son introduction par LAM et coll. [115], beaucoup de travaux ont traité ce sujet.

RONG et coll. [109] étudient le problème du pipeline logiciel pour les boucles qui ne sont pas les plus internes, en appliquant une permutation de boucles et en fusionnant le vidage avec l'initialisation pour réduire l'impact de la latence. Ces travaux sont similaires à ceux présentés dans ce chapitre – bien qu'ils ne ciblent pas la HLS – mais ils se limitent à un ensemble de boucles plus restreint (bornes constantes et domaines rectangulaires), et n'utilisent pas l'analyse de dépendances exacte. De plus la permutation de boucles implique une modification de l'ordonnancement des opérations.

FELLAHI et coll. [116] traitent le problème de la fusion du prologue et de l'épilogue (l'initialisation et le vidage) lors de séquences de boucles pipelinées. Leur recherche est aussi motivée par le fait que le surcoût impliqué par la latence du pipeline devient une limitation dans de nombreux algorithmes multimédia embarqués dans lesquels le nombre d'itérations est petit. L'approche présentée dans cette recherche traite le problème au niveau du code machine VLIW (*Very Long Instruction Word* : Mot d'Instruction Très Long) et ne s'intéresse pas à la mise à plat des boucles, au contraire des approches présentées dans ce chapitre, qui sont présentées dans un contexte de transformation source-à-source pour la HLS.

En utilisant les spécificités des jeux d'instruction EPIC [117] disponibles dans les processeurs Itanium [118], MUTHUKUMAR et coll. [108] proposent de contrôler le vidage du pipeline. Leur approche a comme objectif de compter le nombre d'itérations séparant la définition d'une valeur de son utilisation. Cependant, ici encore, leur approche n'est applicable que pour les boucles avec des bornes constantes et des dépendances uniformes. L'originalité de leur travail est de proposer un mécanisme de correction, qui vide partiellement le pipeline lorsque des dépendances sont violées. Cependant le nombre d'états d'attente impliqués par ce vidage partiel est le même pour toutes les itérations de la boucle externe. À la différence de l'approche proposée dans la suite, il en résulte un nombre d'états d'attente surestimé, car l'analyse n'est pas aussi précise, mais aussi un contrôle plus simple comportant moins de gardes.

5.3.3 Correction de transformations de boucles illégales

L'idée de corriger un ordonnancement dans une étape de post-transformation, comme nous le proposons dans la suite, n'est pas nouvelle, et a été introduite par Bastoul et coll. [119]. L'approche consiste à chercher, dans un premier temps, une combinaison de transformations de boucles intéressante (légale ou non), puis de corriger les éventuelles combinaisons illégales par des décalages. Ce résultat a été étendu par Vasilache et coll. [120], qui considère alors un espace de corrections plus vaste. Les travaux présentés dans ce chapitre diffèrent dans le sens où ils ne tentent pas de modifier l'ordonnancement des opérations, mais d'ajouter des états artificiels pour rendre le pipeline de nid de boucles légal.

5.4 Analyse de la légalité

Cette section aborde le problème de la vérification de la légalité d'une transformation de pipeline de nid de boucles donné vis-à-vis des dépendances de données. La sous-section 5.4.1 décrit le modèle de pipeline et présente les conditions de légalité. Ces conditions peuvent être vérifiées en calculant la distance de réutilisation des dépendances (sous-section 5.4.2). Cependant cette technique ne donne qu'une approximation prudente. En calculant le successeur d'un vecteur d'itération (sous-section 5.4.3), il est possible de construire l'ensemble des dépendances violées par l'application du pipeline de nid de boucles (sous-section 5.4.4). L'algorithme complet est donné en sous-section 5.4.5.

5.4.1 Modèle de pipeline et condition de légalité

Le modèle de pipeline utilisé par la suite et illustré par la figure 5.4 est le suivant. Soit Δ le nombre d'étages du pipeline, c'est-à-dire sa latence, constante et donnée par l'outil de Hls. Dans le modèle de pipeline considéré, toutes les lectures sont exécutées dans le premier étage du pipeline, et toutes les écritures dans le dernier étage. Ces hypothèses ne sont pas essentielles, mais simplifient l'explication. Sauf mention contraire, et pour les raisons précisées en section 5.2, l'intervalle d'initiation Φ vaut 1.

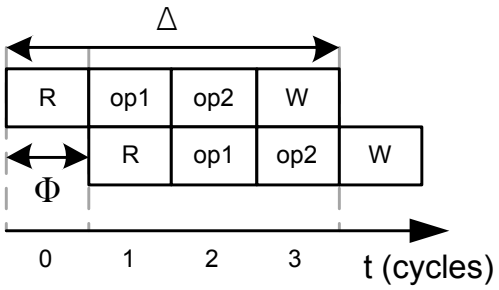


Figure 5.4 – Modèle de pipeline. Δ représente le nombre de cycles (la latence) entre les lectures au premier étage du pipeline et les écritures au dernier étage.

La *distance de réutilisation* d'une dépendance représente le nombre de points du domaine d'itérations qui séparent une source $\vec{x}$ (production d'une valeur) de sa référence $\vec{y}$ (utilisation), selon l'ordre lexicographique. Puisque l'exécution de la boucle suit l'ordre lexicographique, deux itérations consécutives sont séparées par un cycle. Par conséquent, le nombre de cycles séparant la source $\vec{x}$ de la référence $\vec{y}$ correspond à leur distance de réutilisation. D'un autre côté, la valeur produite par la source $\vec{x}$ est disponible Δ cycles après son initiation. Par conséquent le pipeline de nid de boucles n'introduit pas de violation de dépendances de données tant que la distance de réutilisation (en nombre de points) entre la production d'une valeur (à l'itération $\vec{x}$, la source) et son utilisation (à l'itération $\vec{y}$, la référence) et supérieure ou égale à Δ.

Cette condition est trivialement vérifiée dans le cas particulier où les boucles à pipeliner ne portent pas de dépendances, c'est-à-dire quand les boucles sont parallèles. Ce cas se produit, par exemple, quand les dépendances du nid de boucles sont portées par les boucles externes et que seules les boucles internes sont pipelinées (cf. figure 5.5).

```
for (k=0; k<P; k++)
  for (i=0; i<M; i++)
    for (j=0; j<N; j++)
      if (k==0)
        C[i][j]=A[i][k]*B[k][j];
      else
        C[i][j]=C[i][j]+A[i][k]*B[k][j];
```

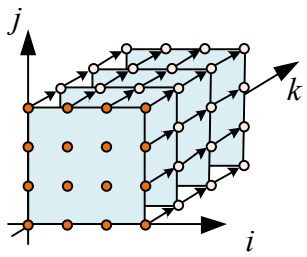


FIGURE 5.5 – Exemple de produit matriciel dans lequel l'accumulation est portée par la boucle la plus externe k. Les deux boucles internes i et j sont alors parallèles, et il est possible d'y appliquer le pipeline de nid de boucles.

Pour appliquer le pipeline à des boucles qui portent des dépendances, voire au nid de boucles complet (comme sur l'exemple en figure 5.3), une analyse plus approfondie est nécessaire. C'est ce que fournit le test de légalité présenté dans les sous-sections suivantes.

5.4.2 Vérifier la légalité en estimant la distance de réutilisation

Calculer la distance de réutilisation entre une source et une référence revient à compter le nombre de points minimum qui les sépare lexicographiquement dans le domaine d'itération (cf. exemple en figure 5.6).

Notons $src(d)$ l'ensemble des vecteurs d'itérations qui sont source d'une valeur (c'est-à-dire l'image de la fonction d). On peut ainsi construire la relation polyédrique R qui, étant donnée une itération source donnée $\vec{x}$, lui associe l'ensemble des vecteurs d'itération dans le domaine d'itération $\mathcal{D}$ qui sont lexicographiquement compris entre $\vec{x}$ et la référence la plus proche lexicographiquement ($d^{-1}(\vec{x})$ représente l'ensemble des références accédant à la valeur produite par la source $\vec{x}$). On note alors :

$$R(\vec{x}) = \{\vec{x} \leftrightarrow \vec{z} \mid \vec{x} \in src(d) \wedge \vec{x} \prec \vec{z} \prec d^{-1}(\vec{x}) \wedge \vec{z} \in \mathcal{D}\} \tag{5.1}$$

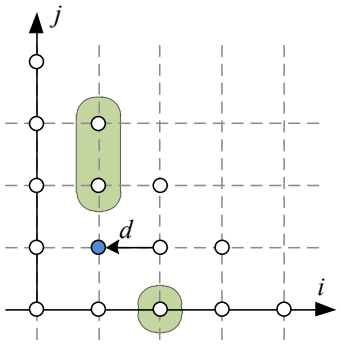


FIGURE 5.6 – Instance de la dépendance d lorsque $(i, j) = (2, 1)$. Il y a trois points lexicographiquement compris entre la source en bleu et la référence. La distance de réutilisation pour cette instance vaut donc 3.

$R(\vec{x})$ est un polyèdre paramétré qui peut être construit grâce à une bibliothèque de manipulation de polyèdres tel que ISL [64] ou PolyLib [96]. Étant donné $R(\vec{x})$, le nombre de points entre la source $\vec{x}$ et sa référence la plus proche est un pseudo-polynôme paramétrique $P(\vec{x})$ dépendant de $\vec{x}$ et des paramètres du domaine d'itération (cf. [92]). $P(\vec{x})$ peut être obtenu en utilisant la bibliothèque Barvinok [93]. Enfin, la valeur minimum de $P(\vec{x})$ sur le domaine d'itération peut être approchée grâce à l'expansion de BERNSTEIN [94]. Cependant le résultat de l'expansion de BERNSTEIN ne donne pas toujours le minimum, mais un minorant de ce minimum. Si ce minorant est supérieur à $\Delta - 1$, il y a toujours au moins $\Delta - 1$ cycles qui séparent la source de la référence, et appliquer le pipeline est donc légal.

Exemple

■ En partant de la dépendance d définie en figure 5.1d, il est possible de calculer l'inverse de d comme suit :

$$d^{-1} : (i, j) \to (i, j + 1) : (i, j) \in \mathcal{D}_{S_0} \wedge (i, j + 1) \in \mathcal{D}_{S_0}$$

D'après l'équation (5.1) :

$$R(i, j) = \left\{ (i', j') \;\middle|\; \begin{array}{l} (i, j) \in src(d) \wedge (i', j') \in \mathcal{D}_{S_0} \\ \wedge\ (i, j) \prec (i', j') \prec d^{-1}(i, j) \end{array} \right\}$$

Après avoir simplifié la relation polyédrique, on obtient alors :

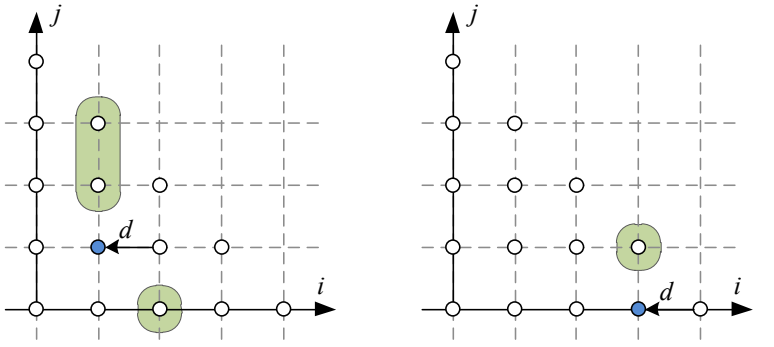
$$R(i, j) = \left\{ (i', j') \;\middle|\; \begin{array}{l} (0 \leq i \wedge i' = i \ \wedge\ 0 \leq j < j' < N - i) \vee \\ (0 \leq i \wedge i' = i + 1 \ \wedge\ 0 \leq j' < j < N - i - 1) \end{array} \right\}$$

Pour $N = 5$, $R(1, 1)$ est représentée par l'ensemble de points entourés sur la figure 5.7a, c'est-à-dire $\{(i' = 1 \wedge 2 \leq j' \leq 3) \vee (i' = 2 \wedge j' = 0)\}$. Le nombre de points qui séparent une source de

sa référence la plus proche dans $\mathcal{D}_{S_0}$, selon la dépendance d, est donc :

$$P(i,j) = card(R(i,j)) = N - i - 1$$

c'est-à-dire 3 lorsque $(i,j) = (1,1)$ et $N = 5$.



(a) $R(1,1)$ lorsque $N = 5$. (b) $R(N-1,0)$ pour tout $N > 1$.

FIGURE 5.7 – Représentation graphique de l'image de R (points entourés) sur le domaine de l'exemple en figure 5.1, étant donné la source de d.

En utilisant l'expansion de BERNSTEIN, on peut alors calculer une borne inférieure pour $P(i,j)$ sur $\mathcal{D}_{S_0}$, pour toutes les valeurs possibles de N. Comme illustré par la figure 5.7b, le minimum est 1, et il est atteint quand $(i,j) = (N-1,0)$. Par conséquent, appliquer le pipeline de nid de boucles avec $\Delta = 4$ sur ce nid de boucles est illégal. ■

La méthode décrite ci-dessus est rapide à calculer, mais ne donne pas toujours une estimation précise de la borne inférieure (la borne est pessimiste). De ce fait, cette analyse peut conclure à tort qu'appliquer le pipeline de nid de boucles est illégal. De plus, le résultat de cette analyse ne fournit pas de moyen pour corriger le pipeline si les conditions de légalités ne sont pas réunies.

5.4.3 Construction de la fonction $next^{\Delta}_{\mathcal{D}}(\vec{x})$

Pour contourner les limitations de la méthode précédente, il est possible de construire une fonction $next^{\Delta}_{\mathcal{D}}(\vec{x})$ qui donne, étant donné un vecteur d'itération source $\vec{x}$, son successeur après Δ itérations dans le domaine d'itération $\mathcal{D}$. Étant donnée $\vec{x}$, source d'une dépendance d, on peut alors vérifier que toutes les références accédant à la valeur produite sont bien exécutées au moins Δ itérations après. L'ensemble des références $\vec{y}$ accédant à la valeur produite par $\vec{x}$ correspond à l'inverse de la fonction de dépendance, $\vec{y} \in d^{-1}(\vec{x})$. Il faut ensuite s'assurer que ces vecteurs d'itération sont bien lexicographiquement supérieurs au vecteur d'itération Δ itérations après $\vec{x}$, $\vec{y} \succeq next^{\Delta}_{\mathcal{D}}(\vec{x})$. On est alors sûr que la valeur produite à l'itération $\vec{x}$ est bien utilisée au moins Δ itérations plus tard.

Pour construire la fonction $next^{\Delta}_{\mathcal{D}}(\vec{x})$, nous utilisons un résultat de BOULET et FEAUTRIER [5] dans leur technique de génération de code (cf. section 4.2.3). Par convention, $next_{\mathcal{D}}(\vec{x}) = \perp$

lorsque le vecteur $\vec{x}$ n'a pas de successeur dans $\mathcal{D}$, et $next_{\mathcal{D}}(\bot) = \bot$.

L'algorithme 2 rappelle la méthode de BOULET et FEAUTRIER, dans lequel $dim(\mathcal{D})$ représente le nombre de dimensions de $\mathcal{D}$, $lexmin(succ_i)$ (donné par ISL) fournit le minimum lexicographique de la relation $succ_i$, et $domain(next_{\mathcal{D}})$ représente le sous-domaine d'itération sur lequel $next_{\mathcal{D}}$ est applicable. Cet algorithme ne construit la fonction $next_{\mathcal{D}}$ que pour les h boucles les plus internes. Ce paramètre permet d'éviter des calculs inutiles, lorsque seules les h boucles les plus internes sont pipelinées.

Algorithme 2 Construit les fonctions $next_{\mathcal{D}}$ et $next_{\mathcal{D}}^{\Delta}$

Require: $1 \leq h \leq dim(\mathcal{D})$
  procedure NEXTBOULET($\mathcal{D}$, h)
    $n \leftarrow dim(\mathcal{D})$
    $next_{\mathcal{D}} \leftarrow \emptyset$
    $\mathcal{R} \leftarrow \mathcal{D}$
    for $p = n \rightarrow (n - h)$ **do**
      $lexGT_p \leftarrow \{\vec{x} \rightarrow \vec{y} \mid \vec{x}_{[0..p-1]} = \vec{y}_{[0..p-1]} \wedge \vec{x}_{[p]} > \vec{y}_{[p]}\}$
      $succ_p \leftarrow \{\vec{x} \rightarrow \vec{y} \mid \vec{x} \in \mathcal{R} \wedge \vec{y} \in \mathcal{D}\} \cap lexGT_p$
      $next_{\mathcal{D}} \leftarrow next_{\mathcal{D}} \cup lexmin(succ_p)$
      $\mathcal{R} \leftarrow \mathcal{D} - domain(next_{\mathcal{D}})$
    end for
    return $next_{\mathcal{D}}$
  end procedure
Require: $1 \leq h \leq dim(\mathcal{D})$
  procedure NEXTPOWER($\mathcal{D}$, Δ, h)
    $next_{\mathcal{D}} \leftarrow nextBoulet(\mathcal{D}, h)$
    return $\overbrace{next_{\mathcal{D}} \circ next_{\mathcal{D}} \circ \ldots \circ next_{\mathcal{D}}}^{\Delta}$
  end procedure

L'algorithme 2 s'explique mieux en suivant ses opérations sur l'exemple de la figure 5.8. Il commence par construire la fonction qui calcule le successeur immédiat sur la boucle la plus interne, à la profondeur p (D_2 et $p = 2$ sur l'exemple de la figure 5.8). Lorsqu'il n'y a plus de successeur sur la dimension la plus interne, c'est-à-dire quand l'itérateur est sur la borne supérieure du domaine d'itération, l'algorithme cherche alors un successeur sur la dimension suivante, à la profondeur $p-1$ (D_1 et $p = 1$ sur la figure 5.8). Cette procédure est répétée jusqu'à ce que toutes les dimensions du domaine aient été parcourues, ou quand la dimension $p - h$ est atteinte. Lorsque l'algorithme termine, le point restant est le maximum lexicographique du domaine, et son successeur est $\bot$ ($D_\bot$ et $p = 0$ sur la figure 5.8).

Par exemple, quand $\vec{x} = (i, j)$, la valeur de $next_{D_{S_0}}(\vec{x})$ pour l'exemple de la figure 5.8 est comme suit :

$$next_{D_{S_0}}(i,j) = \begin{cases} (i, j+1) & \text{if} \quad j < N - i - 1 \\ (i+1, 0) & \text{elseif} \quad i < N - 1 \\ \bot & \text{else} \end{cases}$$

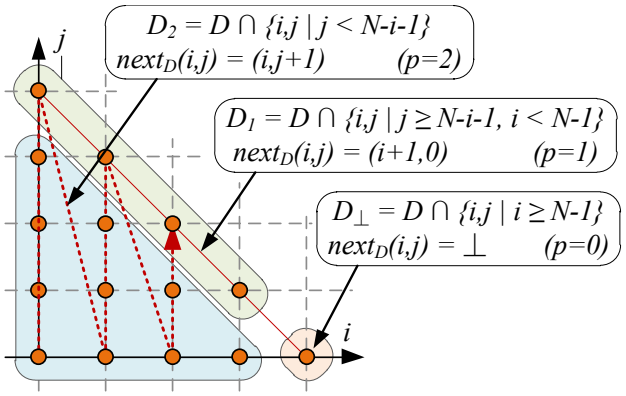


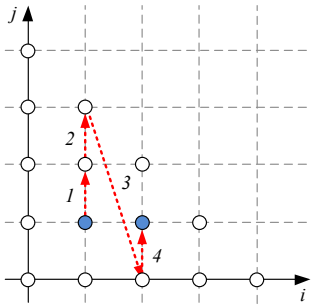
FIGURE 5.8 – Expression du successeur immédiat (la fonction $next_{\mathcal{D}}$) pour l'exemple de la figure 5.1. L'expression varie en fonction de l'itération courante, dans les sous-domaines D_1, D_2 ou $D_\perp$.

Les domaines impliqués dans cet algorithme sont des polyèdres paramétrés. Par conséquent, la fonction $next_{\mathcal{D}}$ peut être calculée grâce à la programmation linéaire en nombres entiers paramétrée [5, 63]. Une solution prend alors la forme d'un Quast (*Quasi-Affine Selection Tree* : Arbre de sélection quasi-affine), représentable par une fonction quasi-affine par morceaux (*piecewise quasi-affine function*, des fonctions linéaires dans lesquelles la division et le modulo par un entier constant sont autorisés). Puisque la latence du pipeline est constante, il est donc possible de construire la fonction $next^{\Delta}_{\mathcal{D}}$ par Δ compositions de $next_{\mathcal{D}}$.

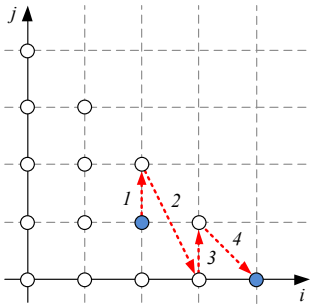
Lorsque la fonction $next_{D_{S_0}}(i,j)$ construite précédemment est composée quatre fois, on obtient alors la fonction $next^4_{D_{S_0}}(i,j)$, qui est donnée par :

$$next^4_{D_{S_0}}(i,j) = \begin{cases} (i,j+4) & \text{if} & j \leq N-i-5 \\ (i+1,3) & \text{elseif} & i \leq N-5 \wedge j = N-i-1 \\ (i+1,2) & \text{elseif} & i \leq N-4 \wedge j = N-i-2 \\ (i+1,1) & \text{elseif} & i \leq N-3 \wedge j = N-i-3 \\ (i+1,0) & \text{elseif} & i \leq N-4 \wedge j = N-i-4 \\ (N-1,0) & \text{elseif} & i = N-3 \wedge j = 1 \wedge N \geq 3 \\ (N-2,0) & \text{elseif} & i = N-4 \wedge j = 3 \wedge N \geq 4 \\ \perp & \text{else} & \end{cases} \tag{5.2}$$

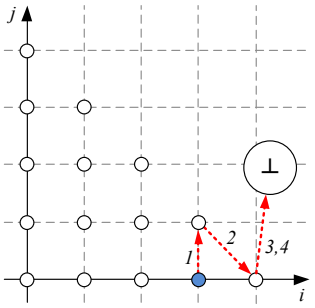
Par exemple, lorsque $N = 5$ et $(i,j) = (1,1)$, la quatrième ligne de l'équation (5.2) est active ($j = N-i-3$ et $i \leq N-3$). Par conséquent, l'expression du successeur 4 itérations plus loin est $(i+1,1) = (2,1)$, qui peut être vérifié sur la figure 5.9a.



(a) En utilisant la 4e ligne de l'équation (5.2), $next^4_{D_{S_0}}(1,1) = (2,1)$



(b) En utilisant la 6e ligne de l'équation (5.2), $next^4_{D_{S_0}}(2,1) = (4,0)$



(c) En utilisant la dernière ligne de l'équation (5.2), $next^4_{D_{S_0}}(3,0) = \perp$

FIGURE 5.9 – Représentation de la fonction $next^4_{D_{S_0}}$ sur l'exemple de la figure 5.1 avec $N = 5$ pour trois exemples.

5.4.4 Construction de l'ensemble des violations de dépendances

Le pipeline de nid de boucles ne viole pas de dépendance de données d si les références $\vec{y}$ d'une source $\vec{x}$ sont exécutées au moins Δ itérations après $\vec{x}$, autrement dit,

$$d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}).$$

Une conséquence de cette condition est que si $next^{\Delta}_{\mathcal{D}}(\vec{x}) = \perp$, c'est-à-dire quand le successeur Δ itérations plus loin n'appartient pas au domaine d'itération, alors la dépendance d est violée lors de l'exécution pipelinée, parce qu'au moins une des référence de $\vec{x}$ recevra la valeur produite « trop tard », à cause de la latence du pipeline. C'est le cas pour la source $(3,0)$ sur l'exemple de la figure 5.1, pour laquelle le successeur après quatre itérations est $\perp$, comme le montre la figure 5.9c.

Cette observation permet de construire l'ensemble $\mathcal{D}^{\dagger}_d$ représentant l'ensemble des itérations sources qui violent la dépendance d :

$$\mathcal{D}^{\dagger}_d = \{\vec{x} \in src(d) \mid d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}) \vee next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\}\} \tag{5.3}$$

Vérifier la légalité du pipeline de nid de boucles vis-à-vis de la dépendance d se résume alors à vérifier que le domaine paramétré $\mathcal{D}^{\dagger}_d$ est vide, ce qui se calcule aisément avec ISL. Vérifier la condition de légalité pour tout un nid de boucles revient à vérifier que l'ensemble $\mathcal{D}^{\dagger}$ est vide, avec :

$$\mathcal{D}^{\dagger} = \bigcup_{d \in P_{\text{RDG}}} \mathcal{D}^{\dagger}_d$$

Exemple

En utilisant l'inverse de d décrit en section 5.4.2, et la fonction $next^4_{D_{S_0}}(i,j)$ de l'équation (5.2), on peut construire le domaine $\mathcal{D}^\dagger_d$ représentant l'ensemble des itérations sources d'une dépendance violée en utilisant l'équation (5.3).

Dans l'exemple de la figure 5.1 avec $\Delta = 4$, et après simplifications, on obtient :

$$\mathcal{D}^\dagger_d = \{i,j | (i,j) \in \mathcal{D}_{S_0} \wedge N-4 < i < N-1 \wedge j < N-i-1\}$$

Puisque d est la seule dépendance du nid de boucles, $\mathcal{D}^\dagger = \mathcal{D}^\dagger_d$. Quand on substitue N par 5 (la valeur choisie dans l'exemple), on a $\mathcal{D}^\dagger = \{(2,0),(2,1),(3,0)\}$ (voir figure 5.10), qui représente l'ensemble des sources qui posent problème sur la figure 5.3b.

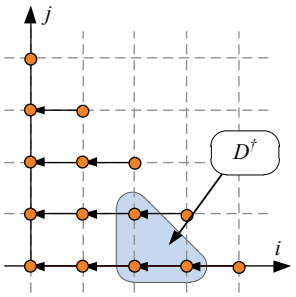


FIGURE 5.10 – Représentation de $\mathcal{D}^\dagger$ pour l'exemple de la figure 5.1 lorsque $\Delta = 4$.

5.4.5 L'algorithme de vérification de légalité

L'algorithme 3 présente le test de légalité pour le pipeline de nid de boucles. L'argument h représente le nombre de boucles internes sur lesquelles le test est appliqué.

L'expansion de BERNSTEIN est utilisée comme filtre pour éviter le calcul de la fonction $next^\Delta_\mathcal{D}$, car ce calcul est très coûteux comme le montreront les résultats du chapitre 6.

La fonction $next^\Delta_\mathcal{D}$ est construite suivant l'algorithme 2. La fonction $restrict(PRDG, h)$ enlève du PRDG les dépendances qui ne sont pas portées par les h boucles les plus internes.

Enfin, cette méthode peut être étendue à un modèle d'exécution pipelinée plus général, dans lequel les lectures et les écritures peuvent se produire à n'importe quel étage du pipeline. Δ pourrait alors être calculé en analysant chaque paire de dépendance. De plus, les dépendances WAR et WAW devraient être prises en considération.

5.5 Insertion de bulles polyédriques

La vérification de la légalité est une étape importante pour l'automatisation du pipeline de nid de boucles. Il est néanmoins possible de faire mieux en *corrigeant* une boucle donnée pour faire en sorte que le pipeline de nid de boucles devienne légal. L'idée est de déterminer *à la compilation* un domaine d'itération dans lequel des états d'attente, ou *bulles*, sont insérées afin

Algorithme 3 Vérifie la légalité du pipeline de nid de boucles

procedure BERNSTEINBOUNDING($\mathcal{D}$, d)
$R \leftarrow \left\{ \vec{x} \rightarrow \vec{z} \;\middle|\; \begin{array}{c} \vec{x} \in src(d) \wedge \vec{z} \in \mathcal{D} \\ \wedge \vec{x} \prec \vec{z} \prec d^{-1}(\vec{x}) \end{array} \right\}$
$P \leftarrow ehrhart_card(R)$
return $bernstein_bound_min(P)$
end procedure
Require: $1 \leq h \leq dim(\mathcal{D})$
procedure LEGALITYCHECK(P_{RDG}, $\mathcal{D}$, Δ, h)
$\mathcal{D}^{\dagger} \leftarrow \emptyset$
$P_{\text{RDG}} = restrict(P_{\text{RDG}}, h)$
for all $d \in P_{\text{RDG}}$ **do**
$l \leftarrow bernsteinBounding(\mathcal{D}, d)$
if $l < \Delta - 1$ **then**
$next^{\Delta}_{\mathcal{D}} \leftarrow nextPower(\mathcal{D}, \Delta, h)$
$\mathcal{D}^{\dagger}_{d} \leftarrow \left\{ \vec{x} \in src(d) \;\middle|\; \begin{array}{c} d^{-1}(\vec{x}) \prec next^{\Delta}_{\mathcal{D}}(\vec{x}) \\ \vee next^{\Delta}_{\mathcal{D}}(\vec{x}) \in \{\perp\} \end{array} \right\}$
$\mathcal{D}^{\dagger} \leftarrow \mathcal{D}^{\dagger} \cup \mathcal{D}^{\dagger}_{d}$
end if
end for
return $\mathcal{D}^{\dagger} = \emptyset$
end procedure

de geler le pipeline. De cette manière, si un nombre suffisant de bulles est inséré entre la source et la référence d'une dépendance violée, l'exécution pipelinée devient légale.

Les méthodes de correction sont contraintes par deux aspects.

La première contrainte concerne l'applicabilité de la méthode de correction. Le mécanisme de correction est restreint aux nids de boucles dont au moins la boucle interne est pipelinable sans insertion de bulles. Dans de tels nids de boucles, les dépendances violées ne sont pas portées par la boucle la plus interne. On peut alors ajouter les bulles à la fin de cette boucle, seulement pour les itérations des boucles externes qui contiennent des sources introduisant une violation de dépendances quand on applique le pipeline de nid de boucles. Cette restriction permet d'obtenir des nids de boucles corrigés relativement simples.

La seconde contrainte est qu'il faut prendre en compte le comportement des outils de HLS lors de l'insertion des bulles. En effet les outils de HLS appliquent des optimisations agressives sur le code source, en particulier l'élimination de code mort (cf. section 2.3.1). Si les bulles insérées sont matérialisées par des boucles n'effectuant aucune opération, l'élimination de code mort supprimerait ces boucles. Pour éviter ce problème, les bulles sont insérées dans le domaine d'itération parcouru par la boucle mise à plat (cf. section 4.2.3). Cet aspect pourrait être contourné par l'introduction d'instruction NOP, que l'outil n'optimiserait pas.

Au contraire, et bien que cela ne soit parfaitement possible (cf. sous-section 5.5.4), les expérimentations ont montré que corriger un nid de boucles dans lequel la boucle interne porte une dépendance violée lorsque le pipeline est appliqué, provoque un accroissement significatif du

nombre de contraintes, ce qui réduit les performances du nid de boucles.

La question clé est alors de déterminer combien de bulles insérer pour rendre le pipeline de nid de boucles légal. En effet, insérer des bulles pour une itération donnée peut par effet de bord corriger plusieurs dépendances violées. La suite de cette section présente trois solutions pour résoudre ce problème, ainsi qu'un exemple dans lequel la boucle la plus interne n'est pas pipelinable sans insertion de bulles.

5.5.1 Complétion simple

La première approche proposée [MDQ11], illustrée par la figure 5.11, consiste à ajouter $\Delta - 1$ bulles à la fin de chaque itération des boucles externes contenant au moins une source dans $\mathcal{D}^\dagger$. En pratique, cela revient à recréer l'épilogue du pipeline, mais seulement pour les itérations des boucles externes qui le nécessitent.

Cette approche est simple mais elle s'avère être trop prudente. En effet il n'est pas nécessaire d'insérer les bulles à la fin de la boucle englobant une source dans $\mathcal{D}^\dagger$. Ajouter ces bulles à la fin de $\mathcal{D}^\dagger$ résulte en un nombre de bulles plus petit, mais suffisant pour assurer la légalité. La construction de cet ensemble de bulles est décrit dans l'algorithme 4, et le résultat de l'algorithme pour l'exemple de la figure 5.1 est présenté en figure 5.11 avec l'approche présentée dans [MDQ11].

Algorithme 4 Construit l'ensemble des bulles polyédriques

Require: $size \geq 0$
  procedure PAD($\mathcal{D}$, $size$)
    $n \leftarrow dim(\mathcal{D})$
    $pad \leftarrow \{\vec{x} \rightarrow \vec{y} \mid \vec{y}_{[0..n-1]} = \vec{x}_{[0..n-1]} \wedge \ \vec{x}_{[n]} \leq \vec{y}_{[n]} \leq \vec{x}_{[n]} + size\}$
    return $pad(\mathcal{D})$
  end procedure
Require: $\mathcal{D}^\dagger \subseteq \mathcal{D}$
  procedure BUBBLESV1($\mathcal{D}$, $\mathcal{D}^\dagger$, Δ)
    $\mathcal{B} \leftarrow \emptyset$
    for all $\mathcal{D}_d^\dagger \in \mathcal{D}^\dagger$ **do**
      $\mathcal{B} \leftarrow \mathcal{B} \cup pad(\mathcal{D}_d^\dagger, \Delta - 1)$
    end for
    return $\mathcal{B} - \mathcal{D}$
  end procedure

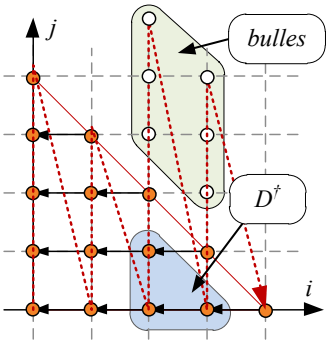
Cependant cette méthode reste encore trop prudente. Par exemple, on remarque que la boucle interne de l'exemple en figure 5.11d lorsque $i = 2$ n'a pas besoin de deux bulles, mais qu'une seule suffit.

5.5.2 Complétion optimisée

L'idée de cette deuxième méthode est de construire l'ensemble des bulles en même temps que $\mathcal{D}^\dagger$, et de compléter chaque boucle interne de $\mathcal{D}$ englobant $\mathcal{D}^\dagger$ avec le nombre exact de bulles nécessaires pour assurer la légalité du pipeline.

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:   Y[j] = func(Y[j]);
   if((i>N-4&&j<N-i+3&&i<N-1)
       ||j<N-i-1)
        j++;
   else
        j=0,i++;
}
```

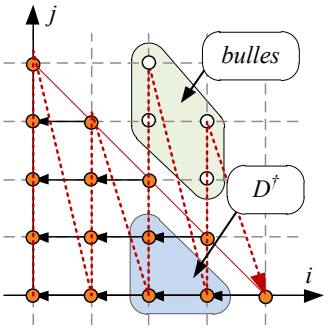
(a) Code généré après l'insertion de bulles décrite dans [MDQ11].



(b) Représentation graphique du domaine d'itération de (a).

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:   Y[j] = func(Y[j]);
   if((i>N-4&&j<N-i+2&&i<N-1)
       ||j<N-i-1)
        j++;
   else
        j=0,i++;
}
```

(c) Code généré après l'insertion de bulles décrite dans l'algorithme 4.



(d) Représentation graphique du domaine d'itération de (c).

FIGURE 5.11 – Illustration de la complétion simple en (c) et (d), comparée à l'approche présentée dans [MDQ11] en (a) et (b), lorsque $\Delta = 4$ (et $N = 5$ pour les représentations graphiques). Les points blancs représentent les bulles insérées.

Soit r la distance de réutilisation entre une source $\vec{x}$ et une référence $\vec{y}$ portant une dépendance d. Si $r < \Delta$, alors la dépendance d est violée lorsqu'on applique le pipeline de nid de boucles avec une latence de Δ. Pour assurer la légalité, il suffit alors d'insérer *au moins* $\Delta - r$ bulles entre $\vec{x}$ et $\vec{y}$.

Le test de légalité est modifié pour calculer l'ensemble $\mathcal{D}_r^\dagger$ des sources introduisant une violation de dépendance d'exactement r cycles, pour $1 \leq r \leq \Delta$. Les boucles internes sont alors complétées par exactement $\Delta - r$ bulles. L'algorithme 5 décrit la méthode, et le résultat est illustré par la figure 5.12.

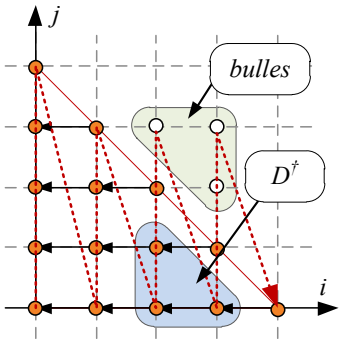
Dans cet exemple, avec $\Delta = 4$, on a $\mathcal{D}_1^\dagger = \emptyset$, $\mathcal{D}_2^\dagger = \{(3,0)\}$ and $\mathcal{D}_3^\dagger = \{(2,0),(2,1)\}$. Par conséquent, il suffit de compléter la boucle interne lorsque $i = 2$ avec une seule bulle, et la boucle interne lorsque $i = 3$ avec deux bulles pour obtenir un pipeline légal.

Algorithme 5 Test de légalité et insertion de bulles entrelacés pour construire l'ensemble des bulles optimisé.

Require: $\mathcal{D}_1 \subseteq \mathcal{D}_2$
  procedure ENCLOSING($\mathcal{D}_1$, $\mathcal{D}_2$)
    $n \leftarrow dim(\mathcal{D}_1)$
    return $projectOut(\mathcal{D}_1, n-1) \cap \mathcal{D}2$
  end procedure
Require: $1 \leq h \leq dim(\mathcal{D})$
  procedure BUBBLESV2(P_{RDG}, $\mathcal{D}$, Δ, h)
    $P_{\text{RDG}} \leftarrow restrict(P_{\text{RDG}}, h)$
    $\mathcal{B} \leftarrow \emptyset$
    for all $d \in P_{\text{RDG}}$ **do**
      $l \leftarrow bernsteinBounding(\mathcal{D}, d)$
      if $l < \Delta - 1$ **then**
        for all $r \in [1..\Delta - 1]$ **do**
          $next^r_{\mathcal{D}} \leftarrow nextPower(\mathcal{D}, r, h)$
          $\mathcal{D}^{\dagger}_{d_r} \leftarrow \{\vec{x} \in src(d) | d^{-1}(\vec{x}) = next^r_{\mathcal{D}}(\vec{x})\}$
          if $\mathcal{D}^{\dagger}_{d_r} \neq \emptyset$ **then**
            $\mathcal{B} \leftarrow \mathcal{B} \cup pad(enclosing(\mathcal{D}^{\dagger}_{d_r}, \mathcal{D}), \Delta - r)$
          end if
        end for
      end if
    end for
    return $\mathcal{B} - \mathcal{D}$
  end procedure

```
i=0;j=0;
while(i<N) {
#pragma ignore_mem_depcy Y
   if(j<N-i)
S0:    Y[j] = func(Y[j]);
   if((i>N-4&&j<4&&i<N-1)
       ||j<N-i-1)
        j++;
   else
        j=0,i++;
}
```

(a) Code généré après l'insertion de bulles décrite dans l'algorithme 5.

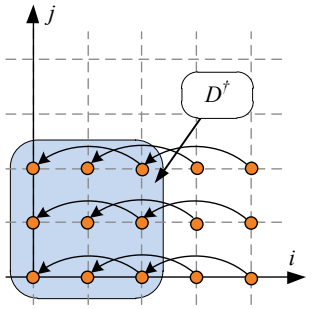


(b) Représentation graphique du domaine d'itération de (a).

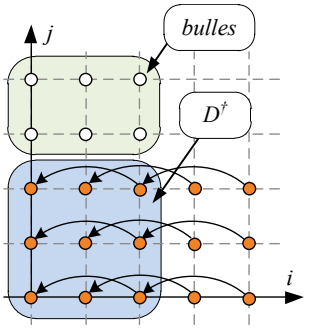
FIGURE 5.12 – Complétion optimisée pour l'exemple de la figure 5.1, lorsque $\Delta = 4$ (et $N = 5$ pour la représentation graphique).

5.5.3 Complétion itérative

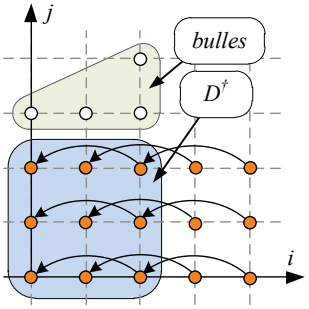
Malgré son apparente efficacité, cette deuxième méthode n'offre toujours pas une approche optimale (en termes de nombre de bulles) pour l'insertion de bulles. Comme le montre l'exemple de la figure 5.13, l'insertion de bulle optimisée de la section précédente insère des bulles inutiles dans certains cas (en particulier quand les dépendances traversent plusieurs itérations de la boucle externe).



(a) Exemple de domaine polyédrique, avec $\mathcal{D}^\dagger$ lorsque $\Delta = 8$.



(b) Résultat de l'algorithme 5 lorsque $\Delta = 8$.



(c) Résultat de la complétion itérative lorsque $\Delta = 8$.

FIGURE 5.13 – Exemple de domaine polyédrique dans lequel les dépendances de données traversent deux itérations de la boucle externe. Lorsque $\Delta = 8$, toutes les sources introduisent une violation de dépendance. L'algorithme 5 insère alors exactement le nombre de bulles nécessaire pour chaque itération de la boucle externe, alors qu'après deux étapes de l'algorithme 6, l'analyse détermine que toutes les dépendances sont respectées lorsqu'une seule bulle est insérée pour les deux premières itérations de la boucle externe, et deux bulles lorsque $i = 3$.

La méthode d'insertion de bulle itérative est présentée par l'algorithme 6. Le principe est de calculer l'ensemble des sources introduisant une violation de dépendance, et de compléter les boucles englobantes avec une seule bulle, puis de relancer l'analyse sur ce nouveau domaine d'itération. L'algorithme se termine lorsque l'analyse ne trouve plus de dépendances violées.

Algorithme 6 Construit l'ensemble des bulles polyédriques itérativement, où *LegalityCheck* de l'algorithme 3 retourne $\mathcal{D}^\dagger$ au lieu de $(\mathcal{D}^\dagger = \emptyset)$.

Require: $1 \leq h \leq dim(\mathcal{D})$
  procedure BUBBLESV3(PRDG, $\mathcal{D}$, h)
    $\mathcal{D}^\dagger \leftarrow LegalityCheck^*(P_{\text{RDG}}, \mathcal{D}, h)$
    $\mathcal{B} \leftarrow \emptyset$
    if $\mathcal{D}^\dagger \neq \emptyset$ **then**
      $\mathcal{B}' \leftarrow pad(enclosing(\mathcal{D}^\dagger, \mathcal{D}), 1)$
      $\mathcal{B} \leftarrow \mathcal{B}' \cup bubblesV3(P_{\text{RDG}}, \mathcal{D} \cup \mathcal{B}', h)$
    end if
    return $\mathcal{B} - \mathcal{D}$
  end procedure

Malgré une plus grande précision lors de l'insertion, on remarquera que l'algorithme n'est toujours pas optimal lorsqu'on s'intéresse au placement des bulles. En effet, la bulle insérée en $(i = 2, j = 4)$ sur la figure 5.13 serait mieux placée en $(i = 3, j = 3)$, comme illustré sur la figure 5.14. Elle éviterait alors aux itérations lorsque $i = 3$ d'attendre un cycle « de trop ». Aucune solution n'a été trouvée en ce qui concerne ce placement.

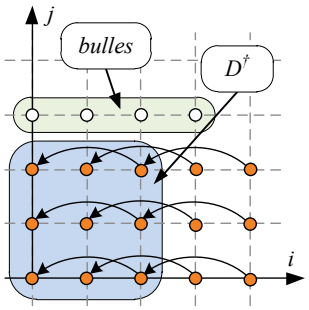


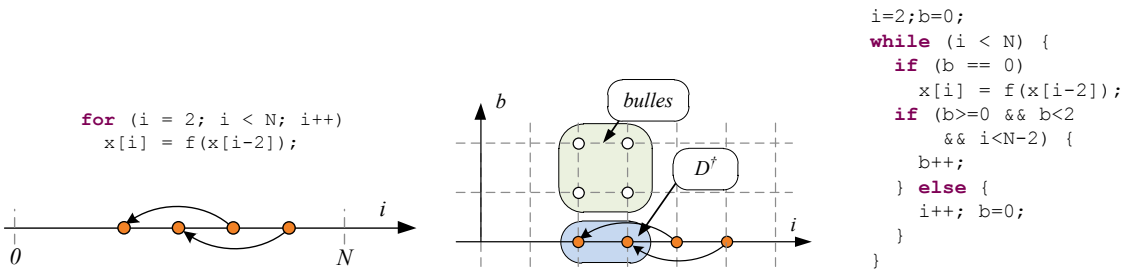
FIGURE 5.14 – Exemple de la figure 5.13 dans lequel la bulle insérée en $(2, 4)$ est déplacée en $(3, 3)$.

5.5.4 Extension à l'insertion de bulles sur la boucle interne

Dans l'introduction de cette section, une des contraintes restreint l'applicabilité de l'insertion de bulles aux nids de boucles dont au moins la boucle interne est pipelinable sans insertion de bulles. En pratique, l'insertion de bulle est tout à fait possible dans ce cas.

Les techniques présentées insèrent les bulles à la fin de la boucle interne, car elles supposent qu'aucune dépendance n'est portée au sein de la boucle interne. Pour inclure les cas où la boucle

interne porte une dépendance violée par l'application du pipeline, il suffit d'ajouter une nouvelle dimension plus interne au domaine d'itération. Cette nouvelle dimension est alors dédiée à la représentation des bulles, comme le montre la figure 5.15. Cependant les expérimentations ont montré qu'en pratique les domaines deviennent très complexes, et les outils de HLS sont alors incapables de générer des circuits performants.



(a) Exemple de boucle.

(b) Résultat de l'algorithme 5 après ajout d'une dimension, pour $\Delta = 4$.

(c) Machine à états générée.

FIGURE 5.15 – Exemple de boucle et de la représentation graphique de son domaine d'itération et de ses dépendances (a). Une nouvelle dimension b est ajoutée pour insérer des bulles.

5.6 Conclusion

Ce chapitre a présenté deux approches pour vérifier de manière conservative, puis de manière précise la légalité du pipeline de nid de boucles pour les Scop. De plus, trois approches ont été présentées pour corriger ces nids de boucles lorsque l'application pipeline introduit des violations de dépendances.

Ces travaux démontrent que les techniques d'analyse de haut niveau, basées sur les représentations mathématiques des boucles, sont tout à fait adaptées aux problèmes liés à la synthèse de haut niveau. Implémentées dans un compilateur source-à-source, elles ont aussi permis de mettre en avant le potentiel des transformations source-à-sources pour contourner les limitations des outils de HLS d'aujourd'hui.

Les détails de l'implémentation dans le compilateur source-à-source GeCoS ainsi que les expérimentations des techniques présentées dans ce chapitre sont présentés dans le chapitre suivant.

Chapitre 6

Implémentation et résultats

6.1 Introduction

Les techniques présentées dans les chapitres précédents ont été implémentées dans le compilateur GeCoS développé par l'équipe Cairn [6]. On a pu ainsi réaliser un compilateur source-à-source, visant à transformer un programme source C avant de lui appliquer la synthèse de haut niveau. Cette implémentation a en outre permis d'évaluer les techniques présentées dans les chapitres précédents en termes de qualité d'analyse et de qualité de résultats.

Ce chapitre comporte deux parties. Dans la section 6.2, nous présentons l'infrastructure de compilation GeCoS ainsi que les développements qui ont été réalisés en lien avec cette thèse. Dans la section 6.3, nous présentons les résultats obtenus en appliquant les techniques des chapitres 4 et 5 à un ensemble représentatif d'applications, de manière à vérifier leur applicabilité.

6.2 GeCoS : Generic Compiler Suite

GeCoS (*Generic Compiler Suite*) est une infrastructure de compilation source-à-source libre [6] intégrée dans l'environnement de développement Eclipse, et basée sur les techniques de MDE (*Model Driven Engineering* : Ingénierie Dirigée par les Modèles). GeCoS cible en particulier la HLS (*High-Level Synthesis* : Synthèse de Haut Niveau), et supporte les types de données de Mentor (`ac_int` et `ac_fixed`) [107]. C'est cette infrastructure qui a été utilisée pour mettre en œuvre les délivrables du projet Nano2012-S2S4HLS soutenu par Inria et STMicroelectronics.

GeCoS fournit entre autres un environnement de transformation basé sur la représentation polyédrique des nids de boucles (cf. figure 6.1), utilisant des bibliothèques tierces (Isl [64] pour manipuler les domaines polyédriques et résoudre les problèmes linéaires en nombre entiers paramétriques, et Cloog [7] pour la génération de code dans le modèle polyédrique). Toutes les transformations présentées dans les chapitres précédents ont été implémentées dans cet environnement.

Dans la présente section, nous décrivons tout d'abord le contexte industriel des travaux et l'intérêt de la compilation source-à-source dans un flot de transformation optimisant. Ensuite, nous rappelons les avantages liés à l'utilisation des techniques d'ingénierie dirigée par les modèles

dans la mise en œuvre d'un tel compilateur source-à-source. Enfin, les développements liés à cette thèse sont présentés.

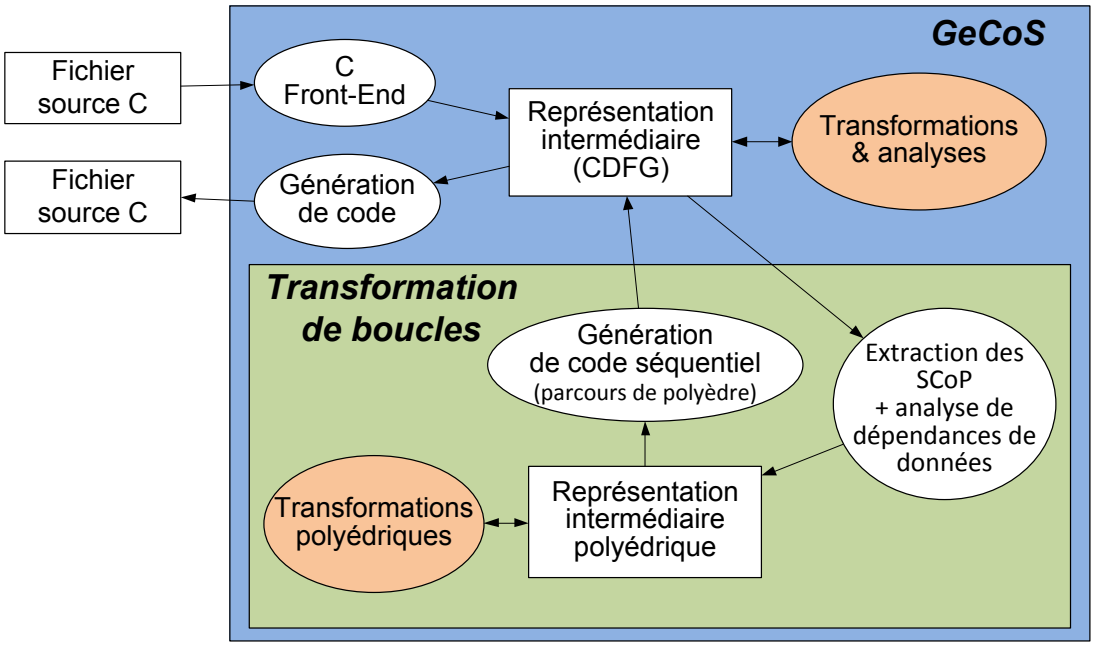


FIGURE 6.1 – L'infrastructure de GeCoS se base sur le frontal C/C++ fourni par le greffon CDT (C Development Tools) intégré à Eclipse pour construire la Représentation Intermédiaire (RI) d'un fichier C. Des transformations peuvent être appliquées à cette RI, et GeCoS propose un générateur de code C. Le flot polyédrique est similaire à celui décrit dans le chapitre 3.

6.2.1 Nano2012-S2S4HLS

Le projet S2S4HLS (*Source-To-Source For High-Level Synthesis* : source-à-source pour la synthèse de haut niveau), soutenu par Inria et STMicroelectronics, intervient dans le cadre du programme de recherche Nano2012.

Ce projet par du constat que les fabricants et concepteurs de systèmes intégrés sur silicium (tels que STMicroelectronics) s'orientent vers une utilisation en production des outils de HLS, de manière à gagner en productivité. Cependant, comme présenté dans le chapitre 2, ces outils de HLS sont très dépendants de la structure du code source et sont incapable de générer un circuit performant lorsque le code source n'est pas correctement structuré ou annoté.

Ainsi l'étape d'exploration de l'espace de conception, précédemment réalisé lors de l'étape de traduction de la spécification algorithmique vers la description matérielle, s'effectue maintenant sur la description haut niveau (en C/C++). De manière à maximiser l'automatisation de cette étape, le projet S2S4HLS à pour objectif la mise en œuvre d'une boite à outils permettant d'optimiser un code source en C/C++. Ces transformations source-à-source sont regroupées en trois sous-projets ayant chacun un objectif particulier :

– S2S4HLS-SP1 : Transformations de boucles orientées Hls ;
– S2S4HLS-SP2 : Conception de systèmes en virgule fixe ;
– S2S4HLS-SP3 : Synthèse d'architectures multi-modes.

Ces trois sous-projets sont brièvement décrits ci-dessous.

S2S4HLS-SP1 : Transformations de boucles orientées Hls

Les transformations de vectorisation et de pipeline permettent d'accélérer l'exécution des nids de boucles sur un accélérateur matériel. Néanmoins les outils de Hls ne peuvent appliquer ces transformations qu'après avoir déterminé que les boucles ne portent pas de dépendances. Or les nids de boucles ne présentent pas toujours ces caractéristiques, et il faut alors les transformer avant de pouvoir appliquer la vectorisation ou le pipeline. De plus, pour minimiser l'emprunte mémoire destinées à stocker les valeurs, il est possible contracter les tableaux [70]. Cette technique permet de réduire la taille des tableaux au prix d'un adressage parfois plus complexe.

L'objectif du sous-projet SP1 est de simplifier la transformation des nids de boucles pour pouvoir ensuite pipeliner ou vectoriser leur exécution sur un accélérateur matériel. Ce sous-projet est détaillé dans la section 6.2.5.

S2S4HLS-SP2 : Conception de systèmes en virgule fixe

Les spécifications algorithmiques peuvent utiliser des types de données réelles, qui sont généralement mis en œuvre par des types de données en virgule flottante (par exemple les types *float* et *double* en C/C++). Cependant le coût en ressources matérielles nécessaire à la mise en œuvre matérielle des opérateurs travaillant sur des types à virgule flottante est très élevé. Pour éviter d'avoir à utiliser de tels opérateurs, il est possible de représenter ces valeurs sur des types à virgule fixe.

Le principe des types à virgule fixe est de représenter la partie entière et la partie fractionnaire séparément. On détermine alors un nombre de bits pour chacune de ces deux parties, en s'assurant que le nombre de bits pour la partie entière est suffisant pour éviter les débordement, et que le nombre de bits pour la partie fractionnaire minimise le bruit dû à la perte de précision [121].

L'objectif du sous-projet SP2 est d'automatiser, de manière analytique, la conversion d'une spécification en virgule flottante vers une spéficication en virgule fixe.

S2S4HLS-SP3 : Synthèse d'architectures multi-modes

De manière à maximiser le parallélisme instruction exploitable par les outils de Hls dans les graphes de données, il est commun de mettre à plat le contrôle, ce qui permet à l'outil de travailler sur des graphes plus grands, maximisant les opportunités. Cependant, travailler sur des graphes plus grand, en particulier avec des problèmes NP-difficiles, conduit à des temps de synthèse qui explosent.

Pour réduire ces temps de synthèse, une solution consiste à recherche des motifs dans le graphe de données, puis à considérer ces motifs comme des nœuds du graphe [122]. Ainsi chaque motif représente un macro-opérateur, ce qui réduit la taille du graphe. Par la même occasion, les

macro-opérateurs peuvent être synthétisé en architectures multi-modes et réutilisés par d'autres algorithmes, de manière à réduire la surface nécessaire à leur mise en œuvre matérielle.

L'objectif du sous-projet SP3 est de déterminer automatiquement, en amont des outils de HLS, des points de similarité dans des graphes de données, afin d'en extraire des macro-opérateurs, de manière à accélérer la synthèse et minimiser le coût en ressources matérielles.

6.2.2 Compilateur source-à-source

La compilation source-à-source, dans le contexte des transformations optimisantes de haut niveau, apporte plusieurs avantages :

- Les codes sources optimisés sont difficiles à lire et à maintenir. En utilisant un compilateur optimisant source-à-source, les optimisations sont insérées automatiquement à partir d'un code source original lisible et maintenable.
- Regénérer du C permet aux concepteurs d'observer le résultat des transformations, et de le comparer à d'autres implémentations, voire de modifier le résultat de la compilation plus facilement que sur du langage machine ou une description matérielle en HDL (*Hardware Description Language* : Langage de Description Matérielle). De plus une compilation logicielle permet d'obtenir rapidement un prototype qui donne alors un aperçu du comportement après transformation.
- Les transformations réalisées par un compilateur source-à-source ne sont pas liées à un outil (compilateur logiciel ou outil HLS) en particulier. Le code généré par GeCoS peut être synthétisé par n'importe quel compilateur acceptant un programme en langage C++, ou par les outils de HLS.

Le langage C est une cible pertinente pour les compilateurs source-à-source. En effet ce langage est parmi les plus utilisés pour décrire des standards (de codage des flux multimédias, ou de protocoles de communications par exemple). De plus, la majorité des outils de HLS utilisent le langage C (en général un sous-ensemble, parfois étendu par des fonctions spécifiques à la descrption de matériel) comme format d'entrée.

Plusieurs compilateurs source-à-source existent déjà, et ont montré le potentiel d'une approche source-à-source. Parmi ceux-ci, PIPS [123], Pluto [2], Rose [4] ou encore CETUS [3] offrent la possibilité d'optimiser automatiquement des programmes écrits en C, en mettant en avant parallélisme ou localité. Cependant ces compilateurs ciblent principalement la compilation logicielle et le calcul hautes performances.

6.2.3 Ingénierie dirigée par les modèles

Les différentes représentations intermédiaires utilisées dans GeCoS ont toutes été entièrement formalisées dans le méta modèle Ecore [124], suivant les approche d'ingénierie dirigée par les modèles. FLOC'H et coll. [125] décrivent les avantages et inconvénients d'une telle approche lors de la conception de compilateurs optimisants. Les principaux sont rappelés ici.

Dans un compilateur, l'objectif des différentes représentations intérmédiaires est de représenter, ou modéliser, les programmes, c'est-à-dire les données manipulées par le compilateur. Dans

ce sens, la modélisation de ces représentations intermédiaires dans un même formalisme (par exemple UML [126] ou Ecore [124]) offre une homogénéité, toutes étant décrites de la même manière. De plus, formaliser les structures documente, en quelque sorte, les choix réalisés, ce qui est un avantage certain dans un contexte de recherche académique où les prototypes non documentés sont légion. Cette documentation implicite offre en outre des facilités pour la maintenance.

Les outils disponibles travaillant sur les modèles permettent de faciliter et d'accélérer les développements. Les générateurs de code et les outils de manipulation des modèles permettent de créer des prototypes d'application très rapidement. Il est par exemple possible d'effectuer des requêtes très complexes pour réécrire ou rechercher des motifs dans des structures arborescentes grâces à Tom/Gom [127].

Comme le souligne Floc'h dans sa thèse [14], ces avantages viennent au prix de certains prérequis de la part des utilisateurs et des développeurs. Ils doivent ainsi être capables de travailler à des niveaux d'abstraction plus élevés, tout en séparant de manière plus nette la modélisation structurelle des traitements.

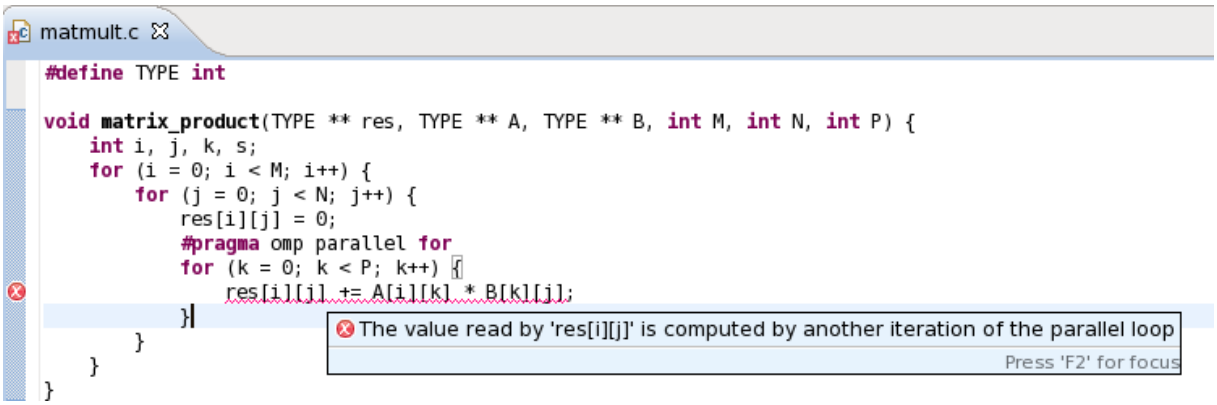
6.2.4 `ompVerify`

Le développement d'applications parallèles est difficile, et l'écart à franchir entre la programmation séquentielle et la programmation parallèle est grand. Lors du développement d'applications séquentielles, les développeurs peuvent compter sur les outils pour leur apporter des informations sur le programme, et ce de manière instantanée. Cependant il existe peu d'outils offrant de telles fonctionnalités pour des programmes parallèles, et ils souffrent souvent de restrictions limitant l'applicabilité.

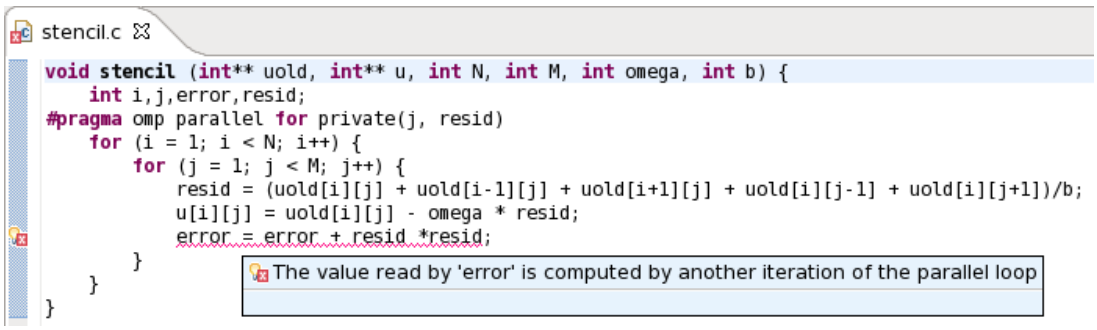
OpenMP permet de spécifier, grâce à des directives de compilation (`#pragma omp parallel for`), qu'une boucle peut être exécutée de manière parallèle. C'est cependant au développeur de s'assurer que l'exécution parallèle préserve la sémantique, ce qui est loin d'être évident.

`ompVerify` [BYR+11] a été développé conjointement par l'équipe Cairn, à l'Irisa, et l'équipe Mélange, à Colorado State University (CSU). L'objectif de cet outil est de proposer une vérification instantanée de certaines directives pour les développeurs d'applications parallèles utilisant le formalisme OpenMP sur des boucles représentables dans le modèle polyédrique. En particulier, `ompVerify` permet de déterminer si une boucle spécifiée comme parallèle via des annotations OpenMP assure le même comportement que son implémentation séquentielle. La figure 6.2 montre deux captures d'écran de `ompVerify`.

`ompVerify` est implémenté dans GeCoS, et se base sur une analyse implémentée dans le compilateur AlphaZ [83], développé à CSU. Après l'analyse polyédrique de GeCoS, le Scop (*Static Control Programs* : Programmes à Contrôle Statique) est converti vers la représentation intermédiaire de AlphaZ. Une fois dans AlphaZ, l'ordonnancement identité est annoté, de manière à faire apparaître des dimensions parallèles (celles correspondant aux boucles annotées avec des pragmas OpenMP). La validité de cet ordonnancement parallèle est alors vérifié (cf. section 3.3), en prenant en compte l'allocation mémoire originale. Comme le montre la figure 6.2, les annotations insérées qui altèrent la sémantique du programme sont reportées directement à l'utilisateur dans l'éditeur.



(a) Multiplication de matrices, dans laquelle la boucle la plus interne est illégalement spécifiée comme parallèle. Les différentes itérations de la boucle interne, exécutées en parallèle, écrivent leur résultat dans le même emplacement mémoire `res[i][j]`, résultant en l'altération de la sémantique originale.



(b) Algorithme de stencil, dans lequel la boucle externe est illégalement spécifiée comme parallèle. Les différentes itérations écrivent leur résultat dans le même emplacement mémoire `error`, résultant en l'altération de la sémantique originale.

FIGURE 6.2 – Exemples de programmes annotés avec des pragmas OpenMP. Les spécifications parallèles illégales sont reportées à l'utilisateur directement dans l'éditeur.

6.2.5 Environnement de manipulation de boucles

Dans un contexte plus général que `ompVerify`, GeCoS propose un environnement de transformation de boucles dirigé par des annotations et des scripts, illustré par la figure 6.3 et que nous décrivons ci-dessous. Cet environnement de transformation de boucles correspond aux délivrables du sous-projet S2S4HLS-SP1.

Édition

Le point d'entrée est un programme C/C++ décrivant un noyau de calcul, ainsi qu'un script de compilation. GeCoS fournit un éditeur pour les scripts de compilation (cf. figure 6.4a), et se base sur le greffon Codan (Code Analysis) de CDT pour proposer des informations instantanément lors de l'édition des programmes sources C/C++.

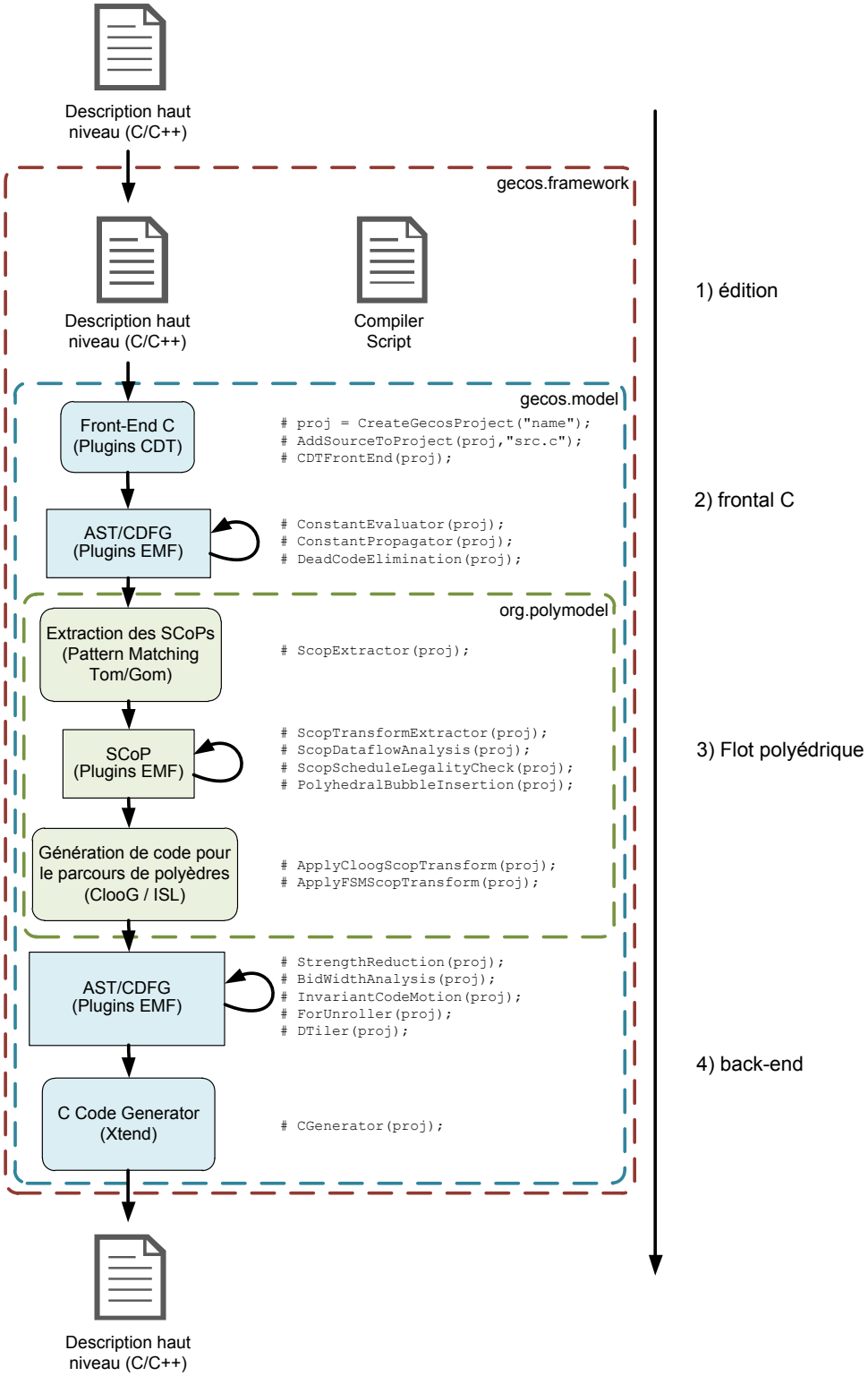


FIGURE 6.3 – Flot de compilation source-à-source de GECOS.

En particulier, grâce à des analyses sur la représentation intermédiaire de GeCoS, l'environnement détecte automatiquement les boucles représentables dans le modèle polyédrique (cf. figure 6.4b). De la même manière que `ompVerify`, cet environnement analyse des ordonnancements polyédriques spécifiés sous forme de pragmas, et reporte ceux qui altèrent la sémantique à l'utilisateur directement dans l'éditeur (cf. figure 6.4c).

Frontal C

Dans le cadre des transformations des boucles, le frontal C de GeCoS consiste dans un premier temps en une transformation de modèle, qui permet d'obtenir une représentation intermédiaire du programme C sous la forme d'une instance d'un modèle EMF (*Eclipse Modeling Framework*) à partir de la représentation intermédiaire de CDT.

Dans un second temps, une série de transformations sont appliquées à la représentation intermédiaire, avec pour objectif d'améliorer l'applicabilité de la détection des parties polyédriques. Ces transformations sont pour l'instant limitées à la propagation et l'évaluation de constantes, ainsi qu'à l'élimination de code mort.

Flot polyédrique

Le flot polyédrique implémenté dans GeCoS commence par l'extraction des Scop. Cette extraction est réalisée grâce à un filtrage effectué par Tom/Gom sur la représentation intermédiaire de GeCoS. C'est cette même passe qui est utilisée dans lors de l'édition pour reporter des informations à l'utilisateur.

De manière similaire à `ompVerify`, les transformations sont spécifiées grâce à des directives dans lesquels les ordonnancements sont donnés manuellement. En utilisant les annotations appropriées (`#pragma scop, scheduling={pluto-isl|feautrier-isl}`), il est possible d'appeler des ordonnanceurs implémentés dans ISL pour transformer automatiquement la boucle. Les ordonnancements sont calculés en fonction des dépendances extraites par l'analyse de dépendances de données exacte sur les tableaux, telle qu'elle a été présentée dans le chapitre 3, et implémentée dans GeCoS.

L'insertion de bulles pour assurer la légalité du pipeline, présentée dans le chapitre 5, est contrôlée par l'annotation `#pragma scop_insert_bubble` pour spécifier les valeurs de latence et de profondeur d'analyse. Toutes les passes sont appelées explicitement dans le script de compilation grâce aux commandes appropriées (cf. figure 6.3).

La génération de code pour le parcours de polyèdre se base sur la bibliothèque tierce CLooG, ainsi que sur une implémentation de la méthode de BOULET et FEAUTRIER présentée dans le chapitre 4.

Post-traitement et générateur de code C

Avant de produire un programme C soumis à la synthèse de haut niveau, plusieurs transformations peuvent être appliquées à la représentation intermédiaire. Pour réduire la complexité

```
# Create a Gecos project
proj = CreateGecosProject("PBI");
AddSourceToGecosProject(proj,"fw.c");
CDTFrontend(proj);

# Extract & analyse SCoPs
scops = ScopExtractor(proj);          #extract SCoPs
GecosScopDataflowAnalysis(proj);      #compute ADA
ScopTransformExtractor(proj);         #extract SCoP pragmas
GScopScheduleLegalityCheck(proj);     #check schedules legality

# Transform SCoPs
ApplyFSMScopTransform(proj);

# Regenerate source code
GecosProjectCGenerator(proj,"src-c-regen-FSM");
```

(a) Les scripts de compilation dans GeCos permettent de définir une séquence de passes à appliquer à un programme. En l'occurrence, ce script construit la représentation intermédiaire associée au fichier `fw.c`, extrait la représentation polyédrique, et regénère le code sous forme de machine à états (cf. section 4.2.3).

```
void floyd_warshall(int n, double path[256][256]) {
 int i, j, k;
 for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
    path[i][j] =
      (path[i][j] < (double) (path[i][k] + path[k][j])) ?
        path[i][j] : (double) (path[i][k] + path[k][j]);
 }
}
```

(b) Implémentation de l'algorithme de Floyd Warshall. L'éditeur détecte instantanément que le nid de boucles est un Scop.

```
void floyd_warshall(int n, double path[256][256]) {
 int i, j, k;
#pragma scop_cloog_options coalescingDepth=2
#pragma scop floyd_warshall, dims = (k,i:PAR,j:PAR)
 for (k = 0; k < n; k++) {
  for (i = 0; i < n; i++)
   for (j = 0; j < n; j++)
#pragma scop_schedule_statement (k,i+j,j)
    path[i][j] =
      (path[i][j] < (double) (path[i][k] + path[k][j])) ?
        path[i][j] : (double) (path[i][k] + path[k][j]);
 }
}
```

(c) L'environnement de transformation permet de spécifier des transformations polyédriques via des annotations. Lorsque celles-ci altèrent la sémantique originale, elles sont alors reportées comme des erreurs sur les accès mémoires dans l'éditeur.

FIGURE 6.4 – Captures d'écran réalisées dans l'environnement de transformation de boucles de GeCos. (a) et (b) représentent un programme C avec des rapports sur les annotations, et (c) présente un script de compilation de GeCos.

du code, comme cela a été présenté dans le chapitre 2, la réduction de force, l'analyse de préci-

sion des types de contrôle, ainsi que le déplacement du code invariant sont trois transformations intéressantes. Les transformations de déroulage de boucles et de pavage dynamique [128, 129] peuvent aussi être appliquées avant la génération de code. La génération du code source C en elle même consiste en un simple parcours de la représentation intermédiaire.

6.3 Résultats expérimentaux

Cette section décrit comment le test de légalité de pipeline a été implémenté dans le compilateur, et fournit des résultats qualitatifs et quantitatifs. Ces résultats montrent que les approches conduisent à des gains en performance significatifs, au prix d'un surcoût en surface modéré.

6.3.1 Protocole expérimental

Pour évaluer les approches décrites dans les chapitres précédents, un ensemble de noyaux de calcul intensif représentatifs des algorithmes utilisés dans les applications embarquées a été sélectionné. Ces noyaux de calculs ont été choisis pour mettre à l'épreuve la robustesse et l'exactitude de l'outil de HLS de référence sur des cas non triviaux. Quand cela était nécessaire, ils ont été modifiés, via des transformations polyédriques (cf. chapitre 3), pour rendre la boucle la plus interne parallèle, et ainsi faire en sorte que la pipeliner soit légal.

Les noyaux de calculs choisis sont les suivants :

- Prodmat : le produit de deux matrices, dans lequel les dépendances sur l'accumulation ont été déplacées sur le deuxième niveau de boucle grâce à une permutation.
- BBFIR : un filtre FIR où la boucle est « penchée » (*skewed*) pour enlever les dépendances sur la boucle la plus interne. C'est le seul noyau de calculs avec seulement deux niveaux de boucle.
- Jacobi : un *stencil* Jacobi 2D, avec la même transformation que BBFIR.
- FW : une implémentation de l'algorithme de Floyd-Warshall, où le nid de boucles est lui aussi penché.
- QRC : une décomposition QR utilisant des opérateurs CORDIC [1] pour laquelle la boucle la plus interne de l'implémentation C originale est déjà parallèle.

Les noyaux de calculs ont suivi le flot décrit dans l'introduction de cette thèse. Après être passés par le frontal de GeCoS, ils ont été analysés pour en obtenir une représentation polyédrique, et pour pouvoir appliquer les transformations source-à-source, toujours au sein de GeCoS (cf. flot de la figure 6.3). Les noyaux transformés ont ensuite été synthétisés par l'outil de HLS Catapult-C pour obtenir du code VHDL (*Very-high-speed integrated circuits Hardware Description Language*). Le code VHDL résultant a ensuite été synthétisé par Quartus, en ciblant un FPGA (*Field Programmable Gate Array* : Réseau de Portes Programmables) Altera Stratix IV.

Sauf mention contraire, les valeurs des données manipulées dans les noyaux de calculs sont codées sur un type en virgule fixe sur 32 bits. Chaque noyau s'est vu assigner une latence de

1. *COordinate Rotation DIgital Computer* : Calcul numérique par rotation de coordonnées.

pipeline Δ, de la façon suivante. Grâce aux directives appropriées, l'outil de HLS de référence (Catapult-C) a été forcé à générer une description matérielle pipelinée, potentiellement fausse (en ignorant les dépendances), en ciblant une fréquence de 100 MHz sur un FPGA Altera Stratix IV. Étant donné que la latence du pipeline est essentiellement liée à la complexité des opérations, et beaucoup moins au contrôle, cette méthode fournit une bonne estimation de la latence.

6.3.2 Résultats qualitatifs

La première expérimentation consistait à vérifier que l'outil de HLS de référence (Catapult-C) était bien capable d'appliquer le pipeline sur la boucle la plus interne sans directive (1D), et de s'assurer qu'il interdisait bien le pipeline du deuxième niveau de boucle (2D). Les résultats pour les noyaux de calcul sont présentés dans la partie gauche du tableau 6.1. Catapult-C n'est capable de pipeliner que le noyau *Prodmat*, dans lequel les accès mémoire sont simples (colonne 1D). Par contre, Catapult-C a aussi autorisé, et appliqué, le pipeline du deuxième niveau de boucle, alors que cela conduit à une violation de dépendance quand la taille des matrices est plus petite que la latence du pipeline (colonne 2D). Pour tous les autres noyaux de calculs, l'analyse de dépendances de Catapult-C est trop prudente, et elle interdit le pipeline de la boucle la plus interne et du deuxième niveau, alors qu'il est légal de pipeliner la boucle la plus interne. Des expérimentations sur une version plus récente de Catapult-C ont montré que ce défaut de conception est aujourd'hui corrigé.

				Temps d'exécution (ms)				
		RHLS		**PBI V1**			**PBI V2**	
Application	Δ	1D	2D	1D	2D	3D	2D	3D
Prodmat	4	ok	illégal	13	230	1671	23	38
BBFIR	4	échoue	interdit	17	2125		131	
Jacobi	8	échoue	interdit	27	623	2918	153	273
FW	3	échoue	interdit	54	69	224	88	183
QRC	13	échoue	interdit	30	2449	879	284	718

ok : le pipeline est légal, et l'outil l'applique.
échoue : le pipeline est légal, et l'outil ne l'applique pas.
interdit : le pipeline est illégal, et l'outil ne l'applique pas.
illégal : le pipeline est illégal, et l'outil l'applique quand même.

TABLE 6.1 – Résultats de l'outil de HLS de référence Catapult-C (RHLS), et le temps d'exécution (Xeon à 2.4GHz) en millisecondes des algorithmes d'insertion de bulles (PBI), pour la latence donnée. L'analyse pour la boucle la plus interne (colonne 1D) prend exactement le même temps pour les 2 méthodes. Lorsque des bulles sont nécessaires l'algorithme 5 (V2) est plus rapide.

La partie droite du tableau 6.1 présente le temps d'exécution requis par les méthodes présentées dans le chapitre 5 (PBI : *Polyhedral Bubble Insertion*). Les algorithmes 4 (V1), 5 (V2), et 6 ont été exécutés sur ces noyaux. Les résultats de l'algorithme 6 ne sont pas présentés. En effet l'implémentation de cet algorithme sur des exemples réalistes (autres que celui utilisé dans le

chapitre 5) ne termine pas après une heure d'exécution. La colonne 1D donne le temps nécessaire pour vérifier que la boucle interne ne porte pas de dépendances. Cette durée ne varie pas suivant les algorithmes. Les colonnes 2D V1 et 2D V2 donnent le temps requis par les algorithmes 4 et 5 pour construire l'ensemble des bulles à insérer lorsque le deuxième niveau de boucle est pipeliné, les colonnes 3D lorsque la boucle est complètement pipelinée.

Ces temps d'exécution dépendent de la forme du domaine d'itération (mais pas de sa taille) et de la latence Δ. Bien que longs dans un contexte de compilateur, ces temps restent acceptables dans le contexte des transformations source-à-source pour la HLS (entre quelques centaines de millisecondes et quelques secondes).

Dans l'ensemble, l'algorithme 5 est plus rapide que l'algorithme 4. Cela s'explique par la façon dont sont construits les problèmes de programmation linéaire. L'algorithme 4 vérifie la légalité du pipeline pour toutes les distances entre 1 et Δ grâce à un seul système d'inégalités. L'algorithme 5 construit plusieurs systèmes (un par distance entre 1 et Δ), qui impliquent une égalité au lieu de deux inégalités, et qui sont par conséquent plus rapides à résoudre.

La génération de la fonction $next_{\mathcal{D}}^{\Delta}$ est une des parties les plus complexes des algorithmes d'insertion de bulles. Le tableau 6.2 montre que le temps d'exécution reste acceptable tant que la latence Δ et la profondeur h sont raisonnables. Bien que le temps d'exécution croisse de manière exponentielle avec la latence et la profondeur, la génération de cette fonction termine même dans les cas extrêmes.

		Temps de calcul (ms)					
Application	h	$\Delta = 2$	$\Delta = 4$	$\Delta = 8$	$\Delta = 16$	$\Delta = 32$	$\Delta = 64$
Prodmat	1	2	2	3	3	3	2
	2	9	11	20	82	657	7350
	3	23	45	222	2089	27250	469139
BBFIR	1	3	4	3	3	3	3
	2	34	74	189	1057	15013	362109
Jacobi	1	1	1	1	1	1	1
	2	7	11	52	442	5156	78068
	3	11	20	80	603	6478	92093
FW	1	1	1	1	1	1	1
	2	6	10	42	358	4335	66184
	3	8	15	58	465	5336	76615
QRC	1	1	1	1	1	1	1
	2	7	9	18	76	615	7019
	3	11	15	48	413	5237	80844

TABLE 6.2 – Temps de calcul (Xeon à 2.4GHz) en millisecondes pour générer la fonction $next_{\mathcal{D}}^{\Delta}$ sur plusieurs applications, avec différentes valeurs de Δ et de h.

Afin de vérifier que l'approche reste applicable sur des exemples complexes (avec une latence élevée), l'algorithme 5 a été appliqué au noyau QRC, avec une fréquence cible de 200 MHz, avec un type de données sur 72 bits, et en appliquant le pipeline sur les trois niveaux de boucle. Pour

ce scénario, Catapult-C retourne une latence de 67 cycles, et l'algorithme 5 a construit l'ensemble des bulles en 25 minutes. Cela montre que l'approche, bien que coûteuse, reste réaliste.

6.3.3 Résultats quantitatifs

Avec l'insertion de bulles dans le domaine d'itération, le contrôle de la boucle devient plus complexe. En effet il faut ajouter des gardes aux instructions (pour ne pas les exécuter quand le contrôle visite les bulles), et des nouvelles contraintes pour les bornes des boucles. Ces gardes et contraintes additionnelles impliquent un surcoût matériel pour le contrôle dans la description matérielle générée par l'outil de HLS. Ce surcoût peut réduire la fréquence maximale atteignable par l'implémentation que l'outil de synthèse logique a produite.

Pour évaluer l'impact réel de l'insertion de bulles, le coût en surface matérielle, ainsi que le temps d'exécution des noyaux de calculs ont été estimés. Le matériel a été généré par Catapult-C en utilisant les directives adéquates pour ignorer les dépendances connues comme étant des faux positifs.

Surcoût en surface matérielle

Le tableau 6.3 fournit une évaluation du coût en ressources matérielles et en fréquence maximale de l'implémentation produite par l'outil de synthèse logique. Pour chaque taille de problème, quatre versions sont considérées :

- *RHLS 1D* correspond à la génération de matériel par Catapult-C, lorsque la boucle la plus interne est pipelinée.
- *PBI 2D V1* représente l'algorithme 4 appliqué aux deux niveaux de boucles les plus internes.
- De la même manière, *PBI 2D V2* représente l'algorithme 5 pour les deux boucles les plus internes.
- *PBI 3D V2* montre les résultats obtenus en appliquant l'algorithme 5 à l'ensemble du nid de boucles.

Les résultats pour le pipeline de nid de boucles de Catapult-C ne sont pas présentés, car soit il n'est pas capable de l'appliquer, soit l'implémentation est fausse (cf. section 6.3.2). Pour chaque version, le tableau 6.3 affiche une estimation du coût en surface matérielle en termes de :

- ALUT : *Adaptive Look-Up Table*, c'est-à-dire l'opérateur logique de base dans la plupart des FPGA ;
- REG : *Register*, l'élément mémoire de base dans la plupart des FPGA ;
- DSP : *Digital Signal Processing blocks* : des macro-opérateurs utilisés dans beaucoup d'applications de traitement du signal, offrant une meilleure densité qu'une implémentation avec des ALUT ;
- Freq. : La fréquence maximale estimée par l'outil de synthèse logique.

Le surcoût en surface matérielle lors de l'insertion de bulles est modéré (moins de 25%), excepté pour FW, pour lequel le coût total double. Cette exception s'explique par deux raisons :

Application	Version	**Caractéristiques matérielles**			
		ALUT	REG	DSP	Freq. (MHz)
Prodmat	RHLS 1D	489	215	4	272
	PBI 2D V1	629	228	4	235
	PBI 2D V2	553	198	4	246
	PBI 3D V2	559	226	4	231
BBFIR	RHLS 1D	553	152	4	185
	PBI 2D V1	727	231	4	213
	PBI 2D V2	649	241	4	241
FW	RHLS 1D	383	74	0	271
	PBI 2D V1	951	108	0	159
	PBI 2D V2	859	87	0	210
	PBI 3D V2	955	99	0	180
Jacobi	RHLS 1D	1012	845	8	164
	PBI 2D V1	1153	936	8	172
	PBI 2D V2	1226	948	8	168
	PBI 3D V2	1417	975	8	172
QRC	RHLS 1D	5375	2461	24	155
	PBI 2D V1	5792	2755	28	158
	PBI 2D V2	5684	2745	28	155
	PBI 3D V2	5772	2730	28	152

TABLE 6.3 – Caractéristiques matérielles des implémentations du pipeline de nid de boucles (PBI 2D V1, PBI 2D V2 et PBI 3D V2) comparées au pipeline de la boucle la plus interne uniquement (RHLS 1D).

– Les opérations de FW n'impliquent que des additions ou comparaisons sur des entiers, qui ont un coût équivalent au contrôle. Par conséquent l'insertion de bulles a relativement plus d'impact que sur les autres benchmarks dans lesquels les opérations impliquent des multiplicateurs.

– Les dépendances portées par les boucles à pipeliner mènent à un ensemble de bulles complexe, qui ajoute un nombre important de gardes dans le code du contrôle.

La fréquence des implémentations matérielles générées est en général plus faible lorsque le domaine d'itération est complexe, en comparaison avec le domaine d'itération original (pour Prodmat et FW), ou équivalent lorsque le domaine des bulles est relativement simple (QRC et Jacobi). Ce comportement était attendu, puisque les contraintes ou gardes additionnelles allongent le chemin critique du contrôle.

Pour des raisons inexpliquées, Catapult-C génère des circuits qui atteignent des fréquences plus élevées lorsque les deux boucles les plus internes sont pipelinées dans le noyau BBFIR, alors que le domaine des bulles est relativement complexe.

Estimation des performances

Le tableau 6.4 affiche le nombre de cycles requis pour exécuter les noyaux de calculs pipelinés, et fournit le temps d'exécution en fonction de la fréquence maximum atteignable, donnée dans le tableau 6.3. Pour chaque noyau de calculs, deux tailles de problème ont été considérées.

Comme le montre la colonne # Cycles, le nombre de cycles est toujours plus petit lorsque le pipeline de nid de boucles est appliqué, car les bulles insérées représentent moins de cycles d'attente que le vidage complet du pipeline. Cependant, alors que la taille du problème croît et que le nombre d'itérations des boucles internes augmente, l'impact du vidage diminue. Par conséquent, la réduction du nombre de cycles ne compense pas toujours la baisse de fréquence et le surcoût en surface matérielle.

On remarquera que l'algorithme 5 (PBI 2D V2) insère toujours moins de bulles que l'algorithme 4 (PBI 2D V1), ou au pire autant (Prodmat), ce qui n'affecte pas le coût en ressources matérielles.

Pour QRC avec une taille de problème très petite (3^3) le gain est relativement faible. La raison vient du fait que le nombre d'itérations est petit comparé à la latence. Par conséquent, le nombre de bulles insérées est comparable au nombre de cycles qu'aurait nécessité un vidage complet du pipeline.

Les résultats pour les problèmes de grande taille montrent qu'en général, comme prévu, le pipeline de nid de boucles atteint son efficacité maximale lorsque le nombre d'itérations est comparable à la latence du pipeline.

6.4 Conclusion

Les travaux de cette thèse ont été étroitement liés au développement du compilateur source-à-source GeCoS, maintenu par l'équipe Cairn. Les implémentations réalisées grâce aux techniques d'ingénierie dirigée par les modèles montrent la pertinence de telles approches dans la mise en œuvre de compilateurs. De plus la compilation dirigée par les scripts en plus des annotations, proche des scripts utilisés dans Pips [123] et des aspects [130] proposés dans Lara [131], permet de contrôler précisément les transformations des programmes.

Dans un premier temps, `ompVerify` a permis la validation du flot polyédrique et de l'implémentation des techniques d'analyse de dépendances et de vérification des ordonnancements parallèles. Par la suite l'environnement de transformation de boucles, ainsi que l'insertion de bulles polyédriques pour assurer la légalité du pipeline de nids de boucles, ont été implémentés en ciblant plus particulièrement la synthèse de haut niveau.

Les résultats obtenus confirment l'intérêt des transformations source-à-source dans le cadre d'un flot de compilation optimisante, en particulier pour la synthèse de haut niveau. Les travaux présentés dans les chapitres 4 et 5 et implémentés dans GeCoS étendent l'applicabilité du pipeline de nids de boucles à une classe plus grande de programmes (les Scop), et donnent des résultats prometteurs dans les cas où la boucle interne itère sur un petit domaine d'itération (jusqu'à 45% de réduction du nombre de cycles, avec un surcoût matériel modéré). Le surcoût en termes de ressources matérielles et la baisse de fréquence impliqués par les gardes introduites pourraient

tout de même être réduits en améliorant la technique de génération de contrôle.

Application	Taille du Problème	**Performances**			
		Version	# Cycles	Temps (ns)	Ratio
Prodmat ($\Delta = 4$)	4^3	RHLS 1D	157	577	
		PBI 2D V1	89	378	1.52
		PBI 2D V2	89	361	1.59
		PBI 3D V2	68	294	1.96
	128^3	RHLS 1D	2179456	8012709	
		PBI 2D V1	2097921	8927323	0.89
		PBI 2D V2	2097921	8528134	0.93
		PBI 3D V2	2097156	9078597	0.88
BBFIR ($\Delta = 4$)	256×8	RHLS 1D	3336	18032	
		PBI 2D V1	2552	11981	1.50
		PBI 2D V2	2030	9279	1.94
	1024×32	RHLS 1D	37548	202962	
		PBI 2D V1	34412	161558	1.25
		PBI 2D V2	32282	148050	1.37
FW ($\Delta = 3$)	16^3	RHLS 1D	6625	24446	
		PBI 2D V1	4178	26276	0.93
		PBI 2D V2	4169	19852	1.23
		PBI 3D V2	4107	22816	1.07
	128^3	RHLS 1D	2260737	8342202	
		PBI 2D V1	2097682	13192968	0.63
		PBI 2D V2	2097673	9988919	0.83
		PBI 3D V2	2097163	11650905	0.71
Jacobi ($\Delta = 8$)	30×16	RHLS 1D	13261	80859	
		PBI 2D V1	10981	63843	1.26
		PBI 2D V2	7831	46613	1.73
		PBI 3D V2	7568	44000	1.83
	30×256	RHLS 1D	2072461	12636957	
		PBI 2D V1	1941421	11287331	1.11
		PBI 2D V2	1937431	11532327	1.09
		PBI 3D V2	1937168	11262604	1.12
QRC ($\Delta = 13$)	3^3	RHLS 1D	90	580	
		PBI 2D V1	89	563	1.03
		PBI 2D V2	82	529	1.09
		PBI 3D V2	81	532	1.08
	32^3	RHLS 1D	23406	151006	
		PBI 2D V1	28670	181455	0.83
		PBI 2D V2	17464	112670	1.34
		PBI 3D V2	17610	115855	1.30

TABLE 6.4 – Performances des implémentations pipelinées des noyaux de calculs, avec les caractéristiques matérielles décrites dans le tableau 6.3, pour différentes tailles de problème.

Conclusion

Pour réaliser des systèmes embarqués performants, il est désormais indispensable de faire appel à des architectures hétérogènes incluant des accélérateurs matériels dédiés. Mais le temps de conception d'un accélérateur peut être très important, et l'utilisation d'outils de HLS est donc une nécessité. Malgré les progrès réalisés ces dernières années, les outils actuels sont le plus souvent incapables d'exploiter le parallélisme disponible dans la spécification en C/C++ de l'application.

Dans cette thèse, nous avons mis en œuvre un flot de compilation source-à-source dont l'objectif est de transformer les nids de boucles de manière à mettre en avant des structures de contrôle que les outils de HLS sont capables d'exploiter pour obtenir un accélérateur matériel plus performant. Les approches ont été mises au point dans un contexte de génération de matériel, et ne sont pas forcément pertinentes dans le cadre de la compilation pour le HPC (*High-Performance Computing* : Calcul Haute Performance).

Contributions

En nous basant sur le modèle polyédrique, nous avons ainsi amélioré l'applicabilité du pipeline de nids de boucles grâce aux trois contributions suivantes.

Analyse de la légalité du pipeline de nids de boucles

Le pipeline de boucles est une transformation essentielle en HLS, car elle permet d'exploiter le parallélisme présent dans les applications pour un coût modéré en termes de ressources matérielles. Les gains en termes de vitesse d'exécution sont importants lorsque le nombre d'itérations de la boucle est grand, mais le temps passé à vider le pipeline devient prépondérant lorsque le nombre d'itérations est comparable à la latence du pipeline. Pour limiter l'impact du vidage sur les performances lorsque le nombre d'itérations est petit, une technique efficace consiste à appliquer le pipeline sur plusieurs niveaux de boucles.

Appliquer le pipeline de nids de boucles consiste à entrelacer l'exécution de deux itérations consécutives des boucles externes, par exemple en remplissant le pipeline pour l'itération $n + 1$ pendant que l'itération n est en train de se vider. Ce mode d'exécution doit cependant prendre en compte les dépendances de données qui existent entre les différentes itérations des boucles externes, en plus de celles liées à la boucle interne.

Dans cette thèse nous avons proposé un nouveau test qui permet de s'assurer que l'application du pipeline de nids de boucles est légale. Ce test est applicable à une classe de programmes plus grande que les approches connues jusqu'alors, à savoir les nids de boucles représentables dans le modèle polyédrique. Le test a été implémenté dans le compilateur GeCoS développé dans l'équipe Cairn, et les résultats ont montré qu'il est réalisable dans un temps acceptable.

Ce premier résultat a montré la pertinence des analyses de haut niveau, en particulier celles basées sur la représentation des nids de boucles dans le modèle polyédrique, pour résoudre des problèmes liés à la synthèse de haut niveau. Bien entendu, bien que présentée dans le cadre du pipeline pour la synthèse de haut niveau, nous pensons que cette technique peut être adaptée facilement à la compilation logicielle.

Correction par insertion de bulles polyédriques

Lorsque le pipeline de nids de boucles n'est pas légal, les outils de HLS interdisent simplement l'application de la transformation. Pourtant nous avons montré dans le chapitre 5 que cette interdiction peut être levée par l'introduction d'états d'attente dans le pipeline, des bulles.

Nous avons mis au point une technique de correction qui insère à la compilation de telles bulles dans le domaine d'itération des nids de boucles. De cette manière, les dépendances initialement violées par l'application du pipeline de nids de boucles sont respectées après insertion de bulles. Cette technique de correction est appliquée directement après l'analyse de légalité, et s'applique aux nids de boucles représentables dans le modèle polyédrique. L'insertion est réalisée en un temps acceptable (pour les premières versions) dans le contexte d'un flot de compilation source-à-source pour la synthèse de haut niveau.

Lors des expérimentations, les résultats ont montrés que l'insertion de bulles polyédriques dans les domaines d'itérations produit un code de contrôle plus complexe, en particulier en termes de nombre de gardes. Ces nouvelles gardes augmentent les coûts en ressources matérielles après synthèse, allant jusqu'à un surcoût de 50% lorsque les instructions impliquent des opérateurs peu coûteux en matériel. En revanche lorsque les opérateurs sont coûteux et que le domaine des bulles insérées implique peu de gardes, le surcoût est relativement faible.

Comparé au simple pipeline de la boucle la plus interne, le pipeline de nids de boucles avec insertion de bulles s'exécute toujours en un nombre de cycles machine plus faible. Cependant, à cause des gardes introduites pour l'insertion des bulles, le chemin critique du contrôle s'allonge, et la fréquence diminue. Par conséquent, en terme de vitesse d'exécution, l'approche n'est intéressante que lorsque la baisse de fréquence est compensée par la baisse du nombre de cycles, ce qui est le cas lorsque le nombre d'itérations de la boucle interne est petit.

Génération de contrôle pour le parcours de polyèdres

La technique de parcours de polyèdres basée sur l'approche de QUILLERÉ et coll. implémentée dans CLOOG permet d'obtenir un code dans lequel le nombre de gardes est réduit. Bien qu'offrant des performances intéressantes dans le contexte de la compilation logicielle, le code généré par CLOOG multiplie les boucles, ce qui le rend mal adapté à la HLS. Un code basé sur une machine à états est au contraire mieux adapté à la génération de matériel. C'est la raison pour laquelle

nous avons implémenté la technique proposée par Boulet et Feautrier [5] dans le compilateur source-à-source Gecos.

Le code généré par l'approche de Boulet et Feautrier parcourt le domaine d'itération de manière exacte en construisant une fonction (appelée *next*) qui permet de déterminer le successeur immédiat d'une itération. Le contrôleur sous la forme d'une machine à états offre l'avantage de mettre à plat les nids de boucles, ce qui facilite l'application de pipeline de nids de boucles par les outils de Hls (lorsque la transformation est légale).

Une fois les nids de boucles aplatis sous forme de machine à états, les outils de Hls ne sont plus capables de les analyser précisément. Par conséquent, les techniques de réduction de force et d'analyse de type ne sont plus appliquées efficacement. À partir de la représentation polyédrique des nids de boucles, il est possible d'appliquer directement ces analyses lors du flot source-a-source.

Perspectives à court terme

L'analyse de la légalité du pipeline de nids de boucles permet de savoir de manière exacte si le pipeline de nid de boucles introduit des violations de dépendances. Les résultats ont montré que cette analyse peut être exécutée en un temps acceptable. Cependant ce temps d'analyse dépend de la latence du pipeline Δ : il faut en effet construire la fonction *next* à l'ordre Δ, opération dont le temps de calcul croit exponentiellement avec Δ. Afin de réduire ce temps d'analyse, il est possible de calculer la fonction *next* sur une fraction seulement du domaine original. Ceci permettrait de ne calculer la fonction *next* que pour une fraction de Δ. Le résultat serait une approximation prudente de la fonction *next* sur le domaine original, mais serait beaucoup plus rapide à construire, en particulier lorsque Δ est grand.

Les techniques d'insertion de bulles présentées dans le chapitre 4 proposent différentes façons de corriger un pipeline de nid de boucles illégal. Le temps d'exécution de ces techniques est relativement faible comparé au temps nécessaire à l'analyse, excepté pour l'approche itérative. Dans tous les cas, le nombre bulles insérées est moins élevé que les approches précédentes. Il reste néanmoins à déterminer une approche insérant le nombre minimal de bulles, et à placer ces bulles de façon à obtenir le contrôle le plus simple possible.

Un domaine des bulles insérées qui est défini par un grand nombre de contraintes résulte en contrôle matériel coûteux. Il serait intéressant de rechercher un compromis entre le nombre de bulles insérées, le placement de ces bulles dans le domaine d'itération, et le nombre de gardes impliquées dans la définition du domaine des bulles. On pourrait ainsi éviter une baisse de la fréquence d'exécution du matériel, et mieux en maîtriser les performances.

Une autre perspective serait d'examiner la génération de code. La technique de parcours de polyèdres proposée par Boulet et Feautrier peut être améliorée en utilisant non seulement le domaine de validité des transitions, mais aussi l'état d'activité des instructions. En se servant de cette activité comme contexte, il est possible d'utiliser l'opération *gist* sur les domaines de validité des transitions et les domaines d'iteration des instructions afin de réduire le nombre de contraintes à évaluer. De plus cette approche permettrait de partitionner la machine à états, et

appliquer la fusion de chemin de données (cf. section 2.3.1) à ces différentes partitions pourrait conduire à un contrôle moins coûteux en ressources matérielles.

La fonction *next* à l'ordre Δ peut avoir un intérêt en dehors de la vérification de la légalité du pipeline de nid de boucles. Connaître à l'avance l'état du programme Δ itérations plus tard permet par exemple de savoir quelles données seront utilisées, et d'anticiper la lecture des données en mémoire principale.

Les techniques présentées dans cette thèse sont limitées aux programmes représentables à l'aide du modèle polyédrique. Pour pouvoir appliquer le pipeline de boucles lorsque le programme ne rentre pas dans cette classe de programme, nous avons mis au point une technique qui assure dynamiquement la légalité du pipeline [AMD13]. Inspirée des mécanismes de désambiguïsation mémoire mis en œuvre dans les processeurs superscalaires, cette technique consiste à conserver dans un banc de registres les adresses des valeurs qui ont été modifiées durant le pipeline. Lorsqu'une adresse est lue alors qu'elle est présente dans le banc de registres, il suffit d'insérer dynamiquement une bulle. Cette technique a été mise au point récemment, c'est la raison pour laquelle elle n'est pas présentée dans ce manuscrit.

Perspectives à moyen terme

Comme la désambiguïsation mémoire, plusieurs mécanismes dynamiques mis en œuvre dans les processeurs superscalaires pourraient être utilisés lors du processus de synthèse pour obtenir des accélérateurs plus performants. Par exemple, des techniques de spéculation ont été utilisées dans le cadre d'une implémentation sur Fpga d'algorithmes de bio-informatique [132], conduisant à des gains significatifs en performance. Formaliser ces mécanismes en termes de transformations de programmes dans un flot source-à-source pour la Hls, et estimer leur impact (le compromis surcoût en ressources matérielles / gains en performances) sur l'accélérateur matériel produit reste un sujet ouvert.

On sait que l'écart séparant les performances des mémoires de celles des unités de calcul n'a cessé de croître durant les vingt dernières années [133]. Dans le cadre de la synthèse de matériel, il est possible de transformer l'allocation mémoire ainsi que les mémoires elles-mêmes pour réduire cet écart. Les recherches actuelles [46] se concentrent sur les communications entre l'accélérateur et la mémoire principale pour y parvenir. On peut noter que le parallélisme multiplie le nombre d'accès simultanés aux mémoires locales, et augmente considérablement la bande passante requise pour être exploité pleinement. Une solution pour augmenter la bande passante serait de multiplier les mémoires locales en les partitionnant en plusieurs bancs, et en insérant des mécanismes de gestion de l'intégrité si nécessaire. Des recherches récentes [45] proposent des approches pour partitionner les mémoires automatiquement, mais il reste à les généraliser aux Scop.

Perspectives à long terme

Sauf avancée technologique majeure, la proportion de *dark silicon* dans les circuits électroniques risque de continuer à croître. Comme le précise Taylor [1], une bonne façon d'obtenir des

circuits performants consiste à améliorer la densité en spécialisant le matériel. Cette spécialisation va conduire à des architectures très hétérogènes, disposant d'accélérateurs à très gros grain. Ces architectures nécessiteront des outils capables d'exploiter pleinement cette spécialisation, à la manière des CGRA (*Coarse-Grain Reconfigurable Architecture* : Architecture Reconfigurable à Gros Grain).

De plus, les communications entre accélérateurs spécialisés risquent de devenir déterminantes pour les performances. Or les études ont montré que la différence entre les performances des unités de calcul et celles des mémoires ne cesse de croître [133]. Une approche consiste à rapprocher le calcul de la mémoire, et éviter le concept de mémoire globale, en suivant une approche similaire aux IMEM (*Intelligent MEMories* : Mémoires Intelligentes) [134]. Cette approche pourrait aussi s'appuyer sur le 3D *stacking* [135, 136] qui permet de produire des circuits sur trois dimensions au lieu de deux, en alternant une couche de logique avec une couche de mémoire par exemple. Rapprocher le calcul de la mémoire remettrait alors en cause les compromis qui sont faits aujourd'hui entre localité et parallélisme.

Ces évolutions technologiques et architecturales impliquent une évolution des outils. D'un côté les outils doivent être capables de spécialiser le matériel efficacement et rapidement, tout en prenant en compte les évolutions technologiques (3D stacking par exemple). D'un autre côté, ils doivent permettre aux applications logicielles d'exploiter pleinement ces nouvelles architectures, en révisant le compromis localité / parallélisme. Enfin, ces évolutions amènent des questions concernant l'apprentissage, et l'expertise nécessaire aux concepteurs pour développer des applications logicielles s'exécutant sur ces architectures. Un environnement de développement fournissant un outillage qui facilite l'analyse de code, tant en termes de qualité que de performances, de manière à fournir des retours rapides et précis aux développeurs, permettrait de réduire les prérequis.

Glossaire

Acl *Affine Control Loops* : Boucles à Contrôle Affine.

Cdfg *Control DataFlow Graph* : Graphe de Contrôle et de Flot de Données.

Cgra *Coarse-Grain Reconfigurable Architecture* : Architecture Reconfigurable à Gros Grain.

Emf *Eclipse Modeling Framework*.

Fpga *Field Programmable Gate Array* : Réseau de Portes Programmables.

Hdl *Hardware Description Language* : Langage de Description Matérielle.

Hls *High-Level Synthesis* : Synthèse de Haut Niveau.

Hpc *High-Performance Computing* : Calcul Haute Performance.

Ip *semiconductor Intellectual Property core/block* : bloc IP matériel.

Kpn Kahn *Process Network* : Réseau de Processus de Kahn.

Mde *Model Driven Engineering* : Ingénierie Dirigée par les Modèles.

Ppn *Polyhedral Process Network* : Réseau de Processus Polyédriques.

Prdg *Polyhedral Reduced Dependence Graph* : Graphe des dépendances polyédriques réduites.

Ram *Random-Access Memory* : Mémoire à Accès Aléatoire.

Rtl *Register Transfert Level* : Niveau de Transferts de signaux entre des Registres.

Vhdl *Very-high-speed integrated circuits Hardware Description Language*.

Vliw *Very Long Instruction Word* : Mot d'Instruction Très Long.

Wcet *Worst Case Execution Time* : Temps d'Exécution dans le Pire Cas.

Asic *Application-Specific Integrated Circuit* : Circuit Intégré Spécifique à une Application.

Asip *Application-Specific Instruction-set Processor* : Processeur à jeu d'Instructions Spécialisées pour une Application.

Ddr-Sdram *Double Data Rate Synchronous Dynamic RAM*.

Dag *Directed Acyclic Graph* : Graphe Acyclique Orienté.

Dsp *Digital Signal Processing* : Traitement du Signal Numérique.

MOPS *Mega Operations Per Second* : Million d'Opérations Par Seconde.

Quast *Quasi-Affine Selection Tree* : Arbre de sélection quasi-affine.

SDRAM *Synchronous Dynamic Random-Access Memory* : Mémoire Dynamique Synchrone à Accès Aléatoire.

Sare *System of Affine Recurrence Equation* : Système d'Équations Affines Récurrentes.

Scop *Static Control Programs* : Programmes à Contrôle Statique.

Soc *System on Chip* : Système sur Puce.

eCPU *Embedded Central Process Unit* : Processeur Programmable Embarqué.

Publications personnelles

[AMD13] Mythri Alle, Antoine Morvan et Steven Derrien, « Runtime Dependency Analysis for Loop Pipelining in High-Level Synthesis », *in Proc. of the 2013 Design Automation Conference*, DAC'2013, 2013. Accepté pour publication.

[BYR+11] Vamshi Basupalli, Tomofumi Yuki, Sanjay V. Rajopadhye, Antoine Morvan, Steven Derrien, Patrice Quinton et David Wonnacott, « `ompVerify` : Polyhedral Analysis for the OpenMP Programmer », *in Proc. of the 7th international conference on OpenMP in the Petascale era*, IWOMP'11, p. 37-53, 2011.

[MDQ11] Antoine Morvan, Steven Derrien et Patrice Quinton, « Efficient Nested Loop Pipelining in High Level Synthesis using Polyhedral Bubble Insertion », *in Proc. of the International Conference on Field Programmable Technologies*, FPT'11, p. 1-10, 2011.

[MDQ13] Antoine Morvan, Steven Derrien et Patrice Quinton, « Polyhedral Bubble Insertion : A Method to Improve Nested Loop Pipelining for High-Level Synthesis », *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 32, n° 3, p. 339-352, 2013.

Bibliographie

[1] Michael B. Taylor, « Is Dark Silicon Useful ? Harnessing the Four Horsemen of the Coming Dark Silicon Apocalypse », *in Proc. of the 49th Annual Design Automation Conference*, DAC'12, p. 1131-1136, 2012.

[2] Uday Bondhugula, Albert Hartono, J. Ramanujam et P. Sadayappan, « A Practical Automatic Polyhedral Parallelizer and Locality Optimizer », *in Proc. of Programming Language Design and Implementation*, PLDI'08, p. 101-113, 2008.

[3] Sang-Ik Lee, Troy A. Johnson et Rudolf Eigenmann, « Cetus – An Extensible Compiler Infrastructure for Source-to-Source Transformation », *in Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing*, vol. 2958 *in Lecture Notes in Computer Science*, p. 539-553, 2004.

[4] Daniel J. Quinlan, « ROSE : Compiler Support for Object-Oriented Frameworks », *Parallel Processing Letters*, vol. 10, n° 2, p. 215-226, 2000.

[5] Pierre Boulet et Paul Feautrier, « Scanning Polyhedra without Do-loops », *in Proc. of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT'98, p. 4-11, 1998.

[6] Cairn EPI, « The GeCoS (Generic Compiler Suite) Source-to-Source Compiler Infrastructure ». En ligne : http://gecos.gforge.inria.fr/.

[7] Cédric Bastoul, « Code Generation in the Polyhedral Model Is Easier Than You Think », *in Proc of the 13th International Conference on Parallel Architectures and Compilation Techniques*, PACT '04, p. 7-16, 2004.

[8] Gary J. Sullivan, Jens-Rainer Ohm, Woojin Han et Thomas Wiegand, « Overview of the High Efficiency Video Coding (HEVC) Standard », *IEEE Trans. on Circuits and Systems for Video Technology*, vol. 22, n° 12, p. 1649-1668, 2012.

[9] Pedro F. Felzenszwalb et Daniel P. Huttenlocher, « Efficient Belief Propagation for Early Vision », *International Journal of Computer Vision*, vol. 70, n° 1, p. 41-54, oct. 2006.

[10] Glenn K. Manacher, « Production and Stabilization of Real-Time Task Schedules », *Journal of the ACM*, vol. 14, n° 3, p. 439-465, juil. 1967.

[11] Ian Kuon et Jonathan Rose, « Measuring the gap between FPGAs and ASICs », *in Proc. of the 2006 ACM/SIGDA 14th international symposium on Field programmable gate arrays*, FPGA'06, p. 21-30, 2006.

[12] ARM, « Cortex-A9 Processor ». En ligne : http://www.arm.com/products/processors/cortex-a/cortex-a9.php.

[13] STMicroelectronics, « Sthorm - STMicroelectronics Many-Core Embedded Systems Architecture ». En ligne : https://community.cmc.ca/community/sthorm.

[14] Antoine Floc'h, *Compilation optimisante pour processeurs extensibles*. Thèse de doctorat, Unniversité de Rennes 1, France, juin 2012.

[15] Emmanuel Casseau, François Charot, Antoine Floc'h, Shafqat Khan, Daniel Ménard, Olivier Sentieys, Christophe Wolinski, Stéphane Guyetant, Stéphane Chevobbe, Arnaud Tisserand, Henk Heijnen, Jean-Pierre Le Glanic et Erwan Raffin, « ROMA Project intermediate progress report », rap. tech., Agence Nationale de la Recherche (ANR) - Irisa, Cea List, Lirmm, Thomson Silicon Components (TSC), 2008. En ligne : http://roma.irisa.fr.

[16] Erwan Raffin, *Déploiement d'applications multimédia sur architecture reconfigurable à gros grain : modélisation avec la programmation par contraintes*. Thèse de doctorat, Unniversité de Rennes 1, France, juil. 2011.

[17] Ganesh Venkatesh, Jack Sampson, Nathan Goulding, Saturnino Garcia, Vladyslav Bryksin, Jose Lugo-Martinez, Steven Swanson et Michael Bedford Taylor, « Conservation Cores : Reducing the Energy of Mature Computations », *ACM SIGARCH Computer Architecture News*, vol. 38, n° 1, p. 205-218, mars 2010.

[18] Gordon E. Moore, « Cramming More Components onto Integrated Circuits », *Electronics*, vol. 38, n° 8, p. 114-117, avril 1965.

[19] Marc Duranton, David Black-Schaffer, Sami Yehia et Koen de Bosschere, « Computing Systems : Research Challenges Ahead : The HiPEAC Vision », rap. tech., Uppsala University, 2011.

[20] Thorsten Grotker, Stan Liao, Grant Martin et Stuart Swan, *System Design with SystemC*. Norwell, MA, USA : Kluwer Academic Publishers, 2002.

[21] Krishna K. Singh et Gayatri Agnihotri, *System design through MATLAB, Control Toolbox and SIMULINK*. Springer, 2001.

[22] Johan Eker et Jörn W. Janneck, « CAL Language Report », rap. tech. UCB/ERL M03/48, University of California at Berkeley, 2003.

[23] Christopher Hylands, Edward Lee, Jie Liu, Xiaojun Liu, Stephen Neuendorffer, Yuhong Xiong, Yang Zhao et Haiyang Zheng, « Overview of the Ptolemy Project », rap. tech. UCB/ERL M03/25, EECS Department, University of California, Berkeley, 2003.

[24] Giovanni De Micheli et Rajesh K. Gupta, « Hardware/Software Co-Design », *IEEE Micro*, vol. 85, n° 3, p. 349-365, 1997.

[25] Jürgen Teich, « Hardware/Software Codesign : The Past, the Present, and Predicting the Future », *Proceedings of the IEEE*, vol. 100, p. 1411-1430, 2012.

[26] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann,

Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat et Per Stenström, « The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools », *ACM Trans. on Embedded Computing Systems*, vol. 7, n° 3, p. 36:1-36:53, avril 2008.

[27] Grant Martin et Gary Smith, « High-Level Synthesis : Past, Present, and Future », *IEEE Design & Test of Computers*, vol. 26, n° 4, p. 18-25, juil. 2009.

[28] Philippe Coussy et Adam Morawiec, *High-Level Synthesis : from Algorithm to Digital Circuit*. Springer, 1re éd., 2008.

[29] Philippe Coussy, Daniel D. Gajski, Michael Meredith et Andres Takach, « An Introduction to High-Level Synthesis », *IEEE Design & Test of Computers*, vol. 26, n° 4, p. 8-17, 2009.

[30] Alex Orailoglu et Daniel D. Gajski, « Flow graph representation », *in Proc. of the 23rd ACM/IEEE Design Automation Conference*, DAC'86, p. 503-509, 1986.

[31] Chia-Jeng Tseng et Daniel P. Siewiorek, « Automated Synthesis of Data Paths in Digital Systems », *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 5, n° 3, p. 379-395, 1986.

[32] Catherine H. Gebotys et Mohamed I. Elmasry, « VLSI Design Synthesis with Testability », *in Proc. of the 25th ACM/IEEE Design Automation Conference*, DAC'88, p. 16-21, 1988.

[33] Thomas L. Adam, Kanianthra Mani Chandy et J. R. Dickson, « A Comparison of List Schedules for Parallel Processing Systems », *Communications of the ACM*, vol. 17, n° 12, p. 685-690, déc. 1974.

[34] Pierre G. Paulin et John P. Knight, « Force-Directed Scheduling for the Behavioral Synthesis of ASICs », *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 8, n° 6, p. 661-679, 1989.

[35] Krzysztof Kuchcinski, « Constraints-Driven Scheduling and Resource Assignment », *ACM Trans. on Design Automation of Electronic Systems*, vol. 8, n° 3, p. 355-383, juil. 2003.

[36] Nahri Moreano, Edson Borin, Cid C. de Souza et Guido Araujo, « Efficient Datapath Merging for Partially Reconfigurable Architectures », *IEEE Trans. on CAD of Integrated Circuits and Systems*, vol. 24, n° 7, p. 969-980, juil. 2005.

[37] Christophe Wolinski, Krzysztof Kuchcinski, Erwan Raffin et François Charot, « Architecture-Driven Synthesis of Reconfigurable Cells », *in Proc. of the 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, DSD'2009, p. 531-538, 2009.

[38] Minjoong Rim, Rajiv Jain et Renato De Leone, « Optimal Allocation and Binding in High-Level Synthesis », *in Proc. of the 29th ACM/IEEE Design Automation Conference*, DAC'92, p. 120-123, 1992.

[39] Sharad Sinha, Udit Dhawan, Siew Kei Lam et Thambipillai Srikanthan, « A Novel Binding Algorithm to Reduce Critical Path Delay During High Level Synthesis », *in Proc. of the 2011 IEEE Computer Society Annual Symposium on VLSI*, ISVLSI'11, p. 278-283, 2011.

[40] David F. Bacon, Susan L. Graham et Oliver J. Sharp, « Compiler Transformations for High-Performance Computing », *ACM Computing Surveys*, vol. 26, n° 4, p. 345-420, déc. 1994.

[41] Florent de Dinechin, « Multiplication by Rational Constants », *IEEE Trans. on Circuits and Systems, II*, vol. 59, n° 2, p. 98-102, fév. 2012.

[42] Jean-Michel Muller, Arnaud Tisserand, Benoît de Dinechin et Christophe Monat, « Division by Constant for the ST100 DSP Microprocessor », *in Proc. 17th Symposium on Computer Arithmetic*, ARITH'05, p. 124-130, juin 2005.

[43] Sumit Gupta, Miguel Miranda, Francky Catthoor et Rajesh Gupta, « Analysis of High-level Address Code Transformations for Programmable Processors », *in Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, DATE'2000, p. 9-13, 2000.

[44] Margaret Martonosi, Anoop Gupta et Thomas E. Anderson, « Tuning Memory Performance of Sequential and Parallel Programs », *Computer*, vol. 28, n° 4, p. 32-40, avril 1995.

[45] Peng Li, Yuxin Wang, Peng Zhang, Guojie Luo, Tao Wang et Jason Cong, « Memory Partitioning and Scheduling Co-optimization in Behavioral Synthesis », *in Proc. of the International Conference on Computer-Aided Design*, ICCAD '12, p. 488-495, 2012.

[46] Christophe Alias, Alain Darte et Alexandru Plesco, « Optimizing DDR-SDRAM Communications at C-level for Automatically-Generated Hardware Accelerators. An Experience with the Altera C2H HLS Tool », *in Proc. of the 21st IEEE International Conference on Application-specific Systems Architectures and Processors*, ASAP'2010, p. 329-332, juil. 2010.

[47] Samuel Bayliss et George A. Constantinides, « Application Specific Memory Access, Reuse and Reordering for SDRAM », *in Proc. of the 7th International Conference on Reconfigurable Computing : Architectures, Tools and Applications*, ARC'11, p. 41-52, 2011.

[48] Daniel Menard, Romain Serizel, Romuald Rocher et Olivier Sentieys, « Accuracy Constraint Determination in Fixed-Point System Design », *EURASIP Journal of Embedded Systems*, vol. 2008, p. 1:1-1:12, jan. 2008.

[49] Éric Martin, Olivier Sentieys, Hélène Dubois et Jean-Luc Philippe, « GAUT : An Architectural Synthesis Tool for Dedicated Signal Processors », *in Proc. of the 1993 Design Automation Conference*, EURO-DAC'93, p. 14-19, 1993.

[50] Mentor Graphics, *Catapult C Synthesis user's and Reference Manual, University Version*, 2010.

[51] Cadence, « C-to-Silicon High-Level Synthesis Tool ». En ligne : http://www.cadence.com/products/sd/silicon_compiler/.

[52] Xilinx, « AutoESL Design Technologies ». En ligne : http://www.autoesl.com/.

[53] Impulse Accelerated, « Impulse-C High-Level Synthesis Tool ». En ligne : http://www.impulseaccelerated.com/.

[54] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H. Anderson, Stephen Brown et Tomasz Czajkowski, « LegUp : High-Level Synthesis

for FPGA-Based Processor/Accelerator Systems », *in Proc. of the 19th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA'11, p. 33-36, 2011.

[55] Cédric Bastoul, Albert Cohen, Sylvain Girbal, Saurabh Sharma et Olivier Temam, « Putting Polyhedral Loop Transformations to Work », *in Proc. of the 16th International Workshop on Languages and Compilers for Parallel Computing*, vol. 2958 *in Lecture Notes in Computer Science*, p. 209-225, 2004.

[56] Sven Verdoolaege et Tobias Grosser, « Polyhedral Extraction Tool », *in Proc of the 2nd International Workshop on Polyhedral Compilation Techniques*, IMPACT'2012, jan. 2012.

[57] Tobias Grosser, Hongbin Zheng, Raghesh Aloor, Andreas Simbürger, Armin Größlinger et Louis-Noël Pouchet, « Polly - Polyhedral optimization in LLVM », *in Proc. of the 1st International Workshop on Polyhedral Compilation Techniques*, IMPACT'2011, avril 2011.

[58] Corinne Ancourt et François Irigoin, « Scanning polyhedra with DO loops », *ACM SIGPLAN Notices*, vol. 26, n° 7, p. 39-50, juil. 1991.

[59] Hervé Le Verge, Vincent Van Dongen et Doran K. Wilde, « Loop Nest Synthesis using the Polyhedral Library », rap. tech. 2288, Irisa, 1994.

[60] Fabien Quilleré, Sanjay V. Rajopadhye et Doran K. Wilde, « Generation of Efficient Nested Loops from Polyhedra », *International Journal of Parallel Programming*, vol. 28, n° 5, p. 469-498, oct. 2000.

[61] Paul Feautrier, « Dataflow Analysis of Array and Scalar References », *International Journal of Parallel Programming*, vol. 20, n° 1, p. 23-53, 1991.

[62] Nicolas Vasilache, Cédric Bastoul, Albert Cohen et Sylvain Girbal, « Violated Dependence Analysis », *in Proc. of the 20th annual International Conference on Supercomputing*, ICS'06, p. 335-344, 2006.

[63] Paul Feautrier, « Parametric Integer Programming », *RAIRO Recherche opérationnelle*, vol. 22, n° 3, p. 243-268, 1988.

[64] Sven Verdoolaege, « ISL : An Integer Set Library for the Polyhedral Model », *in Proc. of the 3rd International Congress on Mathematical Software*, ICMS'10, p. 299-302, 2010.

[65] Paul Feautrier, « Some Efficient Solutions to the Affine Scheduling Problem. II. Multidimensional Time », *International Journal of Parallel Programming*, vol. 21, n° 6, p. 389-420, 1992.

[66] Cédric Bastoul, *Improving Data Locality in Static Control Programs.* Thèse de doctorat, Université Paris 6, Pierre et Marie Curie, France, déc. 2004.

[67] Jean-François Collard, Denis Barthou et Paul Feautrier, « Fuzzy Array Dataflow Analysis », *in Proc of the 5th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPOPP'95, p. 92-101, 1995.

[68] Denis Barthou, *Array Dataflow Analysis in Presence of Non-affine Constraints.* Thèse de doctorat, Université de Versailles St-Quentin, Versailles, France, fév. 1998.

[69] Mohamed-Walid Benabderrahmane, Louis-Noël Pouchet, Albert Cohen et Cédric Bastoul, « The Polyhedral Model is More Widely Applicable Than You Think », *in Proc. of the 19th joint European conference on Theory and Practice of Software, international conference on Compiler Construction*, ETAPS'10/CC'10, p. 283-303, 2010.

[70] Christophe Alias, Fabrice Baray et Alain Darte, « Bee+Cl@k : An Implementation of Lattice-Based Memory Reuse in the Source-to-Source Translator ROSE », *in Proc. of the 2007 ACM SIGPLAN/SIGBED conference on Languages, Compilers, and Tools for Embedded Systems*, LCTES'07, p. 73-82, juin 2007.

[71] Christophe Alias, Fabrice Baray et Alain Darte, « Lattice-Based Array Contraction : From Theory to Practice », rap. tech. 2007-44, École Normale Supérieure de Lyon, 2007.

[72] Michelle Mills Strout, Larry Carter, Jeanne Ferrante et Beth Simon, « Schedule-Independent Storage Mapping for Loop », *in Proc. of the 8th international conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS-VIII, p. 24-33, 1998.

[73] Christophe Mauras, Patrice Quinton, Sanjay V. Rajopadhye et Yannick Saouter, « Scheduling Affine Parameterized Recurrences by means of Variable Dependent Timing Functions », rap. tech. RR-1204, Inria, 1990.

[74] Paul Feautrier, « Some Efficient Solutions to the Affine Scheduling Problem. I. One-Dimensional Time », *International Journal of Parallel Programming*, vol. 21, n° 5, p. 313-347, 1992.

[75] Patrice Quinton, « Automatic Synthesis of Systolic Arrays from Uniform Recurrent Equations », *in Proc. of the 11th annual International Symposium on Computer Architecture*, ISCA'84, p. 208-214, 1984.

[76] Patrice Quinton et Vincent Van Dongen, « The Mapping of Linear Recurrence Equations on Regular Arrays », *Journal of VLSI Signal Processing*, vol. 1, p. 95-113, 1989.

[77] Sanjay V. Rajopadhye, S. Purushothaman et Richard M. Fujimoto, « On Synthesizing Systolic Arrays from Recurrence Equations with Linear Dependencies », *in Proc. of the 6th conference on Foundations of Software Technology and Theoretical Computer Science*, p. 488-503, déc. 1986.

[78] Sanjay V. Rajopadhye, *Synthesis, Optimization and Verification of Systolic Architectures.* Thèse de doctorat, University of Utah, Salt Lake City, Utah 84112, US, déc. 1986.

[79] Hervé Le Verge, « A note on Chernikova's Algorithm », rap. tech. RR-1662, Inria, 1992.

[80] Richard M. Karp, Raymond E. Miller et Shmuel Winograd, « The Organization of Computations for Uniform Recurrence Equations », *Journal of the ACM*, vol. 14, n° 3, p. 563-590, juil. 1967.

[81] Christophe Mauras, *Alpha : un langage équationnel pour la conception et la programmation d'architectures parallèles synchrones.* Thèse de doctorat, Université de Rennes 1, IFSIC, Rennes, France, déc. 1989.

[82] Anne-Claire Guillou, Fabien Quilleré, Patrice Quinton, Sanjay V. Rajopadhye et Tanguy Risset, « Hardware Design Methodology with the Alpha Language », *in Forum on Design Languages*, FDL'01, sept. 2001.

[83] Tomofumi Yuki, Gautam Gupta, DaeGon Kim, Tanveer Pathan et Sanjay V. Rajopadhye, « AlphaZ : A System for Design Space Exploration in the Polyhedral Model », *in Proc. of the 25th International Workshop on Languages and Compilers for Parallel Computing*, LCPC'2012, sept. 2012.

[84] Patrice Quinton et Yves Robert, *Systolic algorithms and architectures*. Prentice Hall, 1991.

[85] Marcus Bednara et Jürgen Teich, « Automatic Synthesis of FPGA Processor Arrays from Loop Algorithms », *The Journal of Supercomputing*, vol. 26, n° 2, p. 149-165, sept. 2003.

[86] Anne-Claire Guillou, Patrice Quinton et Tanguy Risset, « Hardware Synthesis for Multi-Dimensional Time », *in Proc. of the IEEE International Conference on Application-Specific Systems, Architectures, and Processors*, ASAP'2003, p. 40-50, juin 2003.

[87] Yiwan Wong et Jean-Marc Delosme, « Space-Optimal Linear Processor Allocation for Systolic Arrays Synthesis », *in Proc. of the 6th International Parallel Processing Symposium*, IPPS'92, p. 275-282, 1992.

[88] Bart Kienhuis, Edwin Rijpkema et Ed Deprettere, « Compaan : Deriving Process Networks from Matlab for Embedded Signal Processing Architectures », *in Proc. of the 8th International Workshop on Hardware/Software Codesign*, CODES'00, p. 13-17, 2000.

[89] Alexandru Turjan, *Compaan - a Process Network Parallelizing Compiler*. VDM Publishing, 2008.

[90] Sven Verdoolaege, « Polyhedral Process Networks », *in Handbook of Signal Processing Systems*, p. 931-965, 1re éd., 2010.

[91] Claudiu Zissulescu, Todor Stefanov, Bart Kienhuis et Ed Deprettere, « Laura : Leiden Architecture Research and Exploration Tool », *in 13th International Conference on Field Programmable Logic and Application*, vol. 2778 *in Lecture Notes in Computer Science*, p. 911-920, 2003.

[92] Alexandru Turjan, Bart Kienhuis et Ed Deprettere, « Classifying Interprocess Communication in Process Network Representation of Nested-Loop Programs », *ACM Trans. on Embedded Computing Systems*, vol. 6, n° 2, p. 1:1-1:29, mai 2007.

[93] Sven Verdoolaege, Rachid Seghir, Kristof Beyls, Vincent Loechner et Maurice Bruynooghe, « Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions », *Algorithmica*, vol. 48, n° 1, p. 37-66, mars 2007.

[94] Philippe Clauss, Federico Javier Fernández, Diego Garbervetsky et Sven Verdoolaege, « Symbolic Polynomial Maximization Over Convex Sets and Its Application to Memory Requirement Estimation », *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 17, n° 8, p. 983-996, août 2009.

[95] Alexandru Plesco, *Program Transformations and Memory Architecture Optimizations for High-Level Synthesis of Hardware Accelerators*. Thèse de doctorat, École normale supérieure de Lyon, 2010.

[96] Doran K. Wilde, « A Library for Doing Polyhedral Operations », rap. tech. 2157, Irisa, déc. 1993.

[97] David G. Wonnacott, « A Retrospective of the Omega Project », rap. tech. HC-CS-TR-2010-01, Haverford College Computer Science, juin 2010.

[98] Uday Bondhugula, « Automatic Distributed-Memory Parallelization and Code Generation using the Polyhedral Framework », rap. tech. ISc-CSA-TR-2011-3, Indian Institute of Science, sept. 2011.

[99] Wayne Kelly, William Pugh et Evan Rosser, « Code Generation for Multiple Mappings », *in Proc. of the 5th Symposium on the Frontiers of Massively Parallel Computation*, FRONTIERS'95, p. 332-341, fév. 1995.

[100] William Pugh et David G. Wonnacott, « Going Beyond Integer Programming with the Omega Test to Eliminate False Data Dependences », *IEEE Trans. on Parallel and Distributed Systems*, vol. 6, n° 2, p. 204–211, fév. 1995.

[101] Harald Devos, Kristof Beyls, Mark Christiaens, Jan Campenhout, Erik H. D'Hollander et Dirk Stroobandt, « Finding and Applying Loop Transformations for Generating Optimized FPGA Implementations », *in Trans. on High-Performance Embedded Architectures and Compilers I*, vol. 4050 *in Lecture Notes in Comptuer Science*, p. 159-178, 2007.

[102] Harald Devos, Wim Meeus et Dirk Stroob, « CLooGVHDL and JCCI », *in Proc. of the Design, Automation and Test in Europe Conference and Exhibition*, DATE'2009, 2009.

[103] Steven Derrien, Sanjay V. Rajopadhye, Patrice Quinton et Tanguy Risset, *In High-Level Synthesis : From Algorithm to Digital Circuit* , chap. High-Level Synthesis of Loops Using the Polyhedral Model. Springer, 2008.

[104] Tomofumi Yuki, *Beyond Shared Memory Loop Parallelism in the Polyhedral Model*. Thèse de doctorat, Colorado State University, Fort Collins, CO, USA, 2012.

[105] Claudiu Zissulescu, Bart Kienhuis et Ed Deprettere, « Expression Synthesis in Process Networks Generated by LAURA », *in Proc. of the 16th IEEE International Conference on Application-Specific Systems, Architecture Processors*, ASAP '05, p. 15-21, juil. 2005.

[106] Mihai Budiu, Majd Sakr, Kip Walker et SethC. Goldstein, « BitValue Inference : Detecting and Exploiting Narrow Bitwidth Computations », *in Proc. of the 6th International Euro-Par Conference on Parallel Processing*, vol. 1900 *in Lecture Notes in Computer Science*, p. 969-979, 2000.

[107] Mentor Graphics, *Algorithmic C Datatypes*, oct. 2009. En ligne : http://calypto.agranderdesign.com/ac_datatypes.php.

[108] Kalyan Muthukumar et Gautam Doshi, « Software Pipelining of Nested Loops », *in Proc. of the 10th International Conference on Compiler Construction*, CC'01, p. 165-181, 2001.

[109] Hongbo Rong, Zhizhong Tang, R. Govindarajan, Alban Douillet et Guang R. Gao, « Single-Dimension Software Pipelining for Multidimensional Loops », *ACM Trans. on Architecture and Code Optimization*, vol. 4, n° 1, p. 7:1-7:12, mars 2007.

[110] Steven Derrien, Sanjay V. Rajopadhye et Susmita Sur Kolay, « Combined Instruction and Loop Parallelism in Array Synthesis for FPGAs », *in Proc. of the 14th International Symposium on Systems Synthesis*, ISSS'01, p. 165-170, 2001.

[111] Jürgen Teich, Lothar Thiele et Lee Z. Zhang, « Partitioning Processor Arrays under Resource Constraints », *Journal of VLSI Signal Processing*, vol. 17, n° 1, p. 5-20, sept. 1997.

[112] Benjamin Ylvisaker, Carl Ebeling et Scott Hauck, « Enhanced Loop Flattening for Software Pipelining of Arbitrary Loop Nests », rap. tech., University of Washington, 2010.

[113] Christophe Alias, Bogdan Pasca et Alexandru Plesco, « Automatic Generation of FPGA-Specific Pipelined Accelerators », *in Proc. of the 7th International Conference on Reconfigurable Computing : Architectures, Tools and Applications*, ARC'11, p. 53-66, mars 2011.

[114] Claudiu Zissulescu, Bart Kienhuis et Ed Deprettere, « Increasing Pipelined IP Core Utilization in Process Networks Using Exploration », *in Proc. of the 14th International Conference on Field Programmable Logic and Application*, vol. 3203 *in Lecture Notes in Computer Science*, p. 690-699, 2004.

[115] Monica S. Lam, « Software Pipelining : An Effective Scheduling Technique for VLIW Machines », *in Proc. of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI'88, p. 318-328, 1988.

[116] Mohammed Fellahi et Albert Cohen, « Software Pipelining in Nested Loops with Prolog-Epilog Merging », *in Proc. of the International Conference High Performance Embedded Architectures and Compilers*, vol. 5409 *in Lecture Notes in Computer Science*, p. 80-94, 2009.

[117] Michael S. Schlansker et B. Ramakrishna Rau, « EPIC : Explicitly Parallel Instruction Computing », *Computer*, vol. 33, n° 2, p. 37-45, fév. 2000.

[118] Harsh Sharangpani et Ken Arora, « Itanium Processor Microarchitecture », *IEEE Micro*, vol. 20, n° 5, p. 24-43, sept. 2000.

[119] Cédric Bastoul et Paul Feautrier, « Adjusting a Program Transformation for Legality », *Parallel Processing Letters*, vol. 15, n° 1, p. 3-17, mars 2005.

[120] Nicolas Vasilache, Albert Cohen et Louis-Noël Pouchet, « Automatic Correction of Loop Transformations », *in Proc. of the 16th International Conference on Parallel Architecture and Compilation Techniques*, PACT '07, p. 292-304, 2007.

[121] Jean-Charles Naud, Daniel Ménard, Gabriel Caffarena et Olivier Sentieys, « A Discrete Model for Correlation Between Quantization Noises », *IEEE Trans. on Circuits and Systems II : Express Briefs*, vol. 59, n° 11, p. 800–804, 2012.

[122] Xiao Chenglong, *Custom Operator Identification for High-level Synthesis.* Thèse de doctorat, Unniversité de Rennes 1, France, nov. 2012.

[123] François Irigoin, Pierre Jouvelot et Rémi Triolet, « Semantical interprocedural parallelization : An overview of the PIPS project », *in Proc. of the 5th International Conference on Supercomputing*, ICS'91, p. 244-251, juin 1991.

[124] David Steinberg, Frank Budinsky, Marcelo Paternostro et Ed Merks, *EMF : Eclipse Modeling Framework 2.0.* Addison-Wesley Professional, 2e éd., 2009.

[125] Antoine Floc'h, Tomofumi Yuki, Clément Guy, Steven Derrien, Benoit Combemale, Sanjay V. Rajopadhye et Robert B. France, « Model-Driven Engineering and Optimizing Compilers : A bridge too far ? », *in Proc. of the 14th International Conference on Model Driven Engineering Languages and Systems*, MODELS'11, p. 608–622, 2011.

[126] Martin Fowler, *UML Distilled : A Brief Guide to the Standard Object Modeling Language.* Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 3e éd., 2003.

[127] Jean-Christophe Bach, Emilie Balland, Paul Brauner, Radu Kopetz, Pierre-Etienne Moreau et Antoine Reilles, « Tom Manual », rap. tech., Inria, 2009. En ligne : http://hal.inria.fr/inria-00121885/PDF/manual-2.7.pdf.

[128] DaeGon Kim, *Parameterized and Multi-level Tiled Loop Generation.* Thèse de doctorat, Colorado State University, Fort Collins, CO, USA, 2010.

[129] Albert Hartono, Muthu Manikandan Baskaran, J. Ramanujam et P. Sadayappan, « DynTile : Parametric Tiled Loop Generation for Parallel Execution on Multicore Processors », *in Proc. of the 24th IEEE International Symposium on Parallel and Distributed Processing*, IPDPS'2010, p. 1-12, avril 2010.

[130] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier et John Irwin, « Aspect-Oriented Programming », *in Object-Oriented Programming*, vol. 1241 *in Lecture Notes in Computer Science*, p. 220–242, 1997.

[131] João M.P. Cardoso, Tiago Carvalho, José G.F. Coutinho, Wayne Luk, Ricardo Nobre, Pedro Diniz et Zlatko Petrov, « LARA : An Aspect-Oriented Programming Language for Embedded Systems », *in Proc. of the 11th annual International Conference on Aspect-Oriented Software Development*, AOSD'12, p. 179-190, 2012.

[132] Toyokazu Takagi et Tsutomu Maruyama, « Accelerating HMMER search using FPGA », *in Proc. of the International Conference on Field Programmable Logic and Applications*, FPL'09, p. 332-337, 2009.

[133] John L. Hennessy et David A. Patterson, *Computer Architecture - A Quantitative Approach.* Morgan Kaufmann, 5e éd., 2012.

[134] Anders Kugler, « IMEM : An Intelligent Memory for Bump- and Reflection-Mapping », *in Proc. of the ACM SIGGRAPH/EUROGRAPHICS Workshop on Graphics Hardware*, HWWS'98, p. 113-ff., 1998.

[135] Arifur Rahman et Rafael Reif, « System-Level Performance Evaluation of Three-Dimensional Integrated Circuits », *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 8, no 6, p. 671-678, déc. 2000.

[136] Bryan Black, Murali Annavaram, Ned Brekelbaum, John DeVale, Lei Jiang, Gabriel H. Loh, Don McCaule, Pat Morrow, Donald W. Nelson, Daniel Pantuso, Paul Reed, Jeff Rupley, Sadasivan Shankar, John Shen et Clair Webb, « Die Stacking (3D) Microarchitecture », *in Proc. of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 39, p. 469-479, déc. 2006.

Résumé

Grâce aux progrès réalisés dans le domaine des semi-conducteurs, les plateformes matérielles embarquées sont capables de satisfaire les contraintes de performances d'applications de plus en plus complexes. Cette augmentation conduit à une explosion des coûts de conception, ce qui pousse les concepteurs de ces plateformes à utiliser des outils travaillant à des niveaux d'abstraction plus élevés. Aujourd'hui, les outils de synthèse de haut niveau opèrent sur des descriptions C/C++ pour en générer des accélérateurs matériels spécialisés. Ces outils offrent des gains en productivité significatifs par rapport à la génération précédente, qui opérait sur des descriptions structurelles de l'architecture en VHDL ou Verilog. Ces descriptions algorithmiques doivent être retravaillées pour que les outils puissent générer des circuits performants.

Pour faciliter cette tâche, une solution consiste à mettre en œuvre une boite à outils pour des transformations source-à-source orientées synthèse de haut niveau. En particulier, cette thèse s'intéresse aux transformations de boucles, avec pour objectif d'améliorer les performances en exposant des boucles parallèles et en améliorant la localité des accès mémoire. En nous appuyant sur une représentation des boucles dans le modèle polyédrique, nous proposons une approche qui améliore l'applicabilité du pipeline de nids de boucles en vérifiant sa légalité de manière plus précise que les approches existantes. De plus, lorsque la vérification échoue, nous proposons une technique de correction qui insère statiquement des états d'attente pour assurer la légalité du pipeline. Enfin, ce pipeline est mis en œuvre en utilisant une technique de génération de code qui met les nids de boucles à plat. Ces contributions ont été implémentées dans l'infrastructure de compilation source-à-source Gecos, avant d'être appliquées à un ensemble de benchmarks représentatifs des noyaux de calculs cibles de la synthèse de haut niveau. Les résultats montrent un gain en performances significatif, avec un surcoût en surface modéré.

Abstract

Due to the advances in semiconductor technologies, embedded hardware is capable of satisfying the performance constraints of increasingly complex applications. This leads to a design cost explosion, thus pushing the hardware designers to use tools working with higher levels of abstractions. High-Level Synthesis tools generate custom hardware accelerators out of C/C++ specifications. They offer significant productivity gains compared to the previous generation of tools that worked at the level of hardware description languages, such as VHDL or Verilog. These higher level specifications have to be reworked in order for the High-Level Synthesis tools to generate efficient hardware accelerators.

To ease this task, one solution is to provide a source-to-source transformation toolbox targeting High-Level Synthesis. Specifically, this thesis explores loop transformations in order to improve performance by exposing parallel loops and improving the locality of memory accesses. Using polyhedral representation of loop nests, we propose an approach to improve the applicability of nested loop pipelining by verifying its legality in a more precise way than existing approaches. Moreover, we propose a correction mechanism that statically inserts wait states for enforcing the pipeline legality for cases when the verification fails. The resulting pipeline is implemented using a code generation technique that flattens the loop nests. These contributions have been implemented within the GeCoS source-to-source compilation infrastructure, and applied to a set of benchmarks targeted towards High-Level Synthesis. Results show significant performance improvement at the price of a moderate area overhead.

N° d'ordre : ENSC-2013 n°451

École normale supérieure de Cachan - Antenne de Bretagne

Campus de Ker Lann - Avenue Robert Schuman - 35170 BRUZ

Tél : +33(0)2 99 05 93 00 - Fax : +33(0)2 99 05 93 29